

INTERNATIONAL STANDARD

IEEE Std 1481™



**Delay and power calculation standards –
Part 1: Integrated Circuit (IC) Open Library Architecture (OLA)**

IECNORM.COM : Click to view the full PDF of IEC 61523-1:2023



THIS PUBLICATION IS COPYRIGHT PROTECTED
Copyright © 2019 IEEE

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

IEC Secretariat
3, rue de Varembe
CH-1211 Geneva 20
Switzerland
Tel.: +41 22 919 02 11
info@iec.ch
www.iec.ch

Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue
New York, NY 10016-5997
United States of America
stds.info@ieee.org
www.ieee.org

About the IEC

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

About IEC publications

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

IEC publications search - webstore.iec.ch/advsearchform

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 It also gives information on projects, replaced and withdrawn publications.

IEC Just Published - webstore.iec.ch/justpublished

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

IEC Customer Service Centre - webstore.iec.ch/csc

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

IEC Products & Services Portal - products.iec.ch

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

Electropedia - www.electropedia.org

The world's leading online dictionary on electrotechnology, containing more than 22 300 terminological entries in English and French, with equivalent terms in 19 additional languages. Also known as the International Electrotechnical Vocabulary (IEV) online.

INTERNATIONAL IEEE Std 1481™ STANDARD



**Delay and power calculation standards –
Part 1: Integrated Circuit (IC) Open Library Architecture (OLA)**

INTERNATIONAL
ELECTROTECHNICAL
COMMISSION

ICS 25.040.01, 35.060

ISBN 978-2-8322-7539-9

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

Contents

1. Overview	30
1.1 Scope	30
1.2 Purpose	30
1.3 Introduction	30
1.4 Word usage	31
2. Normative references.....	32
3. Definitions.....	32
4. Acronyms and abbreviations	40
5. Typographical conventions	41
5.1 Syntactic elements	41
5.2 Conventions	42
6. DPCS flow.....	42
6.1 Overview	42
6.1.1 Procedural interface	44
6.1.2 Global policies and conventions	44
6.2 Flow of control	44
6.3 DPCM—application relationships	44
6.3.1 Technology library	45
6.3.2 Subrule	45
6.4 Interoperability	45
7. Delay calculation language (DCL)	45
7.1 Character set	45
7.2 Lexical elements	45
7.2.1 Whitespace.....	46
7.2.2 Comments	46
7.2.3 Tokens.....	46
7.2.4 Header names.....	56
7.2.5 Preprocessing directives.....	56
7.3 Context.....	56
7.3.1 Space.....	56
7.3.2 Plane.....	56
7.3.3 Context operation.....	56
7.3.4 Library parallelism.....	56
7.3.5 Application parallelism	57
7.4 Data types	57
7.4.1 Base types.....	57
7.4.2 Native data types.....	57
7.4.3 Mathematical calculation data types	57
7.4.4 Pointer data types.....	58
7.4.5 Aggregate data types	58
7.5 Identifiers.....	64
7.5.1 Name spaces of identifiers	64
7.5.2 Storage durations of objects	64
7.5.3 Scope of identifiers	65
7.5.4 Linkages of identifiers	66

7.6 Operator descriptions	66
7.6.1 String prefix operator	66
7.6.2 Explicit string prefix operator	66
7.6.3 Embedded string prefix operator	67
7.6.4 String prefix semantics	67
7.6.5 Assignment operator	67
7.6.6 New operator	67
7.6.7 SCOPE operator(s)	68
7.6.8 Launch operator	69
7.6.9 Purity operator	69
7.6.10 Force operator	70
7.7 Timing propagation	70
7.7.1 Timing checks	71
7.7.2 Test mode operators	71
7.8 Expressions	73
7.8.1 Array subscripting	74
7.8.2 Statement calls	74
7.8.3 General syntax	74
7.8.4 Method statement calls	74
7.8.5 Assign variable reference	75
7.8.6 Store variable reference	75
7.8.7 Mathematical expressions	75
7.8.8 Mathematical operators	76
7.8.9 Discrete math expression	77
7.8.10 INT discrete	78
7.8.11 PINLIST discrete	78
7.8.12 Logical expressions and operators	78
7.8.13 MODE expressions	78
7.8.14 Embedded C code expressions	80
7.8.15 Computation order	81
7.9 DCL mathematical statements	83
7.9.1 Statement names	83
7.9.2 Modifiers	87
7.9.3 Prototypes	89
7.9.4 Statement failure	92
7.9.5 Type definition statements	92
7.9.6 Interfacing statements	93
7.9.7 DCL to C communication	95
7.9.8 Constant statement	95
7.9.9 Calculation statements	96
7.9.10 METHOD statement	98
7.10 Predefined types	99
7.10.1 ACTIVITY_HISTORY_TYPE	99
7.10.2 HISTORY_TYPE	100
7.10.3 LOAD_HISTORY_TYPE	101
7.10.4 CELL_LIST_TYPE	102
7.10.5 TECH_TYPE	102
7.10.6 DELAY_REC_TYPE	102
7.10.7 SLEW_REC_TYPE	103
7.10.8 CHECK_REC_TYPE	103
7.10.9 CCDB_TYPE	103
7.10.10 CELL_DATA_TYPE	103
7.10.11 PCDB_TYPE	103
7.10.12 PIN_ASSOCIATION	104
7.10.13 PATH_DATA_TYPE	104
7.10.14 STD_STRUCT	104

7.11	Predefined variables.....	105
7.11.1	ARGV	105
7.11.2	CONTROL_PARM.....	105
7.12	Built-in function calls	105
7.12.1	ABS.....	106
7.12.2	Complex number components.....	106
7.12.3	EXPAND.....	106
7.12.4	Array functions.....	106
7.12.5	Messaging functions	107
7.13	Tables.....	109
7.13.1	TABLEDEF statement	109
7.13.2	Table visibility rules.....	111
7.13.3	TABLE statement.....	112
7.13.4	LOAD_TABLE statement.....	115
7.13.5	UNLOAD_TABLE statement.....	117
7.13.6	WRITE_TABLE statement	118
7.13.7	ADD_ROW statement.....	118
7.13.8	DELETE_ROW statement.....	119
7.14	Built-in library functions	120
7.14.1	Numeric conversion functions	120
7.14.2	Tech_family functions.....	122
7.14.3	Trigonometric functions.....	123
7.14.4	Context manipulation functions	124
7.14.5	Debug controls.....	126
7.14.6	Utility functions	126
7.14.7	Table functions	127
7.14.8	Subrule controls	128
7.15	Library control statements	129
7.15.1	Meta-variables.....	129
7.15.2	TECH_FAMILY.....	129
7.15.3	RULENAME	129
7.15.4	CONTROL_PARM.....	129
7.15.5	SUBRULE statement	129
7.15.6	Path list expansion rules.....	130
7.15.7	SUBRULES statement.....	131
7.15.8	Control file.....	131
7.15.9	TECH_FAMILY statement.....	133
7.15.10	SUBRULE and SUBRULES statements.....	134
7.16	Modeling.....	134
7.16.1	Types of modeling.....	134
7.16.2	Model organization	135
7.16.3	MODELPROC statement.....	136
7.16.4	SUBMODEL statement	137
7.16.5	Modeling statements	139
7.16.6	TEST_BUS statement.....	148
7.16.7	INPUT statement.....	149
7.16.8	OUTPUT statement.....	152
7.16.9	DO statement	153
7.16.10	PROPERTIES statement.....	174
7.16.11	SETVAR statement	175
7.17	Embedded C code.....	176
7.18	Definition of a subrule	176
7.19	Pragma.....	177
7.19.1	IMPORT_EXPORT_TAG.....	177

8. Power modeling and calculation.....	177
8.1 Power overview	177
8.2 Caching state information	178
8.2.1 Initializing the state cache	178
8.2.2 State cache lifetime	178
8.3 Caching load and slew information	178
8.3.1 Loading the load and slew cache	179
8.3.2 Load and slew cache lifetime	179
8.4 Simulation switching events	180
8.5 Partial swing events	180
8.6 Power calculation.....	180
8.7 Accumulation of power consumption by the design	182
8.8 Group Pin List syntax and semantics	182
8.8.1 Syntax	182
8.8.2 Semantics	183
8.8.3 Example	183
8.9 Group Condition List syntax and semantics	183
8.9.1 Syntax	184
8.9.2 Semantics	184
8.9.3 Example	184
8.10 Sensitivity list syntax and semantics.....	184
8.10.1 Syntax	185
8.10.2 Semantics	185
8.10.3 Example	185
8.11 Group condition language.....	185
8.11.1 Syntax	186
8.11.2 Semantics	186
8.11.3 Condition expression operator precedence.....	188
8.11.4 Condition expressions referencing pin states and transitions	188
8.11.5 Semantics of nonexistent pins	189
9. Application and library interaction	189
9.1 behavior model domain	189
9.2 vectorTiming and vectorPower model domains	190
9.2.1 Power unit conversion.....	190
9.2.2 Vector power calculation	190
10. Procedural interface (PI).....	191
10.1 Overview	191
10.1.1 DPCM	191
10.1.2 Application.....	192
10.1.3 libdcmr.....	192
10.2 Control and data flow	192
10.3 Architectural requirements.....	192
10.4 Data ownership technique.....	193
10.4.1 Persistence of data passed across the PI.....	193
10.4.2 Data cache guidelines for the DPCM.....	193
10.4.3 Application/DPCM interaction.....	194
10.4.4 Application initializes message/memory handling.....	194
10.4.5 Application loads and initializes the DPCM	194
10.4.6 Application requests timing models for cell instances	194
10.5 Model domain issues	194
10.5.1 Model domain selection	194
10.5.2 Model domain determination	195
10.5.3 DPCM invokes application modeling callback functions	195
10.5.4 Application requests propagation delay	195
10.5.5 DPCM calls application EXTERNAL functions.....	196
10.6 Reentry requirements.....	196

10.7 Application responsibilities when using a DPCM	196
10.7.1 Standard Structure rules	196
10.7.2 User object registration	197
10.7.3 Selection of early and late slew values.....	197
10.7.4 Semantics of slew values	198
10.7.5 Slew calculations	198
10.8 Application use of the DPCM	198
10.8.1 Initialization of the DPCM.....	198
10.8.2 Context creation	199
10.8.3 Dynamic linking.....	199
10.8.4 Subrule initialization	200
10.8.5 Use of the DPCM.....	201
10.8.6 Application control.....	201
10.8.7 Application execution	201
10.8.8 Termination of DPCM.....	201
10.9 DPCM library organization.....	202
10.9.1 Multiple technologies.....	202
10.9.2 Model names.....	202
10.9.3 DPCM error handling.....	202
10.10 C level language for EXPOSE and EXTERNAL functions.....	203
10.10.1 Integer return code	203
10.10.2 The Standard Structure pointer.....	203
10.10.3 Result structure pointer	203
10.10.4 Passed arguments	203
10.10.5 DCL array indexing.....	204
10.10.6 Conversion to C data types	204
10.10.7 include files.....	204
10.11 PIN and BLOCK data structure requirements.....	205
10.12 DCM_STD_STRUCT Standard Structure.....	206
10.12.1 Alternate semantics for Standard Structure fields	208
10.12.2 Reserved fields.....	209
10.12.3 Standard Structure value restriction	209
10.13 DCMTransmittedInfo structure.....	209
10.14 Environment or user variables	209
10.15 Procedural interface (PI) functions summary	209
10.15.1 Expose functions.....	209
10.15.2 External functions.....	217
10.15.3 Deprecated functions.....	220
10.16 Implicit functions.....	222
10.16.1 libdcmlr.....	222
10.16.2 Run-time library utility functions.....	223
10.16.3 Memory control functions.....	223
10.16.4 Message and error control functions	224
10.16.5 Calculation functions	225
10.16.6 Modeling functions	225
10.17 PI function table description	225
10.17.1 Arguments	226
10.17.2 DCL syntax	227
10.17.3 C syntax	227

10.18 PI function descriptions	227
10.18.1 Interconnect loading related functions	227
10.18.2 Interconnect delay related functions	234
10.18.3 Functions accessing netlist information	237
10.18.4 Functions exporting limit information	246
10.18.5 Functions getting/setting model information	247
10.18.6 Functions importing instance name information	260
10.18.7 Process information functions	263
10.18.8 Miscellaneous standard interface functions	265
10.18.9 Power-related functions	274
10.19 Application context	282
10.19.1 pathData association	282
10.20 Application and library interaction	282
10.20.1 behavior model domain	283
10.20.2 vectorTiming and vectorPower model domains	283
10.20.3 Power unit conversion	284
10.20.4 Vector power calculation	284
10.21 Parasitic analysis	285
10.21.1 Assumptions	285
10.21.2 Parasitic networks	285
10.21.3 Basic definitions	285
10.21.4 Parasitic element data structure	287
10.21.5 Coordinates	291
10.21.6 Parasitic subnets	291
10.21.7 Pin parasitics	298
10.21.8 Modeling internal nodes	301
10.21.9 Load and interconnect models	303
10.21.10 Obtaining parasitic networks	307
10.21.11 Persistent storage of load and interconnect models	308
10.21.12 Calculating effective capacitances and driving resistances	311
10.21.13 Parasitic estimation	314
10.21.14 Threshold voltages	319
10.21.15 Obtaining aggressor window overlaps	320
10.22 Noise analysis	327
10.22.1 Types of noise	328
10.22.2 Noise models	329
10.22.3 Noise waveforms	332
10.22.4 Noise network models	338
10.22.5 Calculating composite noise at cell inputs	344
10.22.6 Calculating composite noise at cell outputs	347
10.22.7 Setting noise budgets	351
10.22.8 Reporting noise violations	352
10.23 Delay and slew calculations for differential circuits	354
10.23.1 Sample figures	354
10.23.2 appGetArrivalOffsetsByName	356
10.23.3 API extensions for function modeling	357
10.23.4 Explicit APIs for user-defined primitives	364
10.23.5 APIs for hierarchy	366
10.23.6 Built-in APIs for function modeling	366
10.23.7 API Extensions for VECTOR modeling	367
10.23.8 APIs for XWF	368
10.23.9 Extensions and changes to voltages and temperature APIs	372
10.23.10 Operating conditions	374
10.23.11 On-chip process variation	377
10.23.12 Accessing properties and attributes	383
10.23.13 APIs for attribute within a PIN object	404

10.23.14 Connectivity	412
10.23.15 Control of timing arc existence and state	414
10.23.16 Modeling cores.....	419
10.23.17 Default pin slews and interface version calls	424
10.23.18 API to access library required resources	426
10.23.19 Resource types	427
10.23.20 Library extensions for phase locked loop processing.....	428
10.23.21 API definitions for external conditions	429
10.23.22 Extensions for listing pins.....	433
10.23.23 Memory BIST mapping	434
10.23.24 dpcmGetCellTestProcedure.....	436
10.24 Interconnect delay calculation intraface	437
10.24.1 Control and data flows	437
10.24.2 Model generation functions.....	438
10.24.3 Calculation functions	440
10.24.4 Cell calculation functions.....	441
10.24.5 ICM initialization	446
10.25 DCL run-time support.....	453
10.25.1 Array manipulation functions.....	453
10.25.2 Memory management	457
10.25.3 Structure manipulation functions	457
10.25.4 Initialization functions	462
10.26 Calculation functions	473
10.26.1 delay	473
10.26.2 slew	474
10.26.3 check	474
10.27 Modeling functions	476
10.27.1 modelSearch.....	476
10.27.2 Mode operators	478
10.27.3 Arrival time merging.....	479
10.27.4 Edge propagation communication to the application	480
10.27.5 Edge propagation communication to the DPCM	482
10.27.6 newTimingPin	483
10.27.7 newDelayMatrixRow	483
10.27.8 newNetSinkPropagateSegments	484
10.27.9 newNetSourcePropagateSegments.....	486
10.27.10 newPropagateSegment	487
10.27.11 newTestMatrixRow	488
10.27.12 newAltTestSegment	489
10.27.13 Interactions between interconnect modeling and modeling functions	489
10.28 Deprecated functions	490
10.28.1 Parasitic handling.....	490
10.28.2 Array manipulation functions.....	499
10.28.3 Memory management	501
10.28.4 Initialization functions	504
10.29 Standard Structure (std_stru.h) file.....	515
10.30 Standard macros (std_macs.h) file	535
10.31 Standard interface structures (dcmintf.h) file	543
10.32 Standard loading (dcmload.h) file	547
10.33 Standard debug (dcmdebug.h) file	551
10.34 Standard array (dcmgarray.h) file	578
10.35 Standard user array defines (dcmuarray.h) file.....	582
10.36 Standard platform-dependency (dcmpltfm.h) file	586
10.37 Standard state variables (dcmstate.h) file.....	592

11. Parasitics	595
11.1 Introduction	596
11.2 Targeted applications for SPEF	596
11.3 SPEF specification	596
11.3.1 Grammar	596
11.3.2 Escaping rules	598
11.3.3 File syntax	598
11.3.4 Comments	604
11.3.5 File semantics	604
11.4 Examples	623
11.4.1 Basic *D_NET file	623
11.4.2 Basic *R_NET file	626
11.4.3 *R_NET with poles and residues plus name mapping	627
11.4.4 *D_NET with triplet par_value	628
11.4.5 *R_NET with poles and residues plus triplet par_value	631
11.4.6 Merging SPEF files	633
11.4.7 A SPEF file header section with *VARIATION_PARAMETERS definition	637
11.4.8 CAP and RES statements with sensitivity information in a SPEF file	637
Annex A (informative) Implementation requirements	638
Annex B (informative) Participants	640

Figures

Figure 1—High-level DPCS architecture linkage structure	43
Figure 2—Function graph form.....	135
Figure 3—DPCM/application procedural interface.....	192
Figure 4—PIN and PINLIST.....	205
Figure 5—Parallel drivers example.....	239
Figure 6—Subnet node mapping.....	294
Figure 7—Differential buffer chain.....	354
Figure 8—Timing models for a differential buffer chain	355
Figure 9—Arrival offset for differential signals.....	355
Figure 10—Priority operation	360
Figure 11—Precedence.....	361
Figure 12—Strand ranges.....	362
Figure 13—Various methods of using XWF to model waveforms for slew computation.....	368
Figure 14—Propagation of XWF “handles” by application	369
Figure 15—Application, CCM, and ICM control and data flows	438
Figure 16—Clock separation.....	475
Figure 17—Bias calculation.....	476
Figure 18—Clock pulse width.....	476
Figure 19—Sample MODELPROC results.....	486
Figure 20—Additional MODELPROC results.....	486
Figure 21—Capacitance value example.....	491
Figure 22—Equation for poles and residues	492
Figure 23—Example RC network	498
Figure 24—SPEF targeted applications.....	596

IECNORM.COM : Click to view the full PDF of IEC 61523-1:2023

Tables

Table 1—Keywords.....	46
Table 2—DCL predefined references to Standard Structure fields.....	49
Table 3—DCL compiler generated predefined identifiers.....	52
Table 4—Edge types and conversions.....	54
Table 5—Propagation mode conversions.....	54
Table 6—Calculation mode conversions.....	54
Table 7—TEST_TYPE conversions.....	55
Table 8—Purity operator.....	70
Table 9—Timing resolution modes.....	70
Table 10—Test mode operators table.....	71
Table 11—Mathematical operators.....	77
Table 12—Logical operators.....	78
Table 13—Mathematical operator precedence (high to low).....	81
Table 14—Logical operator precedence (high to low).....	81
Table 15—Type definition for ACTIVE_HISTORY_TYPE.....	99
Table 16—Permitted activityCode values.....	100
Table 17—Type definition for HISTORY_TYPE.....	100
Table 18—Rule history info message types.....	101
Table 19—Table History inform message types.....	101
Table 20—Permitted kind values.....	101
Table 21—LOAD_HISTORY_TYPE.....	101
Table 22—CELL_LIST_TYPE.....	102
Table 23—TECH_TYPE.....	102
Table 24—DELAY_REC_TYPE.....	102
Table 25—SLEW_REC_TYPE.....	103
Table 26—CHECK_REC_TYPE.....	103
Table 27—CCDB_TYPE.....	103
Table 28—CELL_DATA_TYPE.....	103
Table 29—CCDB_TYPE.....	104
Table 30—PIN_ASSOCIATION.....	104
Table 31—PATH_DATA_TYPE.....	104
Table 32—STD_STRUCT.....	105
Table 33—ARGV.....	105
Table 34—CONTROL_PARM.....	105
Table 35—Library function floor.....	121
Table 36—Library function ifloor.....	121
Table 37—Library function ceil.....	121
Table 38—Library function iceil.....	121
Table 39—Library function rint.....	121
Table 40—Library function round.....	122
Table 41—Library function trunc.....	122
Table 42—Library function itrunc.....	122
Table 43—Library function map_tech_family.....	122
Table 44—Library function current_tech_type.....	122
Table 45—Library function subrule_tech_type.....	123
Table 46—Library function subrule_tech_type.....	123
Table 47—Library function get_technology_list.....	123
Table 48—Library function cos.....	123
Table 49—Library function sin.....	123
Table 50—Library function tan.....	124
Table 51—Library function new_plane.....	124
Table 52—Library function get_plane_name.....	124
Table 53—Library function get_space_name.....	124
Table 54—Library function get_max_spaces.....	125

Table 55—Library function <code>get_max_planes</code>	125
Table 56—Library function <code>get_space_coordinate</code>	125
Table 57—Library function <code>get_plane_coordinate</code>	125
Table 58—Library function <code>set_busy_wait</code>	126
Table 59—Library function <code>change_debug_level</code>	126
Table 60—Library function <code>get_caller_stack</code>	126
Table 61—Library function <code>GET_LOAD_HISTORY</code>	126
Table 62—Library function <code>GET_CELL_LIST</code>	127
Table 63—Library function <code>GET_ROW_COUNT</code>	127
Table 64—Library function <code>STEP_TABLE</code>	127
Table 65—Library function <code>GET_LOAD_PATH</code>	128
Table 66—Library function <code>GET_RULE_NAME</code>	128
Table 67—Library function <code>ADD_RULE</code>	128
Table 68—Data type clause	143
Table 69—Arc data types	144
Table 70—Validity of predefined identifiers for <code>STORE</code> clause	152
Table 71—Logic operators (valid for behavior, vectorTiming, and vectorPower model domains)	157
Table 72—Logical equivalence operators (valid for behavior model domain)	157
Table 73—Unary bitwise operators (valid for behavior, vectorTiming, and vectorPower model domains)	157
Table 74—Binary bitwise operators (valid for behavior, vectorTiming, and vectorPower model domains)	158
Table 75—Binary operators (valid for behavior model domain)	158
Table 76—Node primitives for control operators (valid for behavior model domain)	158
Table 77—Node primitives for edge operators (continued) (valid for behavior, vectorTiming, and vectorPower model domains)	159
Table 78—Node primitives for precedence control operators (valid for behavior model domain)	159
Table 79—Node primitives for constant operators (valid for behavior, vectorTiming, vectorPower model domains)	160
Table 80—Node primitives for user-defined operators	160
Table 81—Node primitives for miscellaneous operators	160
Table 82—Binary reduction operators	161
Table 83—Bitwise reduction operators	162
Table 84—Logical reduction operators	163
Table 85—Array of bits operators	164
Table 86—Edge operators	165
Table 87—Higher function nodes	166
Table 88—Constant value nodes	168
Table 89—Miscellaneous operators	169
Table 90—User-defined operators	169
Table 91—Valid modifier enumerations for given node primitive operators	170
Table 92—Syntax for a <code>GroupPinString</code>	183
Table 93—Syntax for a <code>GroupConditionString</code>	184
Table 94—Syntax for a <code>SensitivityPinString</code>	185
Table 95—Syntax for a <code>condition_expression</code>	186
Table 96— <code>PinName_Identifier</code> semantics	187
Table 97— <code>PinName_Level</code> semantics	187
Table 98— <code>PinName_State</code> semantics	187
Table 99—Condition expression operators	188
Table 100—Interaction between multiple technologies and application	202
Table 101—Return code most significant byte	203
Table 102—Return code least significant bytes	203
Table 103—Data types defined in DCL and C	204
Table 104—Header files	205
Table 105—Predefined macro names	206
Table 106—Alternate semantics for Standard Structure fields	208

Table 107—Expose functions.....	209
Table 108—External functions.....	217
Table 109—Deprecated functions.....	220
Table 110—libdcmlr functions.....	222
Table 111—Module control functions.....	223
Table 112—Memory control functions.....	223
Table 113—Message and error control functions.....	224
Table 114—Calculation functions.....	225
Table 115—Modeling functions.....	225
Table 116—PI function table example.....	226
Table 117—Standard Structure field semantics.....	226
Table 118—appGetTotalLoadCapacitanceByPin.....	227
Table 119—appGetTotalLoadCapacitanceByName.....	228
Table 120—appGetTotalPinCapacitanceByPin.....	228
Table 121—appGetTotalPinCapacitanceByName.....	229
Table 122—appGetSourcePinCapacitanceByPin.....	229
Table 123—appGetSourcePinCapacitanceByName.....	230
Table 124—dpcmGetDefCellSize.....	230
Table 125—appGetCellCoordinates.....	230
Table 126—appGetCellOrientation.....	231
Table 127—dpcmGetEstLoadCapacitance.....	232
Table 128—dpcmGetEstWireCapacitance.....	232
Table 129—dpcmGetEstWireResistance.....	233
Table 130—dpcmGetPinCapacitance.....	233
Table 131—dpcmGetCellIOlists.....	234
Table 132—appGetRC.....	234
Table 133—dpcmGetDelayGradient.....	235
Table 134—dpcmGetSlewGradient.....	235
Table 135—dpcmGetEstimateRC.....	236
Table 136—dpcmGetDefPortSlew.....	236
Table 137—dpcmGetDefPortCapacitance.....	237
Table 138—appGetNumDriversByPin.....	237
Table 139—appGetNumDriversByName.....	237
Table 140—appForEachParallelDriverByPin.....	238
Table 141—appForEachParallelDriverByName.....	240
Table 142—appGetNumPinsByPin.....	241
Table 143—appGetNumPinsByName.....	242
Table 144—appGetNumSinksByPin.....	242
Table 145—appGetNumSinksByName.....	242
Table 146—dpcmAddWireLoadModel.....	243
Table 147—dpcmGetWireLoadModel.....	244
Table 148—dpcmGetWireLoadModelForBlockSize.....	245
Table 149—appGetInstanceCount.....	245
Table 150—dpcmGetCapacitanceLimit.....	246
Table 151—dpcmGetSlewLimit.....	246
Table 152—dpcmGetXovers.....	247
Table 153—dpcmGetFunctionalModeArray.....	247
Table 154—dpcmGetBaseFunctionalMode.....	248
Table 155—appGetCurrentFunctionalMode.....	249
Table 156—dpcmGetControlExistence.....	249
Table 157—dpcmSetLevel.....	250
Table 158—dpcmGetLibraryAccuracyLevelArrays.....	252
Table 159—dpcmSetLibraryAccuracyLevel.....	252
Table 160—dpcmGetExposePurityAndConsistency.....	253
Table 161—DCM_Purity.....	253
Table 162—DCM_Consistency.....	254

Table 163—dpcmGetRailVoltageArray.....	254
Table 164—dpcmGetBaseRailVoltage.....	254
Table 165—appGetCurrentRailVoltage.....	255
Table 166—dpcmGetWireLoadModelArray.....	255
Table 167—dpcmGetBaseWireLoadModel.....	256
Table 168—appGetCurrentWireLoadModel.....	256
Table 169—dpcmGetBaseTemperature.....	257
Table 170—dpcmGetBaseOpRange.....	257
Table 171—dpcmGetOpRangeArray.....	258
Table 172—appGetCurrentTemperature.....	258
Table 173—appGetCurrentOpRange.....	259
Table 174—dpcmGetTimingStateArray.....	259
Table 175—appGetCurrentTimingState.....	260
Table 176—dpcmGetCellList.....	261
Table 177—appGetCellName.....	262
Table 178—appGetHierPinName.....	262
Table 179—appGetHierBlockName.....	263
Table 180—appGetHierNetName.....	263
Table 181—dpcmGetThresholds.....	264
Table 182—appGetThresholds.....	264
Table 183—appGetExternalStatus.....	265
Table 184—appGetVersionInfo.....	265
Table 185—appGetResource.....	266
Table 186—dpcmGetRuleUnitToSeconds.....	266
Table 187—dpcmGetRuleUnitToOhms.....	267
Table 188—dpcmGetRuleUnitToFarads.....	267
Table 189—dpcmGetRuleUnitToHenries.....	268
Table 190—dpcmGetRuleUnitToWatts.....	268
Table 191—dpcmGetRuleUnitToJoules.....	269
Table 192—dpcmGetTimeResolution.....	269
Table 193—dpcmGetParasiticCoordinateTypes.....	270
Table 194—dpcmIsSlewTime.....	270
Table 195—dpcmDebug.....	271
Table 196—dpcmGetVersionInfo.....	271
Table 197—dpcmHoldControl.....	272
Table 198—dpcmFillPinCache.....	272
Table 199—dpcmFreePinCache.....	273
Table 200—appRegisterCellInfo.....	273
Table 201—dpcmGetCellPowerInfo.....	274
Table 202—dpcmGetCellPowerWithState.....	275
Table 203—dpcmGetAETCellPowerWithSensitivity.....	276
Table 204—Integer LSB example.....	276
Table 205—Mask encoding.....	276
Table 206—dpcmGetPinPower.....	277
Table 207—dpcmAETGetSettlingTime.....	277
Table 208—dpcmAETGetSimultaneousSwitchTime.....	278
Table 209—dpcmGroupGetSettlingTime.....	278
Table 210—dpcmGroupGetSimultaneousSwitchTime.....	279
Table 211—dpcmCalcPartialSwingEnergy.....	279
Table 212—dpcmSetInitialState.....	280
Table 213—dpcmFreeStateCache.....	281
Table 214—appGetStateCache.....	281
Table 215—dpcmGetNetEnergy.....	282
Table 216—parasiticElement structure.....	287
Table 217—Parasitic element variables.....	288
Table 218—DCM_ElementTypes.....	289

Table 219—Node variables	289
Table 220—Value variables.....	290
Table 221—Coordinate structure.....	291
Table 222—parasiticSubnet structure.....	291
Table 223—DCM_NodeTypes	292
Table 224—dpcmCreateSubnetStructure	295
Table 225—dpcmGetDefaultInterconnectTechnology	297
Table 226—dpcmScaleParasitics.....	297
Table 227—dpcmGetSinkPinParasitics.....	300
Table 228 Table 228—dpcmGetSourcePinParasitics	300
Table 229—dpcmGetPortNames	301
Table 230—Application timing arcs.....	302
Table 231—dpcmBuildLoadModels	304
Table 232—dpcmBuildInterconnectModels.....	305
Table 233—appGetInterconnectModels.....	305
Table 234—appGetLoadModels.....	306
Table 235—appGetParasiticNetworksByPin.....	307
Table 236—appGetParasiticNetworksByName	308
Table 237—dpcmPassivateLoadModels	309
Table 238—dpcmPassivateInterconnectModels.....	309
Table 239—dpcmRestoreLoadModels	310
Table 240—dpcmRestoreInterconnectModels	310
Table 241—appGetCeff.....	311
Table 242—dpcmCalcCeff	312
Table 243—dpcmCalcSteadyStateResistanceRange	312
Table 244—dpcmCalcTristateResistanceRange	313
Table 245—appSetCeff	314
Table 246—dpcmCalcCouplingCapacitance.....	315
Table 247—dpcmCalcSubstrateCapacitance.....	316
Table 248—dpcmCalcSegmentResistance	317
Table 249—Manufacturing layer type values.....	317
Table 250—dpcmGetLayerArray	318
Table 251—dpcmGetRuleUnitToMeters.....	318
Table 252—dpcmGetRuleUnitToAmps	319
Table 253—appGetDriverThresholds.....	319
Table 254—appGetAggressorOverlapWindows.....	320
Table 255—appSetAggressorInteractWindows.....	322
Table 256—appGetOverlapNWFs.....	324
Table 257—appSetDriverInteractWindows.....	325
Table 258—dpcmCalcOutputResistances	326
Table 259—DCM_NoiseTypes	328
Table 260—docmGetLibraryNoiseTypesArray.....	329
Table 261—appNewNoiseCone	331
Table 262—NWF type	333
Table 263—PWF type	334
Table 264—PWFdriverModel	335
Table 265—dpcmGetPWFarray	335
Table 266—dpcmCreatePWF	336
Table 267—dpcmCopyNWFarray.....	336
Table 268—dpcmCopyPWF.....	337
Table 269—dpcmCreatePWFdriverModel.....	337
Table 270—dpcmGetPWFdriverModelArray	338
Table 271—dpcmGetSinkPinNoiseParasitics.....	341
Table 272—dpcmGetSourcePinNoiseParasitics.....	341
Table 273—dpcmBuildNoiseInterconnectModels.....	342
Table 274—dpcmBuildNoiseLoadModels	343

Table 275—driverPinNoise	344
Table 276—dpcmCalcInputNoise	346
Table 277—relatedPinNoise	347
Table 278—dpcmCalcOutputNoise	349
Table 279—appForEachNoiseParallelDriver	350
Table 280—dpcmSetParallelRelatedNoise	351
Table 281—appSetParallelOutputNoise	351
Table 282—dpcmSetNoiseLimit	352
Table 283—noiseViolationInfo	352
Table 284—appSetNoiseViolation	353
Table 285—dpcmGetNoiseViolationDetails	354
Table 286—appGetArrivalOffsetsByName	356
Table 287—appGetArrivalOffsetArraysByName	356
Table 288—Arc ordering	359
Table 289—PathDataBlock->modifiers enumeration values for priority operation	360
Table 290—DCM_TestTypes	364
Table 291—dpcmPerformPrimitive	364
Table 292—appGetArcStructure	365
Table 293—dpcmGetNodeSensitivity	365
Table 294—dpcmModelMoreFunctionDetail	366
Table 296—appSetXWF	370
Table 297—appGetXWF	371
Table 298—dpcmCalcXWF	372
Table 299—dpcmGetCellRailVoltageArray	373
Table 300—dpcmGetBaseCellRailVoltageArray	373
Table 301—dpcmGetBaseCellTemperature	374
Table 302—dpcmGetOpPointArray	375
Table 303—dpcmGetBaseOpPoint	376
Table 304—dpcmSetCurrentOpPoint	376
Table 305—DCM_ProcessVariations	377
Table 306—New predefined identifiers	378
Table 307—DCM_CalculationModes	379
Table 308—dpcmSetCurrentProcessPoint	379
Table 309—dpcmGetBaseProcessPoint	380
Table 310—dpcmGetProcessPointRange	380
Table 311—dpcmGetRailVoltageRangeArray	381
Table 312—dpcmGetCellRailVoltageRangeArray	382
Table 313—dpcmGetTemperatureRange	382
Table 314—dpcmGetCellTemperatureRange	383
Table 315—dpcmGetPinPinTypeArray	384
Table 316—dpcmGetPinPinType	385
Table 317—dpcmGetPinSignalTypeArray	385
Table 318—dpcmGetPinSignalType	387
Table 319—dpcmGetPinActionArray	387
Table 320—dpcmGetPinAction	388
Table 321—dpcmGetPinPolarityArray	388
Table 322—dpcmGetPinPolarity	389
Table 323—dpcmGetPinEnablePin	389
Table 324—dpcmGetPinConnectClass	390
Table 325—dpcmGetPinScanPosition	390
Table 326—dpcmGetPinStuckArray	391
Table 327—dpcmGetPinStuck	391
Table 328—dpcmGetDifferentialPairPin	392
Table 329—dpcmGetPathLabel	392
Table 330—dpcmGetPowerStateLabel	393
Table 331—dpcmGetCellTypeArray	393

Table 332—dpcmGetCellType	395
Table 333—dpcmGetCellSwapClassArray	395
Table 334—dpcmGetCellSwapClass	396
Table 335—dpcmGetCellRestrictClassArray	396
Table 336—dpcmGetCellRestrictClass	398
Table 337—dpcmGetCellScanTypeArray	398
Table 338—dpcmGetCellScanType	399
Table 339—dpcmGetCellNonScanCell	399
Table 340—DCM_PinMappingTypes	401
Table 341—appSetVectorOperations	401
Table 342—DCM_VectorOperations	402
Table 343—dpcmGetLevelShifter	403
Table 344—dpcmGetPinTiePolarity	404
Table 345—dpcmGetPinReadPolarity	404
Table 346—dpcmGetPinWritePolarity	405
Table 347—dpcmGetSimultaneousSwitchTimes	405
Table 348—appGetSwitchingBits	406
Table 349—dpcmGetFrequencyLimit	407
Table 350—appGetPinFrequency	407
Table 351—dpcmGetBasePinFrequency	408
Table 352—dpcmGetPinJitter	408
Table 353—dpcmGetInductanceLimit	409
Table 354—dpcmGetOutputSourceResistances	409
Table 355—appSetPull	410
Table 356—DCM_PullType	410
Table 357—dpcmGetPull	410
Table 358—dpcmGetPinDriveStrength	411
Table 359—dpcmGetCellVectorPower	412
Table 360—dpcmGetPinConnectivityArrays	412
Table 361—dpcmGetLibraryConnectClassArray	413
Table 362—dpcmGetLibraryConnectivityRules	413
Table 363—DCM_ConnectRules	414
Table 364—dpcmGetExistenceGraph	415
Table 365—dpcmGetTimingStateGraphs	416
Table 366—dpcmGetTimingStateStrings	418
Table 367—dpcmGetVectorEdgeNumbers	419
Table 368—appSetSignalDivision	420
Table 369—appSetSignalMultiplication	421
Table 370—appSetSignalGeneration	423
Table 371—dpcmGetDefPinSlews	424
Table 372—appGetInterfaceVersion	425
Table 373—Valid interface version strings	425
Table 374—dpcmSetResource	426
Table 375—dpcmGetAllResources	426
Table 376—DCM_ResourceTypes	428
Table 377—appGetExternalDelayByPin	429
Table 378—Description of multiArcMultiEdgePath (XXXX are do not care bits)	430
Table 379—appGetExternalDelayByName	431
Table 380—appGetLogicLevelByName	432
Table 381 Table 381—DCM_LogicLevel	432
Table 382—appGetLogicLevelByPin	433
Table 383—dpcmGetPinIndexArrays	433
Table 384—dpcmGetSupplyPins	434
Table 385—DCM_BistInversion	435
Table 386—dpcmGetPhysicalBISTMap	435
Table 387—dpcmGetLogicalBISTMap	436

Table 388—dpcmGetCellTestProcedure	436
Table 389—icmBuildLoadModels	438
Table 390—icmBuildInterconnectModels	439
Table 391—icmCalcInterconnectDelaySlew	440
Table 392—icmCalcCellDelaySlew	442
Table 393—ccmCalcDelaySlew	443
Table 394—ccmEarlyLateIdentical	443
Table 395—ccmGetICMcontrolParams	444
Table 396—icmCalcOutputResistances	444
Table 397—icmCalcTotalLoadCapacitances	445
Table 398—icmCalcXWF	446
Table 399—icmInit	447
Table 400—dcmRT_new_DCM_ARRAY	454
Table 401—DCM_ATYPE enumeration	454
Table 402—DCM_AINIT enumeration	455
Table 403—DCM_ArrayInitUserFunction	456
Table 404—dcmRT_sizeof_DCM_ARRAY	456
Table 405—dcmRT_claim_DCM_ARRAY	456
Table 406—dcmRT_disclaim_DCM_ARRAY	457
Table 407—dcmRT_disclaim_DCM_STRUCT	458
Table 408—dcmRT_disclaim_DCM_STRUCT	458
Table 409—dcmRT_longlock_DCM_STRUCT	459
Table 410—dcmRT_longunlock_DCM_STRUCT	459
Table 411—dcmRT_getNumDimensions	460
Table 412—dcmRT_getNumElementsPer	460
Table 413—dcmRT_getNumElements	461
Table 414—dcmRT_getElementType	461
Table 415—dcmRT_arraycmp	461
Table 416—dcmRT_InitRuleSystem	462
Table 417—dcmRT_BindRule	463
Table 418—dcmRT_AppendRule	464
Table 419—dcmRT_UnbindRule	465
Table 420—dcmRT_FindFunction	466
Table 421—dcmRT_FindAppFunction	466
Table 422—dcmRT_QuietFindFunction	467
Table 423—dcmRT_MakeRC	467
Table 424—dcmRT_HardErrorRC	467
Table 425—dcmRT_SetMessageIntercept	468
Table 426—dcmRT_IssueMessage	468
Table 427—dcmRT_new_DCM_STD_STRUCT	469
Table 428—dcmRT_delete_DCM_STD_STRUCT	469
Table 429—dcmRT_setTechnology	470
Table 430—dcmRT_getTechnology	470
Table 431—dcmRT_getAllTechs	471
Table 432—dcmRT_freeAllTechs	471
Table 433—dcmRT_isGeneric	471
Table 434—dcmRT_takeMappingOfNugget	472
Table 435—dcmRT_registerUserObject	472
Table 436—dcmRT_DeleteRegisteredUserObjects	472
Table 437—dcmRT_DeleteOneUserObject	473
Table 438—delay	473
Table 439—slew	474
Table 440—check	474
Table 441—modelSearch	477
Table 442—Mode propagation operators	478
Table 443—Mode computation operators for delay and slew	479

Table 444—Mode operator enumerators for check	479
Table 445—Edge propagation enumeration pairs	480
Table 446—Edge propagation communication with DPCM	483
Table 447—newTimingPin	483
Table 448—newDelayMatrixRow	484
Table 449—newNetSinkPropagateSegments	485
Table 450—newNetSourcePropagateSegments	487
Table 451—newPropagateSegment	488
Table 452—newTestMatrixRow	488
Table 453—newAltTestSegment	489
Table 454—appGetPiModel	490
Table 455—appGetPolesAndResidues	491
Table 456—appGetCeffective	492
Table 457—appGetRLCnetworkByPin	493
Table 458—appGetRLCnetworkByName	494
Table 459—dpcmCalcPiModel	494
Table 460—dpcmCalcPolesAndResidues	495
Table 461—dpcmCalcCeffective	496
Table 462—dpcmSetRLCmember	496
Table 463—dpcmAppendPinAdmittance	498
Table 464—dpcmDeleteRLCnetwork	499
Table 465—dcm_copy_DCM_ARRAY	499
Table 466—dcm_new_DCM_ARRAY	500
Table 467—dcm_sizeof_DCM_ARRAY	500
Table 468—dcm_lock_DCM_ARRAY	500
Table 469—dcm_unlock_DCM_ARRAY	501
Table 470—dcm_lock_DCM_STRUCT	501
Table 471—dcm_unlock_DCM_STRUCT	502
Table 472—dcm_getNumDimensions	502
Table 473—dcm_getNumElementsPer	502
Table 474—dcm_getNumElements	503
Table 475—dcm_getElementType	503
Table 476—dcm_arraycmp	503
Table 477—dcmCellList	504
Table 478—dcmSetNewStorageManager	504
Table 479—dcmMalloc	505
Table 480—dcmFree	505
Table 481—dcmRealloc	505
Table 482—dcmBindRule	506
Table 483—dcmAddRule	506
Table 484—dcmUnbindRule	507
Table 485—dcmFindFunction	507
Table 486—dcmFindAppFunction	507
Table 487—dcmQuietFindFunction	508
Table 488—dcmMakeRC	508
Table 489—dcmHardErrorRC	509
Table 490—dcmSetMessageIntercept	509
Table 491—dcmIssueMessage	509
Table 492—dcm_rule_init	510
Table 493—DCM_new_DCM_STD_STRUCT	511
Table 494—DCM_delete_DCM_STD_STRUCT	511
Table 495—dcm_setTechnology	512
Table 496—dcm_getTechnology	512
Table 497—dcm_getAllTechs	513
Table 498—dcm_freeAllTechs	513
Table 499—dcm_isGeneric	513

Table 500—dcm_mapNugget.....	514
Table 501—dcm_takeMappingOfNugget.....	514
Table 502—dcm_registerUserObject	514
Table 503—dcm_DeleteRegisteredUserObjects	515
Table 504—dcm_DeleteOneUserObject	515
Table 505—Design flow values	605
Table 506—conn_attr	609
Table 507—Variation effect equations	612

IECNORM.COM : Click to view the full PDF of IEC 61523-1:2023

BNF Syntax

Syntax 7.1—token	46
Syntax 7.2—identifier	49
Syntax 7.3—double_quoted_character	49
Syntax 7.4—constant	53
Syntax 7.5—string_literal	55
Syntax 7.6—operator	55
Syntax 7.7—punctuator	55
Syntax 7.8—native_type	57
Syntax 7.9—mathematical_type	58
Syntax 7.10—pointer_data_type	58
Syntax 7.11—aggregate_type	59
Syntax 7.12—aggregate_access	59
Syntax 7.13—array_type	61
Syntax 7.14—var	62
Syntax 7.15—cast	64
Syntax 7.16—new_operator	67
Syntax 7.17—scope_change	69
Syntax 7.18—launch	69
Syntax 7.19—FORCE operator	70
Syntax 7.20—array_index	74
Syntax 7.21—statement_call	74
Syntax 7.22—method_statement_call	75
Syntax 7.23—assign_variable_reference	75
Syntax 7.24—store_variable_reference	75
Syntax 7.25—expression	76
Syntax 7.26—discrete_expression	77
Syntax 7.27—logical_expression	78
Syntax 7.28—pin_range_list	79
Syntax 7.29—c_statement_reference	81
Syntax 7.30—passed_clause	84
Syntax 7.31—result_prototype	84
Syntax 7.32—conditional_result	85
Syntax 7.33—local_clause	86
Syntax 7.34—default_clause	87
Syntax 7.35—default_clause (result variable)	87
Syntax 7.36—prototype_modifier	90
Syntax 7.37—common_prototype	90
Syntax 7.38—tabledef_prototype	91
Syntax 7.39—load_table_prototype	91
Syntax 7.40—add_row_prototype	91
Syntax 7.41—delay_prototype	91
Syntax 7.42—check_prototype	91
Syntax 7.43—submodel_prototype	92
Syntax 7.44—typedef	92
Syntax 7.45—expose_statement	93
Syntax 7.46—external_statement	94
Syntax 7.47—internal_statement	94
Syntax 7.48—constant_statement	95
Syntax 7.49—calculation_body	96
Syntax 7.50—calc_statement	96
Syntax 7.51—assign_statement	96
Syntax 7.52—delay_statement	97
Syntax 7.53—slew_statement	97

Syntax 7.54—check_statement	98
Syntax 7.55—check_statement	98
Syntax 7.56—ABS	106
Syntax 7.57—IMAG_PART	106
Syntax 7.58—REAL_PART	106
Syntax 7.59—EXPAND	106
Syntax 7.60—IS_EMPTY	107
Syntax 7.61—NUM_DIMENSIONS	107
Syntax 7.62—NUM_ELEMENTS	107
Syntax 7.63—ISSUE_MESSAGE	107
Syntax 7.64—PRINT_VALUE	108
Syntax 7.65—SOURCE_STRANDS_MSB	108
Syntax 7.66—SOURCE_STRANDS_LSB	109
Syntax 7.67—SINK_STRANDS_MSB	109
Syntax 7.68—SINK_STRANDS_LSB	109
Syntax 7.69—tabledef_statement	109
Syntax 7.70—table_statement	112
Syntax 7.71—load_table_statement	115
Syntax 7.72—unload_table_statement	117
Syntax 7.73—unload_table_statement	118
Syntax 7.74—add_row_statement	118
Syntax 7.75—delete_row_statement	119
Syntax 7.76—subrule_statement	130
Syntax 7.77—subrules_statement	131
Syntax 7.78—tech_family_statement	133
Syntax 7.79—tech_family_statement	136
Syntax 7.80—model_procedure	137
Syntax 7.81—submodel_procedure	138
Syntax 7.82—path_separator_stmt	139
Syntax 7.83—path_statement	140
Syntax 7.84—path_list	141
Syntax 7.85—clkflg_clause	142
Syntax 7.86—ckdtype_clause	142
Syntax 7.87—object_type_clause	143
Syntax 7.88—data_type_sequence	143
Syntax 7.89—bus_statement	145
Syntax 7.90—test_statement	145
Syntax 7.91—compare_list	146
Syntax 7.92—compare_clause	146
Syntax 7.93—edges_clause	147
Syntax 7.94—test_type_clause	147
Syntax 7.95—cycleadj_clause	147
Syntax 7.96—checks_clause	148
Syntax 7.97—methods_list	148
Syntax 7.98—store_clause	148
Syntax 7.99—test_bus_statement	149
Syntax 7.100—input_statement	149
Syntax 7.101—methods_clause	150
Syntax 7.102—store_clause	150
Syntax 7.103—output_statement	152
Syntax 7.104—do_statement – BREAK and CONTINUE	154
Syntax 7.105—do_statement	155
Syntax 7.106—statement_reference	155
Syntax 7.107—statement_reference	155
Syntax 7.108—node_sequence	156
Syntax 7.109—function_assignment_expression	172

Syntax 7.110—vector_sequence.....	173
Syntax 7.111—import_export_sequence.....	174
Syntax 7.112—properties_statement.....	175
Syntax 7.113—setvar_statement.....	175
Syntax 7.114—embedded_C_code.....	176
Syntax 7.115—subrule.....	176
Syntax 7.116—pragma_declare.....	177
Syntax 11.1—Alphanumeric characters.....	597
Syntax 11.2—SPEF names.....	597
Syntax 11.3—SPEF_file.....	598
Syntax 11.4—header_def.....	599
Syntax 11.5—unit_def.....	599
Syntax 11.6—name_map.....	599
Syntax 11.7—power_def.....	599
Syntax 11.8—external_def.....	600
Syntax 11.9—conn_attr.....	600
Syntax 11.10—define_def.....	600
Syntax 11.11—variation_def.....	601
Syntax 11.12—internal_def.....	601
Syntax 11.13—d_net.....	601
Syntax 11.14—conn_sec.....	601
Syntax 11.15—cap_sec.....	602
Syntax 11.16—res_sec.....	602
Syntax 11.17—induc_sec.....	602
Syntax 11.18—r_net.....	602
Syntax 11.19—load_desc.....	603
Syntax 11.20—d_pnet.....	603
Syntax 11.21—pconn_sec.....	603
Syntax 11.22—pcap_sec.....	603
Syntax 11.23—pres_sec.....	603
Syntax 11.24—pinduc_sec.....	604
Syntax 11.25—r_pnet.....	604

DELAY AND POWER CALCULATION STANDARDS –

Part 1: Integrated Circuit (IC) Open Library Architecture (OLA)

FOREWORD

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

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

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

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

IEC 61523-1/IEEE Std 1481 was processed through IEC technical committee 91: Electronics assembly technology, under the IEC/IEEE Dual Logo Agreement. It is an International Standard.

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

IEEE Std	FDIS	Report on voting
1481 (2019)	91/1868/FDIS	91/1883/RVD

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

The language used for the development of this International Standard is English.

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

- reconfirmed,
- withdrawn, or
- revised.

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

IEEE Standard for Integrated Circuit (IC) Open Library Architecture (OLA)

Developed by the

Design Automation Standards Committee
of the
IEEE Computer Society

Approved 7 November 2019

IEEE SA Standards Board

IECNORM.COM : Click to view the full PDF of IEC 61523-1:2023

Abstract: Ways for integrated circuit designers to analyze chip timing and power consistently across a broad set of electric design automation (EDA) applications are covered in this standard. Methods by which integrated circuit vendors can express timing and power information once per given technology are also covered. In addition, the means by which EDA vendors can meet their application performance and capacity needs are discussed.

Keywords: chip delay, electronic design automation (EDA), IEEE 1481™, integrated circuit (IC) design, power calculation

IECNORM.COM : Click to view the full PDF of IEC 61523-1:2023

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

IEEE Standards documents (standards, recommended practices, and guides), both full-use and trial-use, are developed within IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (“IEEE SA”) Standards Board. IEEE (“the Institute”) develops its standards through a consensus development process, approved by the American National Standards Institute (“ANSI”), which brings together volunteers representing varied viewpoints and interests to achieve the final product. IEEE Standards are documents developed through scientific, academic, and industry-based technical working groups. Volunteers in IEEE working groups are not necessarily members of the Institute and participate without compensation from IEEE. While IEEE administers the process and establishes rules to promote fairness in the consensus development process, IEEE does not independently evaluate, test, or verify the accuracy of any of the information or the soundness of any judgments contained in its standards.

IEEE Standards do not guarantee or ensure safety, security, health, or environmental protection, or ensure against interference with or from other devices or networks. Implementers and users of IEEE Standards documents are responsible for determining and complying with all appropriate safety, security, environmental, health, and interference protection practices and all applicable laws and regulations.

IEEE does not warrant or represent the accuracy or content of the material contained in its standards, and expressly disclaims all warranties (express, implied and statutory) not included in this or any other document relating to the standard, including, but not limited to, the warranties of: merchantability; fitness for a particular purpose; non-infringement; and quality, accuracy, effectiveness, currency, or completeness of material. In addition, IEEE disclaims any and all conditions relating to: results; and workmanlike effort. IEEE standards documents are supplied “AS IS” and “WITH ALL FAULTS.”

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

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

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

Translations

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

Official statements

A statement, written or oral, that is not processed in accordance with the IEEE SA Standards Board Operations Manual shall not be considered or inferred to be the official position of IEEE or any of its committees and shall not be considered to be, or be relied upon as, a formal position of IEEE. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that his or her views should be considered the personal views of that individual rather than the formal position of IEEE.

Comments on standards

Comments for revision of IEEE Standards documents are welcome from any interested party, regardless of membership affiliation with IEEE. However, IEEE does not provide consulting information or advice pertaining to IEEE Standards documents. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Since IEEE standards represent a consensus of concerned interests, it is important that any responses to comments and questions also receive the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to comments or questions except in those cases where the matter has previously been addressed. For the same reason, IEEE does not respond to interpretation requests. Any person who would like to participate in revisions to an IEEE standard is welcome to join the relevant IEEE working group.

Comments on standards should be submitted to the following address:

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

Laws and regulations

Users of IEEE Standards documents should consult all applicable laws and regulations. Compliance with the provisions of any IEEE Standards document does not 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

IEEE draft and approved standards are copyrighted by IEEE under US and international copyright laws. They are made available by IEEE and are adopted for a wide variety of both public and private uses. These include both use, by reference, in laws and regulations, and use in private self-regulation, standardization, and the promotion of engineering practices and methods. By making these documents available for use and adoption by public authorities and private users, IEEE does not waive any rights in copyright to the documents.

Photocopies

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

Updating of IEEE Standards documents

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

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

In order to determine whether a given document is the current edition and whether it has been amended through the issuance of amendments, corrigenda, or errata, visit IEEE Xplore at <https://ieeexplore.ieee.org> or contact IEEE at the address listed previously. For more information about the IEEE SA or IEEE's standards development process, visit the IEEE SA Website at <http://standards.ieee.org>.

Errata

Errata, if any, for IEEE standards can be accessed via <https://standards.ieee.org/standard/index.html>. Search for standard number and year of approval to access the web page of the published standard. Errata links are located under the Additional Resources Details section. Errata are also available in IEEE Xplore: <https://ieeexplore.ieee.org/browse/standards/collection/ieee/>. Users are encouraged to periodically check for errata.

Patents

Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken by the IEEE with respect to the existence or validity of any patent rights in connection therewith. If a patent holder or patent applicant has filed a statement of assurance via an Accepted Letter of Assurance, then the statement is listed on the IEEE SA Website at <https://standards.ieee.org/about/sasb/patcom/patents.html>. Letters of Assurance may indicate whether the Submitter is willing or unwilling to grant licenses under patent rights without compensation or under reasonable rates, with reasonable terms and conditions that are demonstrably free of any unfair discrimination to applicants desiring to obtain such licenses.

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

IEEE Introduction

This introduction is not part of IEEE Std 1481-2019, IEEE Standard for Integrated Circuit (IC) Open Library Architecture (OLA).

The objective of the delay and power calculation system (DPCS) is to make it possible for integrated circuit designers to consistently calculate chip delay and power across electronic design automation (EDA) applications and for integrated circuit vendors to express delay and power information only once per technology while enabling sufficient EDA application accuracy.

This is accomplished by a coordinated set of standards that support a standard method to describe timing and power characteristics of integrated circuit design units (cells and higher level design elements); a standard method for EDA applications to calculate chip design instance specific delay, slew, and power for logic and interconnects; and standard file formats to exchange chip parasitic and cluster information.

Acknowledgments

Royalty-free nonexclusive permission has been granted by International Business Machines (IBM) Corporation for all written contributions made by IBM under the direction of Harry J. Beatty III.

Royalty-free permission has been granted by Silicon Integration Initiative, Inc. (Si2) to reprint material from Specification for the Open Library Architecture (OLA), Version 2.0-00, March 1, 2003.

IEEE Standard for Integrated Circuit (IC) Open Library Architecture (OLA)

1. Overview

The delay and power calculation system (DPCS) is a coordinated set of standards that support a standard method to describe timing and power characteristics of integrated circuit (IC) design units (cells and higher level design elements); a standard method for electronic design automation (EDA) applications to calculate chip design instance specific delay, slew, and power for logic and interconnects; and standard file formats to exchange chip parasitic and cluster information. The standard specifications covered in this document are as follows:

- Description language for timing and power modeling, called the “delay calculation language” (DCL)
- Software procedural interface (PI) for communications between EDA applications and compiled libraries of DCL descriptions
- Standard file exchange format for parasitic information about the chip design: Standard Parasitic Exchange Format (SPEF)
- Informative usage examples
- Informative notes

Notes and examples are informative. All other components of this specification are considered normative unless otherwise directed.

1.1 Scope

The scope of this standard focuses on delay and power calculation for integrated circuit design with support for modeling logical behavior and signal integrity.

1.2 Purpose

To improve the IEEE Std 1481™-1999 system for integrated circuit designers to more accurately and more completely analyze semiconductor designs across EDA applications and for integrated circuit vendors to express logical behavior, signal integrity, delay, and power information only once per technology while enabling sufficient EDA application accuracy.

1.3 Introduction

The DPCS standard covers delay and power calculation for integrated circuit design with support for modeling logical behavior and signal integrity, which makes it possible for integrated circuit designers to

analyze chip timing and power consistently across a broad set of EDA applications, for integrated circuit vendors to express timing and power information once (for a given technology), and for EDA vendors to meet their application performance and capacity needs. The intended use for this standard is IC timing and power. This standard may be applied to both unit logic cells supplied by the IC vendor and logical macros defined by the IC designer. Although this standard is written toward the integrated circuit supplier and EDA developer, its application applies equally well to representation of timing and power for designer-defined macros (or hierarchical design elements).

These specifications make it possible to achieve consistent timing and power results, but they do not guarantee it. They provide for a single executable software program that computes delay and power based on IC vendor-supplied algorithms (or designer-supplied algorithms for macros) but does not guarantee EDA applications can correctly communicate the design-specific information required for these algorithms. By specifying standard exchange formats for parasitic data and floorplanning information, this standard provides a marked improvement over design environments with no such standards. However, it is the responsibility of the EDA application to correctly correlate the information between these standard exchange files and the actual design. This standard also does not detail how the information contained within the standard exchange files shall be obtained.

As feature sizes for chips have shrink below 0.25 μm , interconnect delay effects have begun to outweigh those of the logic cells. This means placement of cells and wire routing of the interconnects become as important a factor as the type of cell drivers and receivers on the interconnect. As a result, EDA logic design applications (such as synthesis) have begun to interact closely with physical design applications (such as floorplanning and layout). Applications that before could consider only simple delay and power models now need to deal with complex, interrelated delay and power algorithms. Plus, due to the complexities of the delay and power algorithms, the integrated circuit vendor needs to have control of application calculations and not be restricted by the limitations of a broad set of applications demanded by the customers (the designers).

Over the past few years, it has become increasingly apparent that modern very large-scale integration (VLSI) design is no longer bounded only by timing and area constraints. Power has become significantly more important. In an era of hand-held devices ranging from mobile computing to wireless communication systems, managing and controlling power takes on an important role. Several benefits can be attained from low-power designs in addition to extended battery life. Low-power devices often run at a lower junction temperature, which leads to higher reliability and lower cost cooling systems. There are also several challenges for calculation and modeling of power (and delay) in deep submicron (less than 0.25 μm) designs. EDA tools can now accurately calculate and model power by using this DPCS standard.

1.4 Word usage

The word *shall* indicates mandatory requirements strictly to be followed in order to conform to the standard and from which no deviation is permitted (shall equals is required to).^{1,2}

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

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

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

¹ The use of the word *must* is deprecated and cannot be used when stating mandatory requirements, *must* is used only to describe unavoidable situations.

² The use of *will* is deprecated and cannot be used when stating mandatory requirements, *will* is only used in statements of fact.

2. Normative references

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

ISO/IEC 9899:1990, Programming Languages—C.³

ISO/IEC 14882:2003, Programming Languages—C++.

3. Definitions

For the purposes of this document, the following terms and definitions apply. The *IEEE Standards Dictionary Online* should be consulted for terms not defined in this clause.⁴

application, electronic design automation (EDA) application: Any software program that interacts with the delay and power calculation module (DPCM) through the procedural interface (PI) to compute instance-specific timing values. Examples include batch delay calculators, synthesis tools, floor-planners, static timing analyzers, and so on. *See also:* **delay and power calculation module; procedural interface.**

arc: *See:* **timing arc.**

argument: The value or the address of a data item passed to a function or procedure by the caller.

back-annotation: The annotation of information from further downstream steps (toward fabrication) in the design process. *See also:* **back-annotation file.**

back-annotation file: A file containing information to be read by a tool for the purpose of back-annotation, for example, Physical Design Exchange Format (PDEF) and Standard Parasitic Exchange Format (SPEF) files. *See also:* **back-annotation; timing annotation.**

bidirectional: A pin or port that can place logic signals onto an interconnect and receive logic signals from it (i.e., act both as a driver and as a receiver).

bias: The time difference between the data arrival time and a specified signal edge (e.g., of a clock). Also, the *BIAS* clause used in a *CHECK* statement.

C-effective: A capacitance value, often computed as an approximation to a resistor/inductor/capacitor (RLC) network or a model, that characterizes the admittance of an interconnect structure at a particular driver. The reduction of real parasitics and pin capacitances to a C-effective allows the calculation of delay and slew values from cell characterization data that assumes a pure capacitive output load.

bus: In Physical Design Exchange Format (PDEF), a physical collection of nets and/or pnets or of pins and/or nodes. If the items collected in the PDEF bus are logical, the PDEF bus may or may not correspond to a logical bus described in the netlist.

cell: A primitive in an integrated circuit library. For the purposes of this specification, “primitive” means the timing properties of the cell are directly described in the delay and power calculation module (DPCM) without reference back to the application for the internal structure of the cell. This primitiveness typically is

³ISO publications are available from the International Organization for Standardization (<http://www.iso.org/>) and the American National Standards Institute (<http://www.ansi.org/>). IEC publications are available from the International Electrotechnical Commission (<http://www.iec.ch>) and the American National Standards Institute (<http://www.ansi.org/>).

⁴*IEEE Standards Dictionary Online* is available at: <http://dictionary.ieee.org>.

a result of the characterization of that cell by the semiconductor vendor, but it may instead be a result of the construction of a timing model for a sub-circuit by the application and its loading into the DPCM at run time. The term “cell” can arise in the context of the abstraction of a type of cell available in the library or in the concrete selection and placement of a cell in the final design. If the context is not clear, the terms “cell type” and “cell instance” (or just “instance”) shall be used. *See also:* **cell type; instance.**

cell type: Name used to identify a particular cell in the library.

cluster: A grouping of cell instances and/or clusters that are constrained to each other due to physical location or some other shared characteristic(s). It is not valid to have a cell instance explicitly made a member of more than one cluster. *See also:* **region.**

column: In a Physical Design Exchange Format (PDEF) datapath cluster, a cluster of cell, spare_cell, and/or cluster instances placed or constrained to be placed in the vertical (y-axis) direction. *See also:* **row; datapath.**

constraint: A timing property of a design that is supplied as a goal or objective to an electronic design automation (EDA) tool, such as logic synthesis, floorplanning, or layout. The tool shall not start out with a fixed design implementation; it shall build or modify the design to meet the constraint. *See also:* **timing check.**

datapath: A type of electronic design automation cluster that contains rows and/or columns of cluster, cell, skip, and/or spare_cell instances. A PDEF datapath typically corresponds to structured logic. *See also:* **row; column.**

delay: The time taken for a digital signal to propagate between two points.

delay and power calculation module (DPCM): A delay and power calculation system (DPCS)-compliant software component supplied by a semiconductor vendor that is responsible for computing instance-specific timing data under control of an electronic design automation (EDA) application. The DPCM is loaded into memory at run time and linked to the application via the procedural interface (PI). A DPCM typically is created from delay calculation language (DCL) subrules compiled by the DCL compiler and linked together with run-time support modules.

delay and power calculation system (DPCS): The complete system detailed in this specification: the delay calculation language (DCL)-language, the procedural interface (PI) for delay and power calculations, and text formats for physical design and parasitic information.

delay arc: *See:* **timing arc.**

delay calculation language (DCL): The programming language used to calculate instance-specific timing data. DCL contains high-level constructs that can refer to the aspects of the design topology that influence timing and also express the sequence of calculations necessary to compute the desired delay and timing check limit values.

delay calculation language (DCL) compiler: A software program, used in conjunction with a C compiler, that reduces DCL from ASCII text to computer executable format. *See also:* **delay and power calculation module.**

delay equation: Any mathematical expression describing cell delay or interconnect delay.

driver: A pin of a cell instance that, in the current context, is placing or can place a signal onto an interconnect structure.

early mode: The first edge that propagates through a given cone of logic.

fanout: The pin count of a net (the number of pins connected to the net), minus one. This definition includes all input, output, and bidirectional pins on the net with the sole exception of one pin (assumed to be related to the particular timing arc currently of interest). Although less fundamental than pin count, fanout is frequently used in the definition of wire load models.

forward annotation: The annotation of information from further upstream (earlier in the design flow) in the design process. *See also:* **forward annotation file**.

forward annotation file: A file containing information to be read by a tool for the purpose of forward annotation, for example, an Standard Delay Format (SDF) file containing PATHCONSTRAINTS. *See also:* **forward annotation**.

function, procedural interface (PI) function: One of the C functions that comprise the **delay and power calculation system (DPCS)** procedural interface.

gap: In Physical Design Exchange Format (PDEF), spacing between rows and/or columns in a datapath.

gate: In Physical Design Exchange Format (PDEF), the physical abstraction of a library primitive.

hard macro: A cluster whose cell placements relative to each other are fixed. Often the interconnect routing between the cells is also fixed and a parasitics file describing the interconnect is available for the hard macro. The location of the hard macro in the floorplan may or may not be fixed.

hard region: A cluster that has defined physical boundaries in a floorplan. All cells contained in the cluster shall be placed within the boundaries of the cluster.

hierarchical instance: The concrete appearance of a design unit at some hierarchical level. Because higher level design units may be instantiated multiple times, a single such appearance may give rise to multiple instances of the lower level design units within it. Where instances are referred to as “occurrences,” hierarchical instances are referred to simply as instances.

hold timing check: A timing check that establishes only the end of the stable interval for a setup/hold timing check. If no setup timing check is provided for the same arc, transitions, and state, the stable interval is assumed to begin at the reference signal transition and a negative value for the hold time is not meaningful. *See also:* **setup/hold timing check**.

hold time: *See:* **setup/hold timing check; nochange timing check**.

implementation-defined behavior: Behavior, for a correct program construct and correct data, that depends on the software implementation and that each implementation shall document. The range of possible behaviors is delineated by the standard.

implementation limits: Restrictions imposed by an implementation.

input: A pin or port that shall only receive logic signals from a connected net or interconnect structure.

instance, cell instance: A particular, concrete appearance of a cell in the fully expanded (flattened, unfolded, elaborated) design description of an integrated circuit, also referred to elsewhere as an “occurrence.” An instance is a “leaf” of the unfolded design hierarchy. In Physical Design Exchange Format (PDEF), this is a physical cluster or a logical cell. *See also:* **cell; cell type; cluster; hierarchical instance**.

interconnect: A collective term for structures (in an integrated circuit) that propagate a signal between the pins of cell instances with as little change as possible. These structures include metal and polysilicon

segments, vias, fuses, anti-fuses, and so on. But interconnect shall not include such structures if they occur as part of the fixed layout of a cell.

late mode: The very last edge that propagates through a given cone of logic.

layer: In Physical Design Exchange Format (PDEF), a particular level of interconnect on which a logical or physical pin is located.

library (integrated circuit): A collection of circuit functions, implemented in a particular integrated circuit technology, which an integrated circuit designer or electronic design automation (EDA) synthesis application can select in order to implement a design. *See also:* **cell**.

library (software): A collection of object code units that may be linked, either statically or at run time, with other libraries and/or object code to produce a software program.

library control statements: These statements control the logical organization and loading of subrules in a technology library. *See also:* **technology library**.

load-dependent delay: That part of a delay through a cell instance attributed to the admittance (load) presented to the arc sink pin and the internal impedance of the output.

mesh table: A multidimension table that defines every type of delay model in terms of discrete points. Each point represents a delay value in terms of several cell parameters or interconnect parameters. The delay calculation module is expected to interpolate between these points based on a mathematical expression defined by the technology file.

(to) model a cell: The creation of a specific elaboration of a model using *modelSearch*.

modeling procedures: Describe the action of a circuit with respect to timing and power. These actions include creating segments and nodes, determining the propagation properties, and setting the delay and slew equations to use.

modeling statements: Delay calculation language (DCL) statements that map cell configurations to modeling procedures.

net, net instance: An abstraction expressing the idea of an electrical connection between various points in a design. In a hierarchical representation of the design, nets can occur at all levels and may connect to pins of lower hierarchical levels (including cell instances), ports of the current hierarchical level, and each other. In a flattened (unfolded and elaborated) design, electrically connected nets are collapsed and each net instance corresponds to a unique interconnect structure in the implementation.

nochange timing check: A timing check similar to a setup/hold timing check except the setup and hold times are referred to opposite transitions of the reference signal. The stable interval is extended to include the period between these transitions, that is, the time for which the reference signal stays in a specified state. This timing check is frequently applied to memory and latch-banks to establish the stability of the address or select inputs before, during, and after the write pulse.

node: A conceptual point (through which logic signals pass) that has been identified as an aid to modeling the timing properties of a cell but may not correspond to any physical structure. In Physical Design Exchange Format (PDEF), this is a physical pin that does not correspond to a logical structure.

nugget: A data structure used in the procedural interface (PI) for rapid switching between technologies.

output: A pin or port shall only place logic signals onto a connected net or interconnect structure.

parameter: A data item required for the calculation of some result.

parasitics: Electrical properties of a design (resistance, capacitance and impedance) that arise due to the nature of the materials used to implement the design.

period timing check: A timing check that specifies the allowable time between successive periods of a signal.

periphery: The outer part of an integrated circuit where instances of cell types designed specifically to interface the internal circuitry to the “outside world” are placed. This part includes “pad” cells (which are input and output buffers) and power and ground pads; it may also include test circuitry, such as boundary scan cells.

pi-model: A simplification of a general resistor/inductor/capacitor (RLC) network that represents the driving-point admittance for an interconnect.

pin: A terminal point where an interconnect structure makes electrical contact with the fixed structures of a cell instance or the conceptual point where a net connects to a lower level in the design hierarchy.

pin count: The number of cell instance pins that an interconnect structure visits, including all input, output, and bidirectional pins. Pin count is the number of “places” the interconnect goes to on the chip.

pnet: A physical net that has no correspondence to the logical function of the design, such as a route segment that is reserved for future routes across a hard macro, or a power net not described in the design netlist.

pole: The complex frequency where a Laplace Transform is infinite. Combined with residues, this is a convenient mathematical notation for the impedance or transfer function of a passive circuit, such as an resistor/inductor/capacitor (RLC) circuit because poles above this frequency can be ignored in calculations without significant loss of accuracy.

port: A conceptual point at that a cell or a hierarchical design unit makes its interface available to higher levels in the design hierarchy.

primary input: The point where a logic signal arrives at the boundary of the design as currently known to an electronic design automation (EDA) application. For a complete integrated circuit design, for example, this point is the metal pad of an input or bidirectional pad cell.

primary output: The point where a logic signal leaves the design as currently known to an electronic design automation (EDA) application. For a complete integrated circuit design, for example, this point is the metal pad of an output or bidirectional pad cell.

procedural interface (PI): The set of C functions used by an application and a delay and power calculation module (DPCM) to exchange information and determine the timing calculation for a design.

pulse width timing check: A timing check that specifies the minimum time a signal shall remain in a specified state once it has transitioned to that state.

receiver: A pin of a cell instance that is receiving or can receive a signal from an interconnect structure.

recovery/removal timing check: A timing check that establishes an interval with respect to a reference signal transition during which an asynchronous control signal may not change from the active to inactive state. This timing check is frequently applied to flip flops and latches to establish a stable interval for the set and reset inputs with respect to the active edge of the clock or the active-to-inactive transition of the gate.

Two limit values are necessary to define the stable interval. The recovery time is the time before the reference signal transition when the stable interval begins. The removal time is the time after the reference signal transition when the stable interval ends.

If the asynchronous control signal goes inactive during the stable interval, it is unknown whether the flip flop or latch takes on the state of the data input, remains set, or is reset.

recovery time: *See: recovery/removal timing check.*

recovery timing check: A timing check that establishes only the beginning of the stable interval for a recovery/removal timing check. If no removal timing check is provided for the same arc, transitions, and state, the stable interval is assumed to end at the reference signal transition and a negative value for the recovery time is not meaningful. *See also: recovery/removal timing check.*

region: A particular physical section or block of a floor plan. *See also: cluster.*

removal time: *See: recovery/removal timing check.*

removal timing check: A timing check that establishes only the end of the stable interval for a recovery/removal timing check. If no recovery timing check is provided for the same arc, transitions, and state, the stable interval is assumed to begin at the reference signal transition and a negative value for the removal time is not meaningful. *See also: recovery/removal timing check.*

residue: The value that has the complex pole. *See also: pole.*

resistance times capacitance (RC) time constant: The product of some resistance and some capacitance (having the dimensions of time) or a time constant computed in some other way.

return code: A value returned by a function indicating whether the function completed successfully. If the function did not complete successfully, it may return a nonzero return code; the exact value may indicate one of several possible severity conditions: informational, warning, error, severe, terminal error, etc.

route, global route: In Physical Design Exchange Format (PDEF), the physical description of interconnect routing between logical and physical pins of cell, spare_cell, and/or cluster instances.

row: In a Physical Design Exchange Format (PDEF) datapath cluster, a cluster of cell, spare_cell and/or cluster instances placed or constrained to be placed in the vertical (Y-axis) direction. *See also: column; datapath.*

scalar: An integer constant.

sequence point: A certain point in the execution sequence of a program where all side effects of previous evaluations are complete and no side effects of subsequent evaluations have occurred. (Refer to ISO/IEC 9899:1990, Section 5.1.2.3, page 7.⁵)

segment: A portion of an interconnect structure treated as a unit for the purposes of extracting or estimating its electrical properties. *See also: parasitics.*

setup/hold timing check: A timing check that establishes an interval with respect to a reference signal transition during which some other signal may not change value. This timing check is frequently applied to flip flops and latches to establish a stable interval for the data input with respect to the active edge of the clock or the active-to-inactive transition of the gate.

⁵Information on references can be found in Clause 2. .

Two limit values are necessary to define the stable interval. The setup time is the time before the reference signal transition when the stable interval begins and shall be negative if the stable interval begins after the reference signal transition. The hold time is the time after the reference signal transition when the stable interval ends and shall be negative if the stable interval ends before the reference signal transition. If the data signal changes during the stable interval, then the reliability of the resulting state of the flip flop or latch is unknown.

setup time: *See: setup/hold timing check; nochange timing check.*

setup timing check: A timing check that establishes only the beginning of the stable interval for a setup/hold timing check. If no hold timing check is provided for the same arc, transitions, and state, the stable interval is assumed to end at the reference signal transition and a negative value for the setup time is not meaningful. *See also: setup/hold timing check.*

shared port: An output or bidirectional port where some other output port of the cell derives its logic function. The output load at a shared port affects not only the delay to that port itself but also the delay to any ports sharing it.

sink, sink pin: The sink pin is the end of a delay arc (i.e., the destination of the logic signal). For arcs across cell instances, the sink is the driver pin. For arcs across an interconnect, the sink is the receiver pin.

size metric: A value used to estimate properties of interconnect wholly contained in a region. The metric may be freely chosen (for example, square microns or gate sites), but it needs to be consistent between the cells and the wire load models. *See also: wire load model.*

skew timing check: A timing check that specifies the maximum time between two signal transitions. This timing check is frequently applied to dual-clock flip flops to specify the maximum separation of the active edges of the two phases of the clock.

skip: In Physical Design Exchange Format (PDEF), spacing between the ordered cell, spare_cell, and/or cluster instances in rows and/or columns of a datapath.

slew: A measure of the shape of the waveform constituting a logic state transition. A slew value can have the dimensions of time (in which case, it is a *slew time*) or the dimensions of voltage per time (in which case, it is a *slew rate*). The delay and power calculation system (DPCS) allows either interpretation if used consistently.

slew-dependent delay: That part of an input-to-output delay that can be attributed to the signal at the input of the arc taking longer to make a transition than is considered ideal.

slew rate: A measure of how quickly a signal takes to make a transition (i.e., a voltage-per-unit time). Slew rate is inversely related to slew time and is sometimes used incorrectly where slew time is intended.

slew time: A measure of how long a signal takes to make a transition (i.e., the rise time or fall time). Slew time is inversely related to slew rate. The way a slew time value is abstracted from the continuous waveform at a cell pin varies with different cell characterization methods.

soft region: A cluster that does not have a specified physical location in a floorplan. It may have constraints on how closely the cells within the cluster are placed relative to each other. A soft region may be located within a hard region.

source, source pin: The source pin is the start of a delay arc (i.e., the origin of the logic signal). For arcs across cell instances, the source is the receiver pin. For arcs across interconnect, the source is the driver pin.

spare_cell: A cell instance that is presently not part of the logical function of a design and therefore is not included in the design's logical netlist. A spare_cell is typically reserved for future logic modifications to be implemented through changes in the interconnect layers of the chip.

Standard Structure: A particular C structure, defined in `std_stru.h`, which contains fields used to pass data over the procedural interface (PI) (thus avoiding large numbers of arguments). Most functions of the PI have a pointer to a Standard Structure as their first argument.

technology data: Data used to calculate the timing properties of a cell instance based on its context in the design. This term includes information that is not cell type specific and data specific for each cell type in the library. The kind of data used varies with the timing calculation methodology. General data and cell data may be contained in the same file or in separate files. Cell data also may be merged with the timing models of each cell, for example, when a tool performs its own timing calculation.

technology library: A technology library is a program written in delay calculation language (DCL) consisting of one or more subrules, each of which may contain references to other subrules (yet to be loaded). There is no hierarchical limit to the nesting of subrules within the scope of a technology library. Subrules can also be segmented into technology families, which alters the way they are made available to the application.

time-of-flight: The time delay between a signal leaving a driving pin or primary input port and reaching a receiving pin or primary output. Time-of-flight is generally dominated by the time taken to charge the distributed capacitance of the interconnect and the capacitance of the driven pins through the distributed impedance of the interconnect. The internal impedance of the driving port affects the load-dependent delay but not (directly) the time-of-flight.

timing annotation (file): The annotation of a design in one tool with timing data computed by another tool. If timing calculation is performed as an offline process (separately from the application using the timing data), the process of reading the timing data into the tool is known as timing annotation. A timing annotation file stores the data written by the timing calculator and is later read by an application.

timing arc: A pair of ports, pins, or nodes possess some timing relationship such as the propagation delay of a signal from one to the other or a timing check between them. Delay arcs may be between two distinct ports or nodes of a cell or over the interconnect from driver pins to receiver pins.

timing calculation, delay calculation: The process of calculating values for the delays and timing checks associated with the physical primitives (cells) of an integrated circuit design, or part of an integrated circuit design, and their interconnections.

timing check: A timing property of a circuit (frequently a cell) that describes a relationship in time between two input signal events. This relationship needs to be satisfied for the circuit to function correctly.

timing model: A timing model represents the timing behavior of a cell for applications such as simulation and timing analysis. For black-box timing behavior, it represents the definition of pin-to-pin delays between any pair of pins as well as internal nodes. In addition, for sequential cells it provides the definition of timing checks and constraints on any pair of pins and/or internal nodes.

transition: The change of a logic signal from one state to another (as in "... a transition at the input shall cause ...") or the pair of logic states between which a transition may occur (as in "... the delay for a low-to-high transition ...").

unloaded delay: The conceptual delay value for a delay arc of a cell when the output pin is unloaded (unconnected) and the signal at the input pin conforms to some ideal waveform.

undefined behavior: Behavior for which the standard imposes no requirements (e.g., use of an erroneous program construct). Permissible undefined behavior may occur in the following range:

- a) Ignoring a situation completely with unpredictable results
- b) Behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message)
- c) Terminating a translation or execution (with the issuance of a diagnostic message)

NOTE—Many erroneous program constructs do not engender undefined behavior; they are required to be diagnosed.⁶

unspecified behavior: Behavior (for a correct program construct and correct data) that depends on the implementation. The implementation is not required to document which behavior occurs. Usually the range of possible behaviors is delineated by the standard.

via: In Physical Design Exchange Format (PDEF), a physical connection between two different levels of interconnect or between a level of interconnect and a physical or logical pin.

wire load model: A statistical model for the estimation of interconnect properties as a function of the geometric measures available before the completion of layout and routing. Typical model properties include fanout, capacitance, length, and resistance. *See also: size metric.*

4. Acronyms and abbreviations

This clause lists the acronyms and abbreviations used in this standard, as follows:

API	application programming interface
ASIC	application specific integrated circuit
AWE	asymptotic waveform evaluation
BNF	Backus-Naur form
CGHT	clock gating hold test
CGST	clock gating setup test
CST	clock separation test
DCL	delay calculation language
DHT	data hold test
DPCM	delay and power calculation module
DPCS	delay and power calculation system
DPW	data pulse width
DST	data setup test
EDA	electronic design automation
EDIF	Electronic Design Interchange Format
HDL	hardware description language

⁶Notes in text, tables, and figures of a standard are given for information only and do not contain requirements needed to implement this standard.

IC	integrated circuit
PDEF	Physical Design Exchange Format
PI	procedural interface
PVT	process/voltage/temperature
RLC	resistor/inductor/capacitor
RC	resistance times capacitance
SDF	Standard Delay Format
SPEF	Standard Parasitic Exchange Format
SPICE	Simulation Program with Integrated Circuit Emphasis
VHDL	VHSIC Hardware Description Language
VHSIC	very-high-speed integrated circuit
VLSI	very-large-scale integration

5. Typographical conventions

This clause describes the typographical conventions used within this specification.

5.1 Syntactic elements

- Italicized, lowercase words, some containing embedded underscores, are used to denote syntactic categories (terminals and nonterminals), for example, *rule_name*
- Boldface words are used to denote reserved keywords, operators, and punctuation marks as a required part of the syntax.

MODEL_SLEW, *D_NET or:

- The **::=** operator separates the two parts of a Backus-Naur form (BNF) syntax definition. The syntax category appears to the left of this operator, and the syntax description appears to the right of the operator. For example, the following step shows the four options for a *suffix_bus_delim*.
- A vertical bar separates alternative items (use one only) unless it appears in boldface; in which case, it stands for itself. For example,

suffix_bus_delim ::=] | } |) | >

- Square brackets enclose optional items unless it appears in boldface; in which case, it stands for itself. For example,

r_net ::= ***R_NET** *net_ref* *total_cap* [*routing_conf*] {*driver_reduc*} ***END**

indicates *routing_conf* is an optional syntax item for *r_net*, whereas

suffix_bus_delim ::=] | } |) | >

indicates the closing square bracket is part of the *suffix_bus_delim* character set.

- f) Braces enclose a repeated item unless it appears in boldface; in which case, it stands for itself. The item may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule. An item that may appear one or more times is listed first, followed by the item enclosed in braces, such as

pos_integer ::= <digit>{<digit>}

- g) Parentheses enclose items within a group (use one only) unless it appears in boldface; in which case, it is a literal token. In the following example:

bus ::= (**BUS** *physical_name* {*attribute*} ({*net_ref*} | {*pin_ref*} {*node_ref*}))

the first set of parentheses are part of the syntax for a *bus*, and the second set groups the items *net_ref* OR the combination of a *pin_ref* and *node_ref*.

- h) Angle brackets enclose items when no spacing is allowed between the items, such as within an *identifier*. In the following example:

identifier ::= <identifier_char>{<identifier_char>}

the actual character(s) of the identifier cannot have any spacing.

- i) A hyphen (-) is used to denote a range. For example:

upper ::= **A – Z**

indicates that *upper* can be an uppercase letter (from A to Z).

5.2 Conventions

Conventions used in the main text are as follows:

- Italicized* font is used when a term is being defined.
- Monospace font is used for examples, file names, and references to constants such as 0, 1, or x values.

6. DPCS flow

The goals of the DPCS architecture are to make it possible in a multivendor, multitool environment for the following to occur:

- Integrated circuit designers to calculate timing and power consistently
- Integrated circuit vendors to express timing and power information once for a given technology while enabling sufficient EDA application accuracy

6.1 Overview

To meet these goals, the DPCS shall have total control of delay and power calculation, support arbitrary expressions for delay and power values, and have very high performance. In addition, the DPCS architecture shall permit integrated circuit vendors to supply data (equations, coefficients, or algorithms) for delay and power calculation to their customers independent of the release of design tools by EDA vendors.

Figure 1 shows a high-level representation of the DPCS architecture and relationships among several key components for calculating delay and power in an integrated circuit design environment.

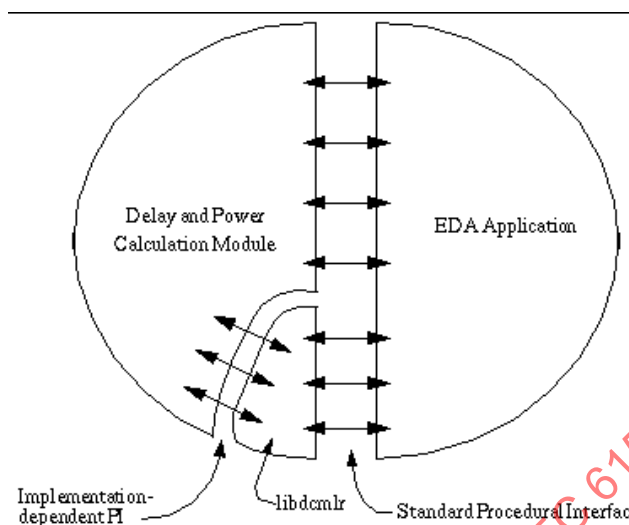


Figure 1—High-level DPCS architecture linkage structure

The DPCM contains the following code (compiled from a DCL source) that enables an application to compute power, delays, and timing constraints (for example, setup and hold tests) efficiently:

- a) The PI is used by both an application and the DPCM to control delay and power calculations.
- b) The Standard Delay Format (SDF) contains constructs for describing computed timing data (for back annotation) and specifying timing constraints (for forward annotation). In integrated circuit design, it is common for a delay calculator application to calculate delays from postlayout data and write out the results in SDF.
- c) The Physical Design Exchange Format (PDEF) contains constructs to describe physical design information. It can be used as a back-annotation medium for passing physical design information from back-end applications (floorplanning and layout) to those on the front end; it can also be used to communicate physical constraints from frontend applications (synthesis, timing analysis, and partitioning) to those on the back end.
- d) SPEF contains constructs to convey information between parasitic extractors and applications that utilize the extracted circuit information.

Figure 1 also illustrates the following key points about applications that require delay and power information:

- Applications shall be written (or modified) to use the DPCS-defined PI for all delay and power calculations.
- The application is fully responsible for mapping between design instances and cell types.
- The application is fully responsible for the state of the design. The DPCM is responsible for calculating delay and power and uses the PI to acquire all required design-specific information, including such data as models of parasitic effects (often contained in SDF, PDEF, or SPEF files).

- The DPCM is composed of one or more subrules. Subrules are compiled C code constructed from an ASCII description following the rules of the DCL. A subrule may implement zero or more timing and power models. The DPCM is dynamically linked with the application during execution.
- Errors in timing and power models may be incrementally corrected by the release of one or more subrules.

6.1.1 Procedural interface

A crucial part of the DPCS standard is the PI between an application and the DPCM. This PI enables the same DPCM to function with multiple applications. Details of the PI are described in Clause 10.

To enable an application to be independent of implementation-specific characteristics of a DPCM, code that implements the functions *dcMRT_InitRuleSystem* and/or *dcMRT_BindRule* shall not be statically linked with the application. Figure 1 illustrates the required linkage structure. The code for the three functions mentioned previously shall reside in a shared object-code library named *libdcMapp+* that is dynamically linked to the application at run time. There may be implementation-dependent procedure calls between *libdcMapp+* and the DPCM, but this architecture isolates the application from such details.

When the EDA application is statically linked, the link editor shall be told which external symbols need be resolved with dynamically loaded code and how to locate that code. The specification of how to locate the code shall allow the application to choose from among multiple implementations of *libdcMapp+*.

6.1.2 Global policies and conventions

The DPCM assumes a single locus of control between it and the application. The DPCM also assumes the application understands any data returned as part of a call as well as any data passed to the DPCM by the application that is passed on to the application as part of a callback. For example, if an application passes a pointer to a port structure as one argument in a call to the DPCM, the DPCM assumes it can pass that same pointer as an argument to a callback function.

In general, the DPCM shall not cache any design-specific data values between calls because the DPCM cannot know when any such data might become invalid. The DPCM can cache within a single application call.

6.2 Flow of control

During execution, the flow of control within the DPCS moves between the application and the DPCM, which subject to the following two constraints:

- There is no “abnormal” flow of control across its PI; that is, all function calls return control through the normal procedure-return mechanism.
- A DPCM shall not terminate the execution of the application to which it is linked.

6.3 DPCM—application relationships

The DPCM has been designed to allow easy integration of dynamically linked executable modules that performs delay and power calculation. The library developer decides how the components of the technology library should be architected and implemented, and organized into physical units called “subrules.” Subrules may be segmented into technology families, which defines the way they are made available to the application (see 10.8). The run-time system assembles these subrules into a unified DPCM. The application then views the DPCM as a single module from which it can request services.

6.3.1 Technology library

A technology library is a program written in DCL consisting of one or more subrules, each of which may contain references to other subrules (yet to be loaded). There is no hierarchical limit to the nesting of subrules within a technology library.

6.3.2 Subrule

A subrule contains a combination of definitions and prototypes used to implement a portion of a technology definition. A subrule is a separate compilation unit and consists of one or more of the following components:

- a) C preprocessor directives
- b) Embedded C code
- c) DCL statements

6.4 Interoperability

Consideration has been given to having multiple implementations to supporting systems for this standard (compiler, run-time linker, and run-time library) and the desire to support at least a minimal level of interoperability among such implementations.

In particular, an application can concurrently access subrules constructed in multiple implementations by doing the following:

- a) Limiting all subrules in a particular technology family to a single implementation
- b) Appending an implementation-specific suffix to the `dcm` prefix for all PI calls whose names begin with `dcm`
- c) Requiring the application to
 - 1) Track which technology families use which implementation
 - 2) Use the appropriately named PI calls
 - 3) Maintain separate implementation-specific Standard Structures

7. Delay calculation language (DCL)

This clause describes the language for delay modeling and calculation used in this standard.

7.1 Character set

The DCL character set shall be the same as that defined in Section 5.2.1 (“Character sets”) of ISO/IEC 9899:1990, excluding trigraph sequences and multibyte characters. The alphabetic escape sequences such as “\n,” representing newline, shall be the same as those defined in Section 5.2.2 (“Character display semantics”) of ISO/IEC 9899:1990.

7.2 Lexical elements

This subclause describes the lexical elements used to define the DCL syntax and semantics.

7.2.1 Whitespace

Whitespace is a contiguous sequence of one or more characters in the set: *space*, *horizontal_tab*, *newline*, *carriage_return*, *vertical_tab*, and *form_feed*.

7.2.2 Comments

A comment shall be either of the following:

- a) A character sequence that starts with */** and ends with the first occurrence of the character sequence **/*. Within a comment, the character sequence */** shall not be recognized as starting a nested comment.
- b) A character sequence that starts with *//* and ends with the first occurrence of either the *newline* or *carriage_return* characters.

7.2.3 Tokens

The syntax for tokens in DCL is given in Syntax 7.1.

```
token ::=
  keyword
  | identifier
  | double_quoted_character_sequence
  | constant
  | string_literal
  | operator
  | punctuation
  | header_name
```

Syntax 7.1—token

Tokens other than *string_literal* and *double_quoted_character_sequence* shall not contain embedded *whitespace*.

7.2.3.1 Keyword

DCL reserves many tokens as *keywords* (see Table 1). These keywords shall be used only in the context as defined by DCL. No keyword shall be used as an identifier (see 7.2.3.2).

Table 1—Keywords

ABS	NODES
ABSTRACT	NUM_DIMENSIONS
ADD_ROW	NUM_ELEMENTS
AGGREGATE	NUMBER
ANYIN	ONE_TO_Z
ANYOUT	OPTIONAL
ARGV	OTHERWISE
ASSIGN	OUTPUT
AUTOLOCK	OUTPUT_PIN_COUNT
BIAS	OUTPUT_PINS
BIT	OVERRIDE
BINARY	PASSED
BLOCK	PATH
BOTH	PATH_DATA
BREAK	PATH_SEPARATOR
BUS	PHASE

BUSY	PIN
BY	PINLIST
CALC	PRIMITIVE
CALC_MODE	PRINT_VALUE
CALC_MODE_SCALAR	PRAGMA
CALL	PROCESS_VARIATION
CELL	PROCESS_VARIATION_SCALAR
CELL_DATA	PROPAGATE
CELL_QUAL	PROPERTIES
CGHT	PROTOTYPE_RECORD
CGPW	PROXY
CGST	PURE
CHAR	QUALIFIERS
CHECK	READ_LOCK
CHECKS	REAL_PART
CKTTYPE	RECOVERY
CLKFLG	REFERENCE
COMMON	REFERENCE_EDGE
COMPARE	REFERENCE_EDGE_SCALAR
COMPILATION_TIME_STAMP	REFERENCE_MODE
COMPLEX	REFERENCE_MODE_SCALAR
COND_NODE	REFERENCE_POINT
CONSISTENT	REFERENCE_POINT_PIN_ASSOCIATION
CONSTANT	REFERENCE_SLEW
CONTINUE	REMOVAL
CONTROL_PARM	REPLACE
CPW	REPEAT
CST	RESULT
CYCLEADJ	RETRY
DATA	RISE
DEFAULT	ROUTE
DEFINES	RULE_PATH
DELAY	SETUP
DELETE_ROW	SETVAR
DESCRIPTOR	SEVERE
DHT	SHARED
DIFFERENTIAL_SKEW	SHORT
DO	SIGNAL
DOUBLE	SIGNAL_EDGE
DPW	SIGNAL_EDGE_SCALAR
DST	SIGNAL_MODE
DYNAMIC	SIGNAL_MODE_SCALAR
EARLY	SIGNAL_POINT
EARLY_MODE	SIGNAL_POINT_PIN_ASSOCIATION
EARLY_MODE_SCALAR	SIGNAL_SLEW
EARLY_SLEW	SIGNED
EDGES	SINK_EDGE
END	SINK_EDGE_SCALAR
ERROR	SINK_MODE
EVAL	SINK_MODE_SCALAR
EXCESS_64	SINK_STRANDS
EXCESS_128	SINK_STRANDS_LSB
EXPAND	SINK_STRANDS_MSB
EXPANDED	SKEW
EXPORT	SLEW
EXPOSE	SOURCE_EDGE
EXTERNAL	SOURCE_EDGE_SCALAR
FALL	SOURCE_MODE
FILE	SOURCE_MODE_SCALAR
FILE_PATH	SOURCE_STRANDS
FILTER	SOURCE_STRANDS_LSB
FLOAT	SOURCE_STRANDS_MSB

FOR	SPACE
FORCE	STATEMENTS
FORWARD	STEP_TABLE_BACKWARDS
FREE_SPACE	STEP_TABLE_CURRENT
FROM	STEP_TABLE_END
FROM_POINT	STEP_TABLE_FORWARDS
FROM_POINT_PIN_ASSOCIATION	STEP_TABLE_START
FUNCTION_HOLD	STEP_TABLE_TO_DEFAULT_RECORD
IMAG_PART	STORE
IMPORT	STRING
IMPORT_EXPORT_TAG	SUBMODEL
IMPURE	SUBRULE
INCONSISTENT	SUBRULES
INFORM	SUPPRESS
INPUT	TABLE
INPUT_PIN_COUNT	TABLEDEF
INT	TABLE_PATH
INTERNAL	TECH_FAMILY
IS_EMPTY	TERM
KEY	TEST
LAST	TEST_TYPE
LATE	TO
LATE_SLEW	TO_POINT
LAUNCHABLE	TO_POINT_PIN_ASSOCIATION
LEADING	TRANSIENT
LOCAL	TRAILING
LOCK	UNCOMPRESSOR
LOCK_ATTRIBUTE_BUSY	UNCOMPRESS_ON_RECALL
LOCK_ATTRIBUTE_LAZY	UNCOMPRESS_ON_STORE
LONG	UNLOAD_TABLE
METHOD	UNSIGNED
METHODS	UNTIL
MODEL	VAR
MODEL_DOMAIN	VECTOR
MODEL_NAME	VOID
MODELPROC	WAIT
MODIFIERS	WARNING
MONOLITHIC	WHEN
NEW	WHILE
NIL	WRITE_LOCK
NOCHANGE	WRITE_TABLE
NODE	Z_TO_ONE
NODE_COUNT	Z_TO_ZERO
NODE_POINT	ZERO_TO_Z
NODE_POINT_PIN_ASSOCIATION	

All keywords shall be case sensitive. Each keyword has two valid syntactic forms: one using only uppercase letters, and one using only lowercase letters. Table 1 shows only the uppercase syntactic form.

7.2.3.2 Identifier

An *identifier* is the name that represents a value, except within the 'PATH, FROM, TO, BUS, 'TEST, INPUT, NODE and OUTPUT subclauses, where it is treated as a literal string. An *identifier* is a sequence of one or more characters starting with an alphabetic character, followed by zero or more alphanumeric or underscore () characters.

Syntax 7.2 presents identifiers in DCL.

```

identifier ::= identifier_first_character {<identifier_character>}
identifier_first_character ::= a-z | A-Z
identifier_character ::= a-z | A-Z | 0-9 | _

```

Syntax 7.2—identifier

An *identifier* is case sensitive. An *identifier* shall not be one of the following:

- A *keyword*
- A reserved word in the C or C++ languages (refer to ISO/IEC 9899:1990 and ISO/IEC 14882:2003)
- Any character sequence beginning with the letters *DCM* in any mixture of case

NOTE—It may be desirable to help ensure that some identifiers are universally unique so as to not collide when disparate library pieces from different companies are combined. Examples of such identifiers are METHOD names and TECH_FAMILY names.

7.2.3.3 Double quoted character sequence

A *double_quoted_character_sequence* has the semantics of an identifier.

The syntax for double quoted characters in DCL is given in Syntax 7.3.

```

double_quoted_character_sequence ::= "literal_character_sequence"
literal_character_sequence ::= <literal_character>{<literal_character>}
literal_character ::= any character in the ASCII character set except
    double_quote, newline, or carriage_return

```

Syntax 7.3—double_quoted_character

7.2.3.4 Predefined references to Standard Structure fields

DCL defines a set of identifiers to reference fields in the Standard Structure. These identifiers shall be visible in all scopes. See Table 2.

Table 2—DCL predefined references to Standard Structure fields

Predefined identifier (data type)	Set by DPCM or passed in by application	Description
BLOCK (PIN)	Passed in during model search, Passed in calculation.	The instance block identification in the design.
CALC_MODE (STRING)	Passed in calculation.	Indicates the type of calculation associated with the current propagate segment calculation: best case, worst case, or nominal.
CALC_MODE_SCALAR (INT)	Passed in calculation.	Indicates the type of calculation associated with the current propagate segment calculation: best case, worst case, or nominal.
CELL (STRING)	Passed in during model search, passed in calculation.	The cell of the circuit under calculation. The unique cell identification is the concatenation of CELL, CELL_QUAL, and MODEL_DOMAIN (separated by a

Predefined identifier (data type)	Set by DPCM or passed in by application	Description
		period).
CELL_DATA (CELL_DATA_TYPE)	Set during model search, passed in calculation.	Initial pointer to the caching system for cell-based store clauses.
CELL_QUAL (STRING)	Passed in during model search, passed in calculation.	The cell of the circuit under calculation. The unique cell identification is the concatenation of CELL, CELL_QUAL, and MODEL_DOMAIN (separated by a period).
CKTTYPE (STRING)	Set by DPCM.	Reserved for library developer's use.
CLKFLG (STRING)	Set during model search, passed in calculation.	Holds the clock flag to be assigned to this calculation.
CYCLEADJ (INT)	Set during model search.	Reserved for library developer's use.
EARLY_SLEW (FLOAT)	Passed in calculation.	Holds the slew associated with the propagate segment for the delay or slew to be calculated. This slew represents the SLEW of the earliest signal to arrive at the segment for which the DELAY or SLEW is being calculated.
FROM_POINT (PIN)	Set during model search, passed in calculation.	Holds the from point of the propagate segment for the delay or slew to be calculated.
FROM_POINT_PIN_ASSOCIATION (PIN_ASSOCIATION)	Set during model search.	Used during the modeling of a cell. Its general usage is to associate library information with the from pin.
FUNCTION (INT)	Set during model search.	Contains the function operator enumeration value.
INPUT_PIN_COUNT (INT)	Passed in during model search.	Holds the number of input pins on the circuit currently being modeled or under calculation.
LATE_SLEW (FLOAT)	Passed in calculation.	Holds the slew associated with the propagate segment for the delay or slew to be calculated. This represents the SLEW of the latest signal to arrive at the segment for which the DELAY or SLEW is being calculated.
MODEL_DOMAIN (STRING)	Passed in during model search, passed in calculation.	The cell of the circuit under calculation. The unique cell identification is the concatenation of CELL, CELL_QUAL and MODEL_DOMAIN (separated by a period).
MODEL_NAME (STRING)	Set during model search.	Holds the name of the modelproc that is currently in control.
NODE_COUNT (INT)	Passed in during model search.	Holds the number of nodes connected to the circuit currently being modeled or under calculation.
NODE_POINT (PIN)	Set during model search.	Holds the pin data structure associated with the node. Generally used during a DO statement.
NODE_POINT_PIN_ASSOCIATION (PIN_ASSOCIATION)	Set during model search.	Used to associate library data with the node.
OUTPUT_PIN_COUNT (INT)	Passed in during model search.	Holds the number of output pins connected to the circuit currently under calculation or model build.

Predefined identifier (data type)	Set by DPCM or passed in by application	Description
PATH (STRING)	Set during model search, passed in calculation.	Holds the path or test statement name.
PATH_DATA (PATH_DATA_TYPE)	Set during model search, passed in calculation.	Holds the pointer to the path and pin base cache
PHASE (STRING)	Set by DPCM.	Gives access to phase. This is a combination of the SOURCE_EDGE and SINK_EDGE. When the SOURCE_ EDGE and the SINK_EDGE are the same value, PHASE is set to I. If they are not the same value, PHASE is set to O.
PROCESS_VARIATION (STRING)	Passed in during calculation.	Indicates the process variation (setting to use).
PROCESS_VARIATION_SCALAR (INT)	Passed in during calculation.	Indicates the process variation setting to use.
REFERENCE_EDGE (STRING)	Set during model search, passed in calculation.	Indicates whether the calculation is for the rising, falling, or both edges.
REFERENCE_EDGE_SCALAR (INT)	Set during model search, passed in calculation.	Indicates whether the calculation is for the rising, falling, or both edges.
REFERENCE_MODE (STRING)	Set during model search, passed in calculation.	Indicates whether the calculation is for early mode, late mode, or both.
REFERENCE_MODE_SCALAR (INT)	Set during model search, passed in calculation.	Indicates whether the calculation is for early mode, late mode, or both.
REFERENCE_POINT (PIN)	Set during model search, passed in calculation.	Holds the from point of the propagate segment for the delay or slew to be calculated and is generally used in TEST and TABLEDEF statements.
REFERENCE_POINT_PIN_ ASSOCIATION (PIN_ASSOCIATION)	Set during model search.	Used to associate library data with a reference_point pin.
REFERENCE_SLEW (FLOAT)	Passed in calculation.	Holds the slew associated with the test segment for which the check is being analyzed. This slew represents the SLEW of the reference or "clock." This allows the REFERENCE_SLEW to alter the calculations performed by the CHECK statement. The CHECK statement can adjust the BIAS by an amount that is a function of the reference's slew.
SIGNAL_EDGE (STRING)	Set during model search, passed in calculation.	Indicates whether the calculation represents the rising edge, falling edge, or both.
SIGNAL_EDGE_SCALAR (INT)	Set during model search, passed in calculation.	Indicates whether the calculation represents the rising edge, falling edge, or both.
SIGNAL_MODE (STRING)	Set during model search, passed in calculation.	Indicates whether the calculation represents early mode, late mode, or both.
SIGNAL_MODE_SCALAR (INT)	Set during model search, passed in calculation.	Indicates whether the calculation represents early, mode, late mode, or both.
SIGNAL_POINT (PIN)	Set during model search, passed in calculation.	Holds the to point of the propagate segment for the delay or slew to be calculated and is generally used in TEST and TABLEDEF statements. This is used during test computations.

Predefined identifier (data type)	Set by DPCM or passed in by application	Description
SIGNAL_POINT_PIN_ ASSOCIATION (PIN_ASSOCIATION)	Set during model search.	Used to associate library data with the signal pin.
SIGNAL_SLEW (FLOAT)	Passed in calculation.	Holds the slew associated with the test segment for which the check is being analyzed. This slew represents the SLEW of the signal or “data.” This allows the SIGNAL_SLEW to alter the CHECK statement’s calculation. The BIAS computed by the CHECK statement can be a function of the signal’s slew.
SINK_EDGE (STRING)	Set during model search, passed in calculation.	Indicates whether the calculation represents the rising edge, falling edge, or both.
SINK_EDGE_SCALAR (INT)	Set during model search, passed in calculation.	Indicates whether the calculation represents the rising edge, falling edge, or both.
SINK_MODE (STRING)	Set during model search, passed in calculation.	Indicates whether the calculation represents early mode, late mode, or both.
SINK_MODE_SCALAR (INT)	Set during model search, passed in calculation.	Indicates whether the calculation represents early mode, late mode, or both.
SOURCE_EDGE (STRING)	Set during model search, passed in calculation.	Indicates whether the calculation is for the rising edge, falling edge, or both.
SOURCE_EDGE_SCALAR (INT)	Set during model search, passed in calculation.	Indicates whether the calculation is for the rising edge, falling edge, or both.
SOURCE_MODE (STRING)	Set during model search, passed in calculation.	Indicates whether the calculation is for early mode, late mode, or both.
SOURCE_MODE_SCALAR (INT)	Set during model search, passed in calculation.	Indicates whether the calculation is for early mode, late mode, or both.
TO_POINT (PIN)	Set during model search, passed in calculation.	Holds the to point of the propagate segment for which the delay or slew is to be calculated. It is used during delay and slew computations.
TO_POINT_PIN_ASSOCIATION (PIN_ASSOCIATION)	Set during model search.	Used to associate library information with a pin representing the to_point.

7.2.3.5 Compiler generated predefined identifiers

The DCL compiler generates the values for the predefined identifiers shown in Table 3.

Table 3—DCL compiler generated predefined identifiers

Predefined identifier (data type)	Description
COMPILATION_TIME_STAMP (STRING)	The time/date stamp when this rule was compiled.
CONTROL_PARM (STRING)	Gives access to the CONTROL_PARM value specified in the SUBRULE statement that loaded this rule.
RULE_PATH (STRING)	Gives access to the RULE_PATH value specified in the SUBRULE statement that loaded this rule.
TABLE_PATH (STRING)	Gives access to the TABLE_PATH value specified in the SUBRULE statement that loaded this rule.

7.2.3.6 Constant

A constant is defined as a sequence of values that do not change during the execution of the program. DCL constants can either be a simple value such as a string, a floating point, an integer, a number, or a complex sequence of values such as an array or a structure.

The syntax for constants in DCL is given in Syntax 7.4.

```
constant ::= string_literal
          | floating_point_constant
          | double_constant
          | integer_constant
          | complex_constant
          | aggregate_constant
          | NIL

string_constant ::= '{any legal character except new_line}'
complex_constant ::= ( real_part , imaginary_part )
real_part ::= floating_point_constant
imaginary_part ::= floating_point_constant
aggregate_constant ::= {array_constant_declare |
                        structure_constant_declare}
structure_constant_declare ::= structure_type_name structure_constant
structure_constant ::= abstract_structure_constant |
                      known_structure_constant
known_structure_constant ::= { field_value { , field_value } }
abstract_structure_constant ::= name_of_structure_type :
                              known_structure_constant
field_value ::= string_literal
             | floating_point_constant
             | integer_constant
             | complex_constant
             | structure_constant
             | array_constant
             | name_of_statement
array_constant_declare ::= base_type
                        | structure_type_name [ '*' { , '*' } ] array_constant
array_constant ::= [ dimension_list { , dimension_list } ]
dimension_list ::= ( constant { , constant } )
                | ( [ dimension_list { , dimension_list } ] )
```

Syntax 7.4—constant

The *floating_constant*, *double_constant*, and *integer_constant* tokens have the same definition as defined in the ISO C standard.

7.2.3.6.1 Predefined constant NIL

NIL is a predefined constant of ISO C type void * with a value of (void *) 0, which shall indicate the PINLIST, PIN, STRING, ARRAY, or VOID has no value defined.

7.2.3.6.2 Edge type enumerations

The values for the SINK_EDGE and SOURCE_EDGE predefined identifiers are enumerations of the edge values as shown in Table 4. Use of these reserved words results in the corresponding string values in the generated code.

The remaining edge types defined in a DCM_EdgeTypes structure: SAME, BOTH, ALL, COMPLIMENT, and TERMINATEBOTH shall be considered illegal enumeration values for SINK_EDGE and SOURCE_EDGE.

Table 4—Edge types and conversions

Keyword	Definition	String value	enum Value
FALL	The FALL edge means a signal transitions from a high level to a low level. The levels in DCL are arbitrary as there are no specified reference levels because they are implied in the calculation.	'F'	1
ONE_TO_Z	ONE_TO_Z means the signal is transitioning from a high level to a high impedance state.	'1Z'	7
RISE	The RISE edge means a signal transitions from a low level to a high level. The levels in DCL are arbitrary as there are no specified reference levels because they are implied in the calculation.	'R'	0
TERM	TERM means the edge terminates and does not propagate.	'T'	5
Z_TO_ONE	Z_TO_ONE means the signal is transitioning from a high impedance state to a high level.	'Z1'	8
ZERO_TO_Z	ZERO_TO_Z means the signal is transitioning from a low level to a high impedance state.	'0Z'	9
Z_TO_ZERO	Z_TO_ZERO means the signal is transitioning from a high impedance state to a low level.	'Z0'	10

7.2.3.6.3 Propagation type enumerations

SINK_MODE and SOURCE_MODE are converted to enumerations as shown in Table 5.

Table 5—Propagation mode conversions

Keyword	Definition	String value	enum value
EARLY	The first edge that propagates through a given cone of logic.	'E'	0
LATE	The last edge that propagates through a given cone of logic.	'L'	1

7.2.3.6.4 Calculation mode enumeration

CALC_MODE values are converted to enumerations as shown in Table 6.

Table 6—Calculation mode conversions

Definition	String value	enum value
Best case	'B'	0
Worst case	'W'	1
Nominal case	'N'	2

7.2.3.6.5 Test type enumeration

TEST_TYPE values are converted enumerations as shown in Table 7.

Table 7—TEST_TYPE conversions

Keyword	Definition	enum value
CGHT	Clock gating hold test	7
CGPW	Clock gating pulse width test	6
CGST	Clock gating setup test	8
CPW	Clock pulse width test	2
CST	Clock separation test	3
DHT	Data hold Test	10
DIFFERENTIAL_SKEW	Differential skew test	15
DPW	Data pulse width test	4
DST	Data setup test	5
HOLD	Hold test	1
NOCHANGE	No change test	14
RECOVERY	Recovery test	11
REMOVAL	Removal test	12
SETUP	Setup test	0
SKEW	Skew test	13

7.2.3.7 String literal

A *literal_character* is any member of the character set except the single quote ('), backslash (\), or *newline_character*. Each of these restricted characters may be present in a single quoted string if it is preceded by a backslash.

The syntax for string literals in DCL is given in Syntax 7.5.

```
string_literal ::= '{<literal_character>'}
```

Syntax 7.5—string_literal

string_literals follow the same semantics as ISO C string literals (see Section 6.1.4 of ISO/IEC 9899:1990).

7.2.3.8 Operators

The syntax for DCL operators is given in Syntax 7.6.

```
operator ::= $ | ^ | * | / | + | = | ** | || | && | ! | == | !=  
| > | < | >= | <= | -> | <- | <-> | ->X<- | <-X-> | NEW  
| EVAL | :: | . | :> | :^ | ` | `& | `~ | `> | `< | `^
```

Syntax 7.6—operator

7.2.3.9 Punctuator

The syntax for punctuators is given in Syntax 7.7.

```
punctuator ::= ( | ) | [ | ] | { | } | , | ; | : | . | # | &  
| < | > | %{ |}% | ' | "
```

Syntax 7.7—punctuator

The punctuators (), [], %{ }%, and { } shall occur in balanced pairs. Quotation marks ' and " shall also appear in pairs surrounding the name or string they are enclosing.

7.2.3.10 Name

A *name* is either an *identifier* or a *double_quoted_character_sequence*. The use of these punctuators is incorporated within the subsequent syntax charts in this subclause.

7.2.4 Header names

Preprocessing tokens for header names shall only appear with a `#include` preprocessing directive. The header name shall be defined the same as in Section 6.1.7 of ISO/IEC 9899:1990.

7.2.5 Preprocessing directives

DCL preprocessing directives are exactly the set defined in Section 6.8 of ISO/IEC 9899:1990 and have the same semantics.

7.3 Context

Each program or library shall consist of one or more contexts. A context represents a separation of execution into logically different streams. A context shall consist of a unique combination of a space and a plane. Execution in one context shall not alter the computation in another context except where explicitly directed.

7.3.1 Space

A space is a system of one or more modules linked together to form an execution system. There may be zero or more spaces resident at any time. Spaces can be loaded or unloaded as needed. Multiple spaces may share one or more modules from another space. The system shall not load the same module more than once. If a module is used in more than one space, each module shall appear to the programmer as a separate and independent copy.

7.3.2 Plane

A plane separates data associated with a space. Each space shall have one or more planes, and the data on one plane shall be independent of any other plane except where explicitly directed. During execution as many planes as needed may be created within a space.

7.3.3 Context operation

A context shall represent an independent view of variables and executable modules.

The Standard Structure shall contain the context identifier. The context identifier identifies the context the associated function should be executed in.

7.3.4 Library parallelism

A context shall be able to accept the assertion of function requests from other contexts. When one context assigns another context, a function to perform the function is executed on the assigned context using that context's variables. When a context has completed its assigned functions, it remains available for future assignments.

The execution shall proceed in the order the function requests were received by the executing context. If the function to be executed returns results, those results shall not be accessible to the requesting context until the executing context has finished executing the requested function. One context requesting another context to execute a function shall not block the requesting context.

7.3.5 Application parallelism

Whatever parallelization method is chosen by the application, the application developers should plan for library implementations that might be thread based. The application is in control of its parallelization, and the library has no responsibility in managing it. Applications can use child processes to accomplish parallelization. It shall be an error for the application to spawn a child process for the purposes of parallel execution of a library if the following conditions exist:

- a) The application does not use shared memory. All memory used both by the application and the library shall be shared. Furthermore, if the application asserts a memory manager on the library, that memory manager shall supply shared memory.
- b) If the library attempts to load additional modules. It shall be the responsibility of the application to use only libraries that have loaded all their modules before the fork. DCL uses function pointers to represent its statement types. These pointers may be undefined between processes if the load has occurred after the fork.

It is the responsibility of the application designer to be aware of these limitations when designing the application. The library has no knowledge of the application's intended parallelism, and the library cannot be prevented from loading a module or creating its own contexts.

The application may also request that the library create contexts on its behalf. The application may use these contexts by calling an expose statement passing as the first argument a Standard Structure that contains one of these contexts. Parallel operation may be obtained by exercising each context on a different thread. The application shall not attempt call a context while that context is busy performing other work for the application.

An application thread is allowed to change contexts as it executes. Application threads shall only change contexts during a call to one of the module's primary entry points, those statements whose pointers are exchanged during `dcmRT_bindRule`, or run-time library support functions.

7.4 Data types

DCL supports three categories of data types, which include *native*, *aggregate*, and *array*. The native data types are defined in terms of the ISO/IEC 9899:1990 data types. See Table 103 in ISO/IEC 9899:1990 for details of the mapping between DCL and ISO C data types.

7.4.1 Base types

Native types are passed by value (see 7.4.2). It shall be the library's responsibility to help ensure that application data types adhere to the lifetime constraints (see 7.5.2).

NOTE—If a string received from the application on a call is passed to a different context, the originating function shall wait until the requested context completes before returning to the application.

7.4.2 Native data types

The syntax for native data types in DCL is given in Syntax 7.8.

```
native_type ::= mathematical_type | pointer_data_type
```

Syntax 7.8—native_type

7.4.3 Mathematical calculation data types

The syntax for calculation data types in DCL is given in Syntax 7.9.

```
mathematical_type ::= DOUBLE | FLOAT | INT | CHAR | SHORT
                   | LONG | COMPLEX
```

Syntax 7.9—mathematical_type

7.4.3.1 C types

Types DOUBLE, FLOAT, INT, CHAR, SHORT, and LONG assume the type definition defined by the C programming language for the target machine.

7.4.3.2 COMPLEX type

The COMPLEX type represents a complex number consisting of a real component and an imaginary component. The real and imaginary values are represented as two numbers of type double.

7.4.4 Pointer data types

The syntax for pointer data types in DCL is given in Syntax 7.10.

```
pointer_data_type ::= STRING | PIN | PINLIST | VOID
```

Syntax 7.10—pointer_data_type

7.4.4.1 STRING

STRING type is a pointer to the first character in a sequence of ASCII characters, which is terminated with a zero immediately after the last character of that sequence.

7.4.4.2 PIN

PIN is a pointer to an arbitrary application structure whose first member is of type STRING.

7.4.4.3 PINLIST

PINLIST is a consecutive sequence of PIN types terminated by a zero immediately after the last valid pin in the sequence.

7.4.4.4 VOID

VOID represents arbitrary data that are not type defined. This type is useful when interacting with C-based applications or embedded C code that is not using DCL based structures.

7.4.5 Aggregate data types

Aggregate data include a collection of one or more data elements represented by a single identifier. There are two classes of aggregate data types: structures and arrays. In the case of arrays, all the data elements shall be of the same type. In the case of structures, the data elements may vary in type. The syntax for aggregate data types is given in Syntax 7.11.

```

aggregate_type ::= result_type | statement_type
result_type_statement_members ::= (
    assign_statement_name
  | calc_statement_name
  | expose_statement_name
  | external_statement_name
  | internal_statement_name
  | tabledef_statement_name
  | typedef_statement_name
  | ABSTRACT )
result_type ::= ([structure_var] [SYNC | SHARED]
    result_type_statement_members [field_var] | [structure_var]
    result_type_statement_members [TRANSIENT])
statement_type ::= result_type ( ) [PURE | IMPURE]
    [CONSISTENT | INCONSISTENT] [LAUNCHABLE]

```

Syntax 7.11—aggregate_type

The aggregate name shall represent the address of the data. Accessing aggregate data member is given in Syntax 7.12.

```

aggregate_access ::=
    aggregate_name [ [ integer_expression { , integer_expression } ] { .
        ( field_name | statement_type_access ) } ]
statement_type_access ::=
    field_name [ [ index_expression { , integer_expression } ] ] (
        comma_expression_list )

```

Syntax 7.12—aggregate_access

7.4.5.1 Result types

RESULT clauses in statements and DATA clauses in TABLEDEFs define aggregate data types that shall be analogous to structures in ISO/IEC 9899:1990. The name of the associated statement or table can be used as the name of the aggregate data type. The RESULT type field ordering is defined in 7.9.2.3.1 and 7.9.4.

Example

```

calc(resultType): passed(int:x)
    result(double:fp=2*x+3 & integer:x+1);
calc(example): passed(resultType:x)
    result(double:x.fp*4 - 2*x);
calc(exercise): result(double:example(resultType(5)));

```

7.4.5.1.1 TRANSIENT attribute

Structures can be *backed*, where memory has been allocated by the system, or *transient*, which represents structures that have their memory representation as part of the statement's stack space. For structures created with the transient attribute, the space resides in the statement that created the structure. When that statement concludes, the memory is automatically returned to the system. Backed structures may be assigned or arguments to a transient target. In these situations, no memory claim counting shall occur. The resulting structure of a statement is a transient structure.

In some situations, TRANSIENT structure variables and non-TRANSIENT structure variables have to be assigned to target variables or passed as arguments. The following rules govern the assignments to TRANSIENT structure slots:

- a) TRANSIENT variables can be assigned the entire result structure of a statement reference.
- b) TRANSIENT variables can be passed as parameters to arguments accepting a TRANSIENT structure of a compatible type.
- c) Non-TRANSIENT structure variables can be passed as parameters to arguments accepting a TRANSIENT structure of a compatible type.
- d) TRANSIENT variables shall not be marked field VAR.
- e) Structure definitions shall not contain other structures marked transient.

7.4.5.2 Abstract type

The abstract type shall match any result type that does not have the transient attribute. The compiler shall have a mode that verifies during the execution of the program that all variable assignments of abstract type to a result type or passed as arguments identified as a result type are compatible with the target. Compatible shall mean that the abstract structure may have more fields than the target, but all fields present in both are in the identical order as those contained in the target and that each field shall match in type and have compatible var permissions and other attributes.

7.4.5.3 Statement types

A statement type represents a reference to any statement having the compatible PASSED arguments and RESULT sequences.

Statement types may have attributes associated with them, as follows:

- PURE indicates the associated statements are pure. It shall be an error to assign an impure statement to a statement type possessing the pure attribute.
- CONSISTENT indicates the associated statements are consistent. It shall be an error to assign an inconsistent statement to a statement type possessing the consistent attribute.
- IMPURE indicates the associated statements are to be treated as impure. Both impure and pure statements may be assigned to a statement type possessing the impure attribute. In addition, it shall be an error to use the consistent attribute in conjunction with the impure attribute.
- INCONSISTENT indicates the associated statements are inconsistent. Both consistent and inconsistent statements can be assigned to statement types possessing the inconsistent attribute.
- LAUNCHABLE indicates the statement may be given to other contexts to process.

Compatible passed or result argument list shall mean the same number of arguments or results where each argument is a matching type including var permissions and other attributes and in the order of the defining statement to any level of structure or array nesting.

This enables the passing of statement pointers into other statements.

Example

```
exercise() passes statementType1() to example() as a aggregate type:
calc(statementType1): passed(int:x)
  result(int:resType(x)+ 3*x);
calc(statementType2): passed(int:z)
  result(integer:resType(z)+6*z+ 1);
calc(example): passed(statementType1(): y)
  result(int: y(5));
calc(exercise):
  result(int:example(statementType1)+ example(statementType2));
```

7.4.5.3.1 Array types

The syntax for array types in DCL is given in Syntax 7.13.

```
array_type ::= [aggregate_type
  | mathematical_type
  | STRING | PIN | VOID]
dimension_declaration {dimension_declaration}
dimension_declaration ::= [SHARED] [array_var]
  [ dimension_list ]
dimension_list ::= variable_dimension_list
variable_dimension_list ::= * { , * }
```

Syntax 7.13—array_type

An array type describes a contiguously allocated nonempty set of objects with a particular member object type, which is called the *element type*. Array types are modeled by their element type, their number of dimensions, and the number of elements in each dimension of the array.

An array dimension shall be *variable*. The number of elements in a *variable* dimension shall not be specified when the array is declared and may vary during program execution.

The following is true for all array types:

- a) Arrays can have an arbitrary number of dimensions up to a limit of 255. For any particular array, all of its dimensions shall have the same type—fixed or variable.
- b) The length of each dimension in a multidimensional array shall not depend on the index used for any of the other dimensions of that array—“ragged” arrays shall not be supported.
- c) Array indexes shall start with zero (0) in each dimension.
- d) Array types shall be passed and returned as pointers to a contiguous memory region. The array data shall be organized as defined by ISO/IEC 9899:1990.

Array types shall be constant unless declared to be VAR. A VAR array may have its elements changed during program execution. An array type that has the VAR declaration may be passed as an argument to a statement requiring a constant array. A constant array type shall not be passed as an argument to a statement requiring a VAR array type.

The array VAR modifier indicates array elements may be altered by statements other than the statement that created the array. Assignments for non-var to non-var, var to var, or var to non-var are effected through pointer manipulation; no copying of array values is performed.

Arrays may contain other arrays. The notation for declaring an array of arrays is a list of dimension declarations. Each dimension declaration represents an array contain within the array with the preceding dimension declaration.

7.4.5.3.2 Modification of data

Statements define data by sequence of types and variable names listed in either the local, result, or data clauses. Structure types are considered to have nested defining structures in which each level of nesting represents a type sequence of the preceding level's result or data clause. The nesting depth continues until a sequence of types consists solely of native types.

A statement shall have permission to modify the data it defines. A statement shall have the right to access any data for the purpose of reading. A statement shall have permission to modify data it did not define if the defining statement has given permission.

7.4.5.3.2.1 Var permissions

VAR indicates that permission is given to modify data by other statement. There are three levels of VAR permission (Syntax 7.14). Structure var gives others permission to modify the contents of a structure consistent with the defining structure's var settings. Array var allows the contents of an array to be modified. Field var allows the value of the identified field to be modified by statements other than the defining statement.

```
structure_var ::= VAR  
array_var ::= VAR  
field_var ::= VAR
```

Syntax 7.14—var

Example

```
FORWARD CALC(name) :  
  RESULT(INT[*,*]: constIntArray & NUMBER VAR [*]: varNumVec);
```

7.4.5.3.3 Type conversions

Type conversions shall be performed based on the following:

- Changing the type of an argument to the type of the expected parameter
- Changing the type of an expression term to the type of other terms in the same expression

Only the conversions enumerated in the following subclauses are valid in DCL.

7.4.5.3.3.1 Implicit conversions

The following implicit type conversions shall be performed:

- CHAR, SHORT, INT, and LONG types

Calculations using a mixture of CHAR, SHORT, INT, and LONG shall be automatically converted to the type with the greater number of bits. Assignments of types with a greater number of bits to a type of a lesser number of bits shall result in a truncation. An assignment of types with a lesser number of bits to a target of a greater number of bits shall result in the sign bit being propagated across the unmatched bits of the longer type.

b) *INT* to *DOUBLE* or *FLOAT*

An *INT* value shall be converted to *DOUBLE* (*FLOAT*) when the expected parameter or expression term has type *DOUBLE* (*FLOAT*). An *INT* value shall be converted to *DOUBLE* when a division operator is present in the expression.

c) *FLOAT* to *DOUBLE*

A *FLOAT* value shall be converted to *DOUBLE* when the expression term has type *DOUBLE*.

d) *PIN* to *STRING*

A *PIN* value shall be converted to *STRING* when the expected parameter or expression term has type *STRING*.

e) *native_array_type*, *pointer_data_type* or *aggregate_type* to *VOID*

A *native_array_type*, *pointer_data_type*, or *aggregate_type* value shall be converted to *VOID* when the expected parameter or expression term has type *VOID*.

7.4.5.3.3.2 Explicit conversions

The following explicit type conversions shall be performed:

a) *AGGREGATE* data type to *AGGREGATE* data type

An *AGGREGATE* data type shall be converted to an equivalent *AGGREGATE* data type when the target parameter has the same complete structure or a contiguous subset of the same structure.

b) A complete structure shall mean the source and target have the same number of elements, each being the same type, in the same order, with the same var permissions, with the same attributes such as sync or shared, nested to any number of levels.

c) A contiguous subset shall mean source has more elements than the target but for the targets element sequence the source shall match it as defined in the complete structure beginning with the first element.

Example

```
CALC(resType): passed(int:x)
    result(double:fp = 2*x+3 & int:x+1);
CALC(resType2): passed(int:x)
    result(double:y=2**x & int:z=x-1);
CALC(example): passed(resType:x)
    result(double:x.fp * 4 - 2 * x);
CALC(exercise): result(double:example(resType(5))+example(resType2(3)));
```

7.4.5.3.3.3 Abstract type conversion

ABSTRACT type may be assigned or converted to any aggregate target with compatible var permissions. There shall exist an optional run-time check to help ensure the assignment of the contents of the abstract source are compatible with the contents of the target. A structure of a know type can be the source to a target of abstract type.

In some situations, the compiler cannot determine the intended source type, such as when accessing a field within an abstract structure. To enable this feature, a casting operation is allowed on abstract types.

Cast syntax is described in Syntax 7.15.

```
cast ::= type_definition : abstract_variable_name | nested_cast  
nested_cast ::= ( type_definition : abstract_variable_name )  
               . fieldName  
               | ( type_definition : nested_cast ) . fieldName
```

Syntax 7.15—cast

7.4.5.3.3.4 Transient conversions

Assignments of transient structures to backed targets shall result in a copy of the structure. Any nested structures and arrays contained in the transient source are not copied. Assignments of backed structures to transient targets are pointer assignments, and there shall be no modification of the source structure's reference count.

7.4.5.3.3.5 SHARED attribute

A data attribute of SHARED indicates that the memory reference management system shall serialize the reference counting for these aggregates. No other serialization is required. It is the responsibility of the library developer to help ensure the algorithms employed do not attempt simultaneous update or access. The results of simultaneous update or access could be unpredictable.

7.4.5.3.3.6 SYNC attribute

A data attribute of SYNC indicates that access for both reference management system and data references shall be serialized. Arrays shall not have the sync attribute.

7.5 Identifiers

Identifiers are named entity within the program.

7.5.1 Name spaces of identifiers

If more than one declaration of a particular identifier is visible at any point in the translation unit, the syntactic context determines proper reference to different entities. Thus, there are separate name spaces for various categories of identifiers, as follows:

- Identifiers declared in RESULT clauses
- Identifiers naming methods and technology families
- All other identifiers

7.5.2 Storage durations of objects

An object has a storage duration that determines its lifetime. The three storage durations are static, automatic, and managed.

An object defined by an ASSIGN statement or whose identifier is declared with external or internal linkage shall have static storage duration. For such an object, storage shall be reserved prior to program start-up. The object shall exist and retain its last-stored value throughout the execution of the entire program.

An object defined by a STORE clause shall have static storage duration. For such an object, storage shall be reserved when the enclosing MODELPROC statement is evaluated. The object shall exist and retain its last-stored value throughout the execution of the entire program.

All identifiers and aggregate objects that are not managed shall have automatic storage duration. For such objects, storage shall be allocated when the defining statement is called and shall persist until the defining statement goes out of scope.

For each managed object, such as an array or a structure, storage shall be allocated using the NEW operator and shall persist as long as either the application or the DPCM maintains a claim on that object. The DPCM shall create a temporary claim on each managed object that is passed or returned to the application. This claim shall persist until the application either calls a DPCM function or returns control to the DPCM. At that time, the claim may be removed by the DPCM.

7.5.3 Scope of identifiers

An identifier shall be visible (i.e., can be used) only within a region of program text referred to as its “scope.” The 12 different types of scope are as follows:

- a) Space
- b) Plane
- c) TECH_FAMILY
- d) Global
- e) Subrule
- f) Statement-prototype
- g) Statement
- h) Modeling procedure
- i) STATEMENTS
- j) RESULT
- k) PASSED
- l) Discrete

When lexically identical identifiers exist in the same name space, identifiers in outer scopes shall be hidden until the inner scope terminates. The DCL scope hierarchy is as follows:

Space
Plane
TECH_FAMILY
Global
Subrule
Statement-prototype
Statement
RESULT
PASSED
Discrete
Modeling procedure
STATEMENTS
Statement

The scope of an identifier shall start where the identifier is first declared and extend to the end of the scope in which it was declared. An identifier shall be declared before it is referenced. Multiple MODELPROC statements of the same name shall not occur within the same TECH_FAMILY.

7.5.4 Linkages of identifiers

A statement identifier declared in different scopes can be made to refer to the same object or statement by a process called “linkage.” Four subrule scope options affect linkage: EXPORT, IMPORT, FORWARD, and OPTIONAL. Expose chaining also defines how EXPOSE identifiers are linked within a TECH_FAMILY’s scope (see 10.8.3.1).

7.5.4.1 EXPORT

A statement identifier shall be made visible outside its declared subrule scope and within the same TECH_FAMILY by using the EXPORT option on a statement definition.

7.5.4.2 IMPORT

A statement identifier EXPORTED in one subrule scope shall be made visible within another subrule scope and within the same TECH_FAMILY by using the IMPORT option on a statement prototype.

7.5.4.3 FORWARD

A statement identifier referenced in a subrule scope before its definition (within the same scope) shall be made visible by using the FORWARD option on a statement prototype.

7.5.4.4 OPTIONAL

Imported or EXPOSE statements that have the OPTIONAL modifier shall be linked with an error code generating function if the defining function cannot be linked during execution. A DPCM error code generator shall receive a return code of severity 2 from the reference.

7.5.4.5 Chaining of EXPOSE identifiers

EXPOSE statement identifiers with the same name in separate subrules within the same TECH_FAMILY are chained together at run time. The application and DPCM are presented with a single EXPOSE entry point for this identifier. The first EXPOSE statement in a chain is always used as the reference to any EXPOSE statement within a particular subrule (even if there is an EXPOSE statement defined within the subrule containing the reference).

7.6 Operator descriptions

This subclause details the operators used in DCL.

7.6.1 String prefix operator

This subclause details the use of string prefix operators.

7.6.2 Explicit string prefix operator

The explicit string prefix operator is a unary operator that modifies the semantics of logical comparisons between string operands. This operator can be used with string operands within logical expressions.

The explicit string prefix operator is the * (asterisk) character. This operator shall occur as a separate token that precedes a string literal or an identifier having a string value.

7.6.3 Embedded string prefix operator

The embedded string prefix operator is a unary operator that modifies the semantics of logical comparisons between string operands. This operator can be used with string operands within logical expressions, table qualifiers, and table references.

The embedded string prefix operator is the * (asterisk) character. This operator shall occur as the first character of the string literal, the first character of a name, or the first character of the value of an identifier. To disable this special meaning of an asterisk, it shall be preceded with a backslash character (\).

7.6.4 String prefix semantics

The semantics of the explicit and embedded string prefix operator are identical. When the string prefix operator is present, the logical comparison shall be performed considering only a subset of the characters in the strings. The comparison shall start with the first character after the prefix operator, if present, in each string and shall proceed for the length of the string with the string prefix operator. If both strings have the prefix operator, then the comparison shall proceed for the length of the shorter string.

If the explicit string prefix operator is used and its operand contains a leading asterisk, this leading asterisk shall *not* be considered an operator; it shall be treated as the first character of the string.

7.6.5 Assignment operator

For native types, except arrays, the assignment operator (=) shall assign the expression value on the right to a variable on the left. For arrays, this operator shall make the variable on the left (which shall be an array name) a reference to the array on the right.

7.6.6 New operator

The new operator (NEW) shall be used to create memory space for aggregate types. NEW is used when a new instance of an array or structure is needed. After the NEW operator creates the space for an array, the values of the individual members shall be undefined for members that are not an address and a value of zero for all members that are an address (Syntax 7.16).

```
new_operator ::= NEW ( [AGGREGATE] type_definition
    [, destructor_name] )
```

Syntax 7.16—new_operator

7.6.6.1 Memory management

DCL maintains a reference counting system for all aggregate types not marked TRANSIENT. The reference count represents the number of identifiers that have a valid reference to the aggregate type. When an aggregate type is assigned to a an identifier, the aggregate type the identifier was referencing has its reference count decremented and the newly assigned aggregate has its reference count incremented.

When the reference count of an aggregate becomes zero, its memory is returned to the system. When an aggregate is returned to the system, the reference counts of the other aggregate members it may contain are decremented.

When an aggregate is passed as an argument to a statement, the reference count of that aggregate is incremented. When the statement concludes, the reference count is decremented. This helps ensure for the life of the called statement the aggregate shall be valid and not returned to the system.

Aggregates passed or returned to the application have their reference counts incremented. When the application reenters the library at the same level or lower on the program stack, the reference count associated with the aggregate is decremented.

7.6.6.2 AGGREGATE directive

The aggregate directive may only be used when allocating arrays of aggregate types. When the aggregate directive is used, the new operator creates both the array and aggregate members the array contains. The aggregate directive does not create aggregates nested within the array aggregate members. In situations where the array member is an aggregate type that contains one or more nested aggregates, the nested aggregate values are nil.

7.6.6.3 Destructor statements

Destructor statements are statements that have the responsibility of returning the space of an object that no longer has any active references remaining. These statements are only required when the objects are self-referencing either directly or indirectly. The statement defined to be a destructor shall accept as the only argument the aggregate defined within the new operator and have no result values. These statements are called when the reference count for the allocated aggregate is decremented to zero.

Example

This typically occurs when a circular system of references exists and a reference to the circle exists from another identifier. When the last identifier is removed that points to the circle, the circular list remains because those aggregates in the circle all have at least one reference; however, there is no reference remaining to the circle. To help ensure proper memory utilization, construct an object that contains the circular system as a member and the object itself is not a member of a circular system itself. When creating this object, identify a destructor statement that is designed to remove the circular system member. When the last reference to the object is removed, the destructor is called and the circular system is removed.

7.6.7 SCOPE operator(s)

Scope operators are used to change the frame of reference in a program (see Syntax 7.17). The scope operator may change the cached data reference, the tech_family referenced, or the context referenced. The type of reference changed depends on the data type used as the left argument preceding the operator called the “scoping prefix,” as follows:

- `::` Scope operator causes the identified item to be referenced from the scope contained in the prefix without changing the current context for the duration of this call or reference. The `::` scope operator can change the recall reference, the tech_family, or the space used for a statement reference.
- For recalling stored information or calling a method statement associated with either the PATH_DATA or CELL_DATA. To recall information, the statement used to store the data shall be visible to statement making the recall. To call a method statement, the method statement shall be visible to the statement making the call. This usage of this operator takes a scoping prefix of either a PATH_DATA or CELL_DATA type, and the right operand is either a statement’s name that performed the caching as part of the store operation or a method statement call. The scoping prefix `PATH_DATA::statement_name` or `CELL_DATA::statement_name` accesses cached information associated with the statement that cached its result values as part of a store operation. The scoping prefix `PATH_DATA::method_name()` or `CELL_DATA::method_name()` determines what action statement is associated with a method (7.8.4), and what the value of a store variable is, based on the value of the corresponding predefined identifier PATH_DATA or CELL_DATA (7.2.3.4).

- To call statements in other tech_families, use a scope prefix of type TECH_TYPE. The format of a typical cross tech_family call would look like TECH_TYPE::external_statement_name ([expression {comma_expression_list}]).
- To call statements in other spaces, use a scope prefix of type STD_STRUCT that contains the desired space context setting. The format of a typical cross tech_family call would look like STD_STRUCT_TYPE::external_statement_name(). This form of the scope operator the Standard Structure used shall contain a valid TECH_FAMILY setting in the called context.
- >: Changing Scope operator performs a similar operation to the scope operator :: but causes the identified statement to be referenced with the context contained in the left argument. When the called statement executes, the Standard Structure shall have the context associated with the left argument. When the statement called completes, the context shall be returned to that of the caller regardless of whether an error is returned.

```
scope_change ::= (STD_STRUCT_TYPE | TECH_TYPE) (:: | >:)
               external_statement_reference
recall ::= [ (PATH_DATA | CELL_DATA) :: ]
           statement_name { . field_name }
```

Syntax 7.17—scope_change

7.6.8 Launch operator

The launch operator ^: causes a statement to be executed to another context. The context the statement is assigned to is called the “target context,” and it shall process the statements in the order they were received. The target context continues to process statements until there are no more statements left to process. When there are no more statements left to process, the target context remains available but idle waiting on the next statement to execute.

Any context may launch a statement on any other context at any time. There is no implied order between the statements launched from different contexts. If a context launches a statement whose target context is the launching context, the statement executed immediately.

The Standard Structure on the left of the operator that represents the context the function on the right side of the operator shall have been declared SYNC. The statement identified on the right of the launch operator may have arguments. Aggregate arguments shall be sync or shared.

The results of a statement launched, if any, shall have the sync attribute automatically associated with it. The result structure is locked at the time of launch preventing access until the statement completes, at which time the structure is unlocked.

Launch syntax is described in Syntax 7.18.

```
launch ::= std_struct_variable ^: statement_reference
```

Syntax 7.18—launch

7.6.9 Purity operator

A reference to an impure statement shall be treated as PURE if the reference is preceded by the purity operator (^). The purity operator is defined in Table 8.

Table 8—Purity operator

Symbol	Definition
$\wedge$	The unary purity operator declares the statement reference immediately following it to be PURE, thereby overriding any default assumptions based on the optimization rules. References prefixed with the purity operator shall be optimized as a PURE reference.

7.6.10 Force operator

FORCE allows the programmer to override the conservative nature of the language in certain situations (see Syntax 7.19). FORCE shall only be allowed on operations in which the override is limited to the action being overridden with force. The use of FORCE asserts that the programmer is assuming responsibility for the correctness of this action.

```
forcedTarget ::= FORCE ( nested_reference )
forcedType  ::= FORCE ( type_declaration )
forcedParameter ::= FORCE ( parameter_expression )
```

Syntax 7.19—FORCE operator**Examples**

- An aggregate member without the proper var permissions can have its value changed using the FORCE operator, but FORCE shall not allow the assignment of a non-var aggregate member to a var target, as errors caused from this action could appear far away from the force operator.
- Restricted aggregates can be passed to a launch target specifying transient arguments using the FORCE operator, but FORCE shall not allow a transient aggregate to be passed to a launch target.
- FORCE shall allow a SYNC or SHARED aggregate to contain other restricted aggregates but not a transient structure.

The EVAL operator converts a pin range into an one-dimensional array of strings where each string contains a name of a pin contained within the pin range. The array shall contain the same number of elements as there are pins in the expanded pin range.

7.7 Timing propagation

Static timing includes the need to resolve timing when two or more input signals converge at a node. For each signal, a time period (window) can be defined based on the earliest and latest possible times when that signal can arrive at the node. The resolution process at a node determines a window for the node's output signal based on the windows of the input signals for that node.

DCL supports five ways (modes) to perform this resolution, which are defined in Table 9 and are used in PROPAGATE, EDGE, and TEST clauses.

Table 9—Timing resolution modes

Symbol	Definition
$\rightarrow$	Early(output) = earliest of all early(input) Late(output) = early(output)
$\leftarrow$	Late(output) = latest of all late(input) Early(output) = late(output)
$\leftrightarrow$	Early(output) = earliest of all early(input) Late(output) = latest of all late(input)

Symbol	Definition
->X<-	Early(output) = latest of all early(input) Late(output) = latest of all late(input)
<-X->	Early(output) = earliest of all early(input) Late(output) = earliest of all late(input)

7.7.1 Timing checks

Static timing includes the need to compare the windows of different signals at different nodes. One signal is always chosen as the reference. The comparison done is determined by a combination of test type, test type enumeration, and test mode operator (see Table 10 and 7.16.5.4.3).

Table 10—Test mode operators table

Symbol	Definition
->	Signal shall arrive later than reference + bias Bias shall be computed using early signal and late reference values.
<-	Signal shall arrive earlier than reference – bias Bias shall be computed using late signal and early reference values.
<->	Signal shall arrive earlier than reference – offset1, and signal shall arrive later than reference + offset2

There are four possible window-edge comparisons—the early or late edge of the signal window can be compared with the early or late edge of the reference window. The test mode operand refines the window comparison by specifying which ends of the windows to compare, along with some additional semantics.

Typically, two windows are maintained for each signal, corresponding to rising and falling transitions. The EDGE clause selects which window is used for the comparison.

7.7.2 Test mode operators

This subclause details the use of test mode operators.

7.7.2.1 CGHT

The CGHT is similar to the HOLD test mode operator except the reference edge shall be from a clock pin and the signal shall be from a data pin.

7.7.2.1.1 CGPW

The CGPW mode operator () is similar to the CPW mode operator except the pin specified for the test shall be from a clock gate (logic signal).

7.7.2.2 CGST

The CGST mode operator () is similar to the SETUP test mode operator except the reference edge shall be from a data pin and the signal shall be from a clock pin.

7.7.2.3 CPW

The CPW test mode operator specifies the edge identified as the signal shall be offset from the edge identified by the reference. The amount of the offset is at least as large as the bias value. The edge identified as the signal shall occur before the edge identified as the reference for late mode and positive bias values. The edge identified as the signal shall arrive after the edge identified as the reference for early mode and positive bias values. Both edges specified for this test shall be from the same pin and a clock.

7.7.2.4 CST

The CST mode operator specifies the edge identified as the signal shall be offset from the edge identified as the reference. The amount of the offset is at least as large as the bias value. The edge identified as the signal shall occur before the edge identified as the reference for late mode and positive bias values. The edge identified as the signal shall arrive after the edge identified as the reference for early mode and positive bias values. The edges specified for this test mode operator shall be from different clock pins.

7.7.2.5 DHT

The DHT mode operator specifies the separation of two data signals on the same cycle. The DHT specifies an edge of one data signal against the another edge of another data signal. The constraint is calculated by the CHECKS clause. As with other hold type test mode operators, this test helps to ensure the SIGNAL_EDGE comes after the REFERENCE_EDGE. The difference between this test mode operator and DST (see 7.7.2.8) is DST implies there is a cycle adjustment made to the reference signal before the data separation test is performed. There is no cycle adjust added to the reference in DHT.

7.7.2.6 DIFFERENTIAL_SKEW

DIFFERENTIAL_SKEW test specifies the maximum time difference between two signals on different pins. This test is used to compare the two arrival times of a differential signal. The pins shall be either both clock pins or data pins and must be of the opposite polarity.

7.7.2.7 DPW

The DPW test mode operator specifies the edge identified as the signal shall be offset from the edge identified by the reference. The amount of the offset is at least as large as the bias value. The edge identified as the signal shall occur before the edge identified as the reference for late mode and positive bias values. The edge identified as the signal shall arrive after the edge identified as the reference for early mode and positive bias values. Both edges specified for this test mode operator shall be from the same pin and not a clock.

7.7.2.8 DST

The DST mode operator is used to determine the offset between two data signals. This separation is established by the CHECKS subclause. The specified edge on the data pin specified as the SIGNAL_EDGE shall arrive before the edge specified on the data pin specified as the REFERENCE_EDGE for positive values of the offset.

7.7.2.9 HOLD

The HOLD test mode operator specifies (for positive bias values) the earliest edge identified as the signal shall arrive after the latest edge identified as the reference. The HOLD test mode operator shall be used in conjunction with early mode or after the both modes operator. The reference edge shall be from a clock pin.

7.7.2.10 NOCHANGE

The NOCHANGE test mode operator specifies the edge identified as the signal does not change during the duration of the setup period, the entire pulse width, and the hold period. The edge identified as the reference represents the earliest edge of the clock. The termination of the pulse width period is the opposite edge of the reference. The edge identified as the signal represents the earliest edge of the data. The nochange test mode operator requires the use of the combined early late mode operator. There are two bias values for this test.

The bias value preceding the combined early late mode operator determines the period preceding the reference edge. The bias value following the combined early late operator determines the period following the opposite edge of the reference. The reference edge shall be from a clock or control pin and the signal edges shall be from a logic pin.

7.7.2.11 RECOVERY

The RECOVERY test mode operator specifies the latest inactive edge identified as the SIGNAL_EDGE shall arrive before the earliest edge identified as the REFERENCE_EDGE. The bias value shall be positive. The SIGNAL_EDGE pin shall be from a control pin. The recovery test mode operator shall be used in conjunction with the late test mode operator or preceding the both modes operators. The REFERENCE_EDGE shall be from a clock pin.

7.7.2.12 REMOVAL

The REMOVAL test mode operator specifies the earliest inactive edge identified as the SIGNAL shall arrive after the latest edge identified as the reference. The bias value shall be positive. The SIGNAL pin shall be from a control pin. The removal test mode operator shall be used in conjunction with the late test mode operator or following the both modes operators. The reference edge shall be from a clock pin.

7.7.2.13 SETUP

The SETUP test mode operator specifies (for positive bias values) the latest edge identified as the SIGNAL shall arrive before the earliest edge identified as the reference. The SETUP test mode operator shall be used in conjunction with the late mode or preceding the both modes operators. The reference edge shall be from a clock pin.

7.7.2.14 SKEW

The SKEW test mode operator specifies the edge identified as the source shall occur within a window of time either before or after the edge identified as the reference. The bias sets the magnitude of the window. The bias value shall be positive. The use of this test mode operator with the early test mode operator indicates that the signal can occur after the reference up to the bias limit. The use of this test mode operator with the late mode operator indicates the signal may occur before the reference by an amount up to the bias value. Both the signal and reference shall be from clock pins, and only the early mode or late mode operators shall be used.

7.8 Expressions

An expression is a sequence of operators and operands that does the following:

- a) Specifies the computation of a value
- b) Designates an object or a function
- c) Generates side effects
- d) Performs a combination of these

The order of evaluation of subexpressions and the order in which side effects take place are unspecified, except as indicated by the syntax or when explicitly specified.

7.8.1 Array subscripting

The syntax for array subscripting is given in Syntax 7.20.

```

array_index ::= [ integer_comma_expression_list ]
              { [ integer_comma_expression_list ] }
integer_comma_expression_list ::= expression { , expression }

```

Syntax 7.20—array_index

A reference to an array element shall be made using the array name followed by a nonempty list of subscript expressions, which are separated by commas and surrounded by square brackets. Each subscript expression shall be of integer data type and evaluate to an integer value.

NOTE—The ISO C array reference `a[b][c][d]` is expressed as `a[b,c,d]` in DCL.

Referencing arrays of arrays is a list of array indexes where each index represents an access to a different array. The last level of nesting represents accessing the data and the previous levels are other arrays.

Example

array `a` is defined to be as follows: `double[b,c][d][e,f,g]:a`

array access `a[1,2][3][4,5]` accesses a double value where `a[1,2][3]` accesses a two-dimensional array containing double values.

7.8.2 Statement calls

This subclause describes statement calls in DCL.

7.8.3 General syntax

The syntax for statement calls is given in Syntax 7.21.

```

statement_call ::= statement_name ( [comma_expression_list] )
statement_reference ::= statement_call [ array_index ]
                      { [ array_index ] } { . field_reference }
field_reference ::= result_name [ ( [array_index]
                      { [ array_index ] } ) ] ( [comma_expression_list] ) ]

```

Syntax 7.21—statement_call

Arguments passed to a statement shall be read only within that statement, with the sole exception of VAR arrays (see 7.4.5.3.2.1), which shall also be writable by the statement.

7.8.4 Method statement calls

See Syntax 7.22 for method statement calls.

```
method_statement_call ::= method_statement_name ( )
method_statement_reference ::=
    method_statement_call [array_index] { . field_reference }
method_call ::= method_statement_reference
    | PATH_DATA:: method_statement_reference
    | CELL_DATA:: method_statement_reference
```

Syntax 7.22—method_statement_call

A method statement call made without specifying CELL_DATA:: scope shall default to a method statement call with the PATH_DATA:: scope.

7.8.5 Assign variable reference

See Syntax 7.23 for assign variable references.

```
assign_variable_reference ::= assign_statement_name
    [ . field_reference ]
```

Syntax 7.23—assign_variable_reference

The value referenced shall be the last calculation for the named ASSIGN statement.

7.8.6 Store variable reference

The syntax for store variable references is given in Syntax 7.24.

```
store_variable_reference ::= scoped_variable_reference
    | slot_variable_reference
scoped_variable_reference ::= store_reference
    | PATH_DATA:: store_reference
    | CELL_DATA:: store_reference
store_reference ::=
    recallable_statement_name [array_index | . field_reference]
recallable_statement_name ::= calc_statement_name
    | internal_statement_name
    | external_statement_name
    | expose_statement_name
    | tabledef_statement_name
slot_variable_reference ::= array_index store_reference
```

Syntax 7.24—store_variable_reference

The value referenced shall be that stored during model elaboration for the current PATH_DATA or CELL_DATA. If neither the PATH_DATA:: nor CELL_DATA:: scope operators are specified, then the use of PATH_DATA:: shall be assumed.

7.8.7 Mathematical expressions

This subclause details the mathematical expressions allowed in DCL and gives their syntax; see Syntax 7.25.

```

expression ::=
    constant
    | identifier
    | string_literal
    | scope_change
    | recall
    | launch
    | discrete_expression
    | statement_reference
    | assign_variable_reference
    | store_variable_reference
    | method_call
    | built-in_function_call
    | c_statement_reference
    | variable_reference
    | + expression
    | expression + expression
    | - expression
    | expression - expression
    | expression * expression
    | expression / expression
    | expression ** expression
    | expression `|` expression
    | expression `&` expression
    | expression `^` expression
    | expression `>` expression
    | expression `<` expression
    | `~` expression
    | ( expression )
variable_reference ::= expr_variable [ { [array_index]
    { . field_reference } } ]
expr_variable ::=
    assign_statement_name
    | passed_argument_name
    | result_field_name
    | local_field_name
    | predefined_variable_name

```

Syntax 7.25—expression

7.8.8 Mathematical operators

The mathematical operators are defined in Table 11.

Table 11—Mathematical operators

Symbol	Definition
*	Multiply: The left operand is multiplied by the right operand.
/	Divide: The left operand is divided by the right operand. The result of division of integer operands results in a double result.
%	Remainder: Integer division is performed between the left and right operand. The residue of that operation is called the “remainder.”
+	Addition: add the right operand to the left operand using twos complement arithmetic, unary plus: Treat the right operand as positive.
–	Subtraction: The right operand is subtracted from the left operand using twos complement arithmetic, unary minus: Treat the right operand as negative.
**	Exponentiation: The left operand is raised to the power of the right operand.
`	Bitwise OR of the left and right operands.
`&	Bitwise AND of the left and right operands.
`~	Bitwise complement: complement of the right operand using ones complement arithmetic.
`>	Shift right where the left operand is shifted to the right by the number of bits contain in the right operand.
`<	Shift left where the left operand is shifted to the left by the number of bits contain in the right operand.
`^	Bitwise exclusive OR of the left and right operand.

7.8.9 Discrete math expression

Discrete math represents sums or products of expressions. Discrete math expressions compute the sum of terms or the product of terms. The resulting type of the discrete expression is that of the *discrete_expression_body*.

The syntax for discrete math expressions is given in Syntax 7.26.

```

discrete_expression ::= discrete_declaration discrete_operator
    { discrete_expression_body }
discrete_declaration ::= [pin_list_discrete]
    | [integer_discrete]
pin_list_discrete ::=
    PINLIST : loop_variable_name = pinlist_expression
integer_discrete ::= discrete_bounds
    | discrete_bounds BY integer_expression
discrete_bounds ::= discrete_start discrete_end
discrete_start ::= INT : loop_variable_name = integer_expression
discrete_end ::= TO integer_expression=
discrete_operator ::= + | *
discrete_expression_body ::= expression

```

Syntax 7.26—discrete_expression

The *discrete_operator* represents the type of discrete operation being performed, as follows:

- * indicates a product operation on the loop_body expression.
- + indicates a summation operation on the loop_body expression.

Both the *integer_discrete* BY expression and the *discrete_end* expression shall be evaluated exactly once, after the *discrete_start* expression has been evaluated.

7.8.10 INT discrete

The integer discrete expression indicates the loop variable has type INTEGER and can be incremented or decremented only by an integer value.

7.8.11 PINLIST discrete

The pinlist discrete expression indicates the loop variable has type PINLIST and is stepped through the list of PINs in the PINLIST.

7.8.12 Logical expressions and operators

This subclause details the logical expressions and logical operators allowed in DCL.

The syntax for logical expressions is given in Syntax 7.27.

```

logical_expression ::=
    prefix_expression == prefix_expression
  | prefix_expression != prefix_expression
  | prefix_expression >= prefix_expression
  | prefix_expression <= prefix_expression
  | prefix_expression < prefix_expression
  | prefix_expression > prefix_expression
  | ! logical_expression
  | ( logical_expression )
  | logical_expression && logical_expression
  | logical_expression || logical_expression
prefix_expression ::= *string_expression | expression

```

Syntax 7.27—logical_expression

The DCL logical operators are defined in Table 12.

Table 12—Logical operators

Symbol	Definition
	Or
&&	And
!	Not
==	Equal
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

7.8.13 MODE expressions

7.8.13.1 Pin range

A pin range expression represents one or more input pins or one or more output pins on a circuit.

7.8.13.2 Pin range syntax

The syntax for a pin range is given in Syntax 7.28.

```
pin_range_list ::= pin_range { , pin_range }
pin_range ::= pin_name | ANYIN | ANYOUT | pin_name - pin_name
            | [ pin_string ] [ range_expression ] [ pin_string ]
            | [ pin_string ] < range_expression > [ pin_string ]
pin_name ::= name | scalar
pin_string ::= name | scalar
range_expression ::= scalar [ - scalar ]
```

Syntax 7.28—pin_range_list

NOTE—The use of angle brackets (<>) or square brackets ([]) affects the expansion of the Pin Range (see 7.8.13.4).

7.8.13.3 Pin range semantics

The pin range semantics are as follows:

- *pin_name*
Represents a single pin with the specified name.
- **ANYIN**
Represents an unordered sequence of all input and bidirectional pins not yet explicitly enumerated in a previous INPUT statement within the current MODELPROC or SUBMODEL sequence.
- **ANYOUT**
Represents an unordered sequence of all bidirectional and output pins not yet explicitly enumerated in a previous OUTPUT statement within the current MODELPROC or SUBMODEL sequence.
- *pin_name - pin_name*
Represents the set of pins determined by the following algorithm:

Let the lexically smaller pin name be PINnew
Let the lexically larger pin name be PINstop

Repeat {

Add PINnew to the set of resultant pin paths
Increment PINnew according to the Name Incrementation rule below

} until (PINnew is lexically greater than PINstop)

7.8.13.4 Constraints

The constraints are as follows:

- a) The specified pin names shall have the same number of characters. The specified pin names shall contain only characters within A to Z, a to z, and 0 to 9.
- b) Name incrementation rule:
 - 1) Names are incremented by lexically incrementing the right-most character of the name according to the character incrementation rules below. When a character being incremented in a name cycles back, then the character to its left (if any) is incremented.
- c) Character incrementation rules:
 - 1) A character in a name is lexically incremented through a specific range of characters. This range depends on the initial character.

- 2) If the initial character is in the range A to Z, then this character shall be incremented through the range of characters A to Z. When the character Z is incremented, it becomes A.
- 3) If the initial character is in the range a to z, then this character shall be incremented through the range of characters a to z. When the character z is incremented, it becomes a.
- 4) If the initial character is in the range 0 to 9, then this character shall be incremented through the range of characters 0 to 9. When the character 9 is incremented, it becomes 0.
- 5) Cycling back is defined as the incrementation step in which Z becomes A, or z becomes a, or 9 becomes 0.

Examples

A0-B7 produces names A0 A1...A7 A8 A9 B0 B1...B7

AB0-BC1 produces names AB0...AB9 AC0...AC9...AZ0...AZ9 BA0...BC0 BC1

- `[pin_string] [range_expression] [pin_string]`
Represents the set of pin names determined by the following algorithm.
- The first pin name is constructed from the concatenation of the preceding string (if any), an opening square bracket, the smaller integer, a closing square bracket, and the following string (if any). The integer is incremented by 1, and as long as the result is less than or equal to the larger integer, another pin name is generated in the same fashion.
- The number of digits used to express the lexically lower range value controls the minimum number of digits in the expansion of the following pin range:

Examples

A[0-9] produces names A[0] A[1]... A[9]

[0-99]B produces names [0]B [1]B... [99]B

A[3-00] produces names A[00] A[01] A[02] A[03]

c[1-3]addr produces names c[1]addr c[2]addr c[3]addr

- `[pin_string] <range_expression > [pin_string]`
Represents the set of pin names determined by the following algorithm.
- The first pin name is constructed from the concatenation of the preceding string (if any), the smaller integer, and the following string (if any). The integer is incremented by 1, and as long as the result is less than or equal to the larger integer, another pin name is generated in the same fashion.
- The number of digits used to express the lexically lower range value controls the minimum number of digits in the expansion of the following pin range:

Examples

A<0-9> produces names A0 A1... A9

X<0-99>B produces names X0B X1B... X99B

A<3-00> produces names A00 A01 A02 A03

c<1-3>addr produces names c1addr c2addr c3addr

7.8.14 Embedded C code expressions

The syntax for an embedded C code expression is given in Syntax 7.29.

```
c_statement_reference ::= $ name | c_reference_sequence
c_reference_sequence ::= $ name ( expression_list )
{ name ( expression_list ) }
```

Syntax 7.29—c_statement_reference

The C statement reference taken in its entirety shall represent a legal C construct resulting in a type compatible with the encapsulating DCL expression or assignment.

7.8.15 Computation order

This subclause details the computation order used in DCL.

7.8.15.1 Precedence for mathematical expressions

Mathematical expressions shall be evaluated according to the operator precedence shown in Table 13.

Table 13—Mathematical operator precedence (high to low)

Operators	Associativity
() [] .	Left to right
:^: :: :>:	Left to right
+ - (unary operators)	Right to left
**	Left to right
* / %	Left to right
+ -	Left to right
`>`<	Left to right
: .	Left to right
`&	Left to right
`^`	Left to right
	Left to right

7.8.15.2 Precedence for logical expressions

Logical expressions shall be evaluated according to the operator precedence shown in Table 14.

Table 14—Logical operator precedence (high to low)

Operators	Associativity
!	Right to left
< <= > >=	Left to right
== !=	Left to right
&&	Left to right
	Left to right

The && operator shall be evaluated left to right as there is a sequence point after the evaluation of the first operand. If the first operand evaluates to `false`, the second operand shall not be evaluated.

The || operator shall be evaluated left to right as there is a sequence point after the evaluation of the first operand. If the first operand evaluates to `true`, the second operand shall not be evaluated.

7.8.15.3 Passed parameters

Parameters passed in a statement or variable reference shall be evaluated in the order they appear in the reference, which is left to right.

7.8.15.4 WHEN clause

Logical expressions in WHEN clauses shall be evaluated in the order they appear in a statement until the controlling expression evaluates to `true`. If no logical expression evaluates to `true`, the OTHERWISE clause shall be evaluated.

7.8.15.4.1 Break

Break causes the inner most loop containing the associated when or otherwise to terminate.

7.8.15.4.2 Continue

Continue causes the innermost loop body associated with a when or otherwise to reexecute without doing any logical tests, and in the case of a FOR loop, it shall also not execute the loop modifier sequence.

7.8.15.5 REPEAT - UNTIL clause

A sequence of conditional result expressions shall be repeatedly evaluated until the controlling logical expression evaluates to `true`.

7.8.15.6 WHILE loops

When a WHILE loop is evaluated, if the logical expression evaluates `true`, the loop body is executed. This process repeats until the logic expression evaluates `false`, at which point the loop terminates.

7.8.15.7 FOR clause

The FOR clause consists of an initializer sequence, a conditional test, a modifier sequence, and a loop body. The initializer initializes the values prior to entering the conditional test sequence for the first time. The conditional test evaluates the logical expression contained within the conditional test sequence, and if that test evaluates `true`, it causes the loop body to be executed. Upon a successful pass through the loop body, the modifier sequence is executed and control is passed again to the conditional test sequence.

7.8.15.8 LOCK clause

LOCK clauses control serialization. Serialization is the process of preventing simultaneous access to either executable code or data. Simultaneous access can occur when there is more than one computing engine sharing either code or data, or a single computing engine running more than one thread.

The LOCK clause acquires for a context the access to a section of code or data. When the access is acquired, the locking context shall be permitted to either operate on the data or execute the code. Others not having access rights shall be denied access.

Lock requests are made before the first expression is valued and released when the LOCK clause is exited.

7.8.15.8.1 WRITE_LOCK

WRITE_LOCK requests exclusive access to either the data for updating or the code for execution. Once access has been granted to the context requesting the write lock, all other requests are blocked until the write_lock is released. The write_lock may be used to lock both code or data.

7.8.15.8.2 READ_LOCK

READ_LOCK requests access to the associated structure for the purpose of reading its values. Within a read_lock, modification of the data shall not be permitted. A read_lock does not block other contexts requesting a read_lock. A read_lock shall block other contexts requesting a write_lock or a wait. A context requesting a read_lock shall be blocked if a write_lock by another context has already been granted.

7.8.15.8.3 WAIT

WAIT blocks the requesting context if any other context holds a write_lock or read_lock on the associated structure. WAIT access once granted shall not block any other context from locking the data and continues to execute regardless of any other locks imposed after the access was granted.

NOTE—The wait clause is typically used to control access to the results of a launch.

7.8.15.8.4 BUSY

BUSY evaluates the expressions within its scope if access is denied to the structure identified by the lock clause.

7.8.15.8.5 RETRY

RETRY causes the flow of control to revert back to the start of the associated lock clause causing access to be requested again. If the retry has no associated logical expression, the retry is unconditionally performed. When the option logical_expression associated with the retry is present and that expression evaluates to true, the retry is performed otherwise control resumes after the end of the lock clause scope.

7.9 DCL mathematical statements

DCL statements are divided in the following categories: clauses, modifiers, prototypes, statement failure, interfacing statements, calculation statements, and methods.

7.9.1 Statement names

Statement names shall be unique except for the following:

- definitions and if present their matching prototype
- TABLEDEF with its matching TABLE statements
- MODELPROC and its matching MODEL statement
- Statement Clauses

This subclause defines the PASSED and RESULT clauses.

7.9.1.1 PASSED clause

The PASSED clause is an semicolon delimited list that declares the quantity, types and names of the required formal input parameters for a DCL statement. Its syntax is given in Syntax 7.30.

```
passed_clause ::= PASSED ( passed_argument_list )
passed_argument_list ::= passed_type_list { ; passed_type_list }
passed_type_list ::= passed_type : variable_name
passed_type ::= native_array_type | native_type | aggregate_type
variable_name ::= name { , name }
```

Syntax 7.30—passed_clause

7.9.1.2 RESULT clause

The RESULT clause is a semicolon delimited list that indicates the quantity, types, and names of variables returned from a DCL statement. There are two types of RESULT clauses: prototype and conditional.

7.9.1.3 Result prototypes

A prototype RESULT clause defines the types, names (except the unnamed RESULT variable), and order of variables returned by a statement whose definition has not been encountered. Its syntax is given in Syntax 7.31.

```
result_prototype ::= RESULT ( result_type_list )
result_type_list ::= result_default_type
    | result_named_list [ ; result_default_type ]
result_default_type ::= native_array_type | native_type
result_named_list ::= result_named_type { ; result_named_type }
result_named_type ::= aggregate_type_definition : variable_name_list
```

Syntax 7.31—result_prototype

7.9.1.3.1 Conditional logic

The conditional RESULT clause defines the types, names (except the unnamed RESULT variable), and logical and mathematical expressions that compute values for all variables returned from a DCL statement.

If the conditional RESULT clause uses the REPEAT RESULT()...UNTIL() syntax, the expressions in the RESULT clause shall be repeatedly evaluated until the UNTIL logical expression evaluates to true.

A conditional RESULT clause can itself contain multiple RESULT clauses. The set of variables returned by a DCL statement shall be the union of all variables mentioned in any of the RESULT clauses of that statement. The order of the variables returned shall be defined by the first appearance of each variable in any RESULT clause, scanning from the beginning to the end of the statement definition.

All result variables shall be assigned a value before returning from a statement.

The syntax for a conditional result is given in Syntax 7.32.


```
conditional_result ::=
    RESULT ( variable_sequence [ ; default_result_expression ] )
    | REPEAT code_body UNTIL ( logical_expression )
    | outer_when_sequence
variable_sequence ::= variable_expression_list
    | condition_logic
    | variable_sequence ; variable_expression_list
    | variable_sequence ; condition_logic
variable_expression_list ::=
    ( data_type : assignment ) | : statement_reference
    { ; ( data_type : assignment ) | : statement_reference } [ ; ]
assignment ::= variable_name = expression
default_result_expression ::= result_type : expression
conditional_logic ::=
    REPEAT code_body UNTIL ( logical_expression )
    | when_list [, OTHERWISE opt_result_definition]
    | FOR (
        (initializer_sequence) , (logical_expression) ,
        (modifier_sequence) ) code_body
    | lock_type code_body
    [, BUSY code_body
    [, RETRY [ ( logical_expression ) ] ]
when_list ::= when_logic {, when_logic}
when_logic ::= WHEN ( logical_expression ) code_body [BREAK]
outer_when_sequence ::= outer_when_list , OTHERWISE code_body
outer_when_list ::=
    WHEN ( logical_expression ) code_body
    {, WHEN ( logical_expression ) code_body }
initializer_sequence ::= [ variable_expression_list ]
modifier_sequence ::= [ variable_expression_list ]
code_body ::= { [ variable_sequence] }
data_type ::= aggregate_type_definition | native_type
lock_type ::= WRITE_LOCK | READ_LOCK | WAIT
```

Syntax 7.32—conditional_result

7.9.1.4 Default variable

A RESULT clause may have an unnamed, default variable. Such a variable shall do the following:

- Appear last in the RESULT clause
- Be separately typed; it may not appear as part of a list with similarly typed variables
- Be the textually last variable defined for the statement

The default variable shall be referenced using the statement_name syntax rather than the statement_name varname syntax.

Example

```
RESULT (INTEGER: 5)
```

7.9.1.5 LOCAL clause

The LOCAL clause shall define the types and names of variables that have a local scope within a DCL statement. Within a local clause, the values of all the local variables defined shall be initialized. The value of a local variable may be altered in a RESULT clause.

7.9.1.5.1 LOCAL conditional logic

The conditional logic within a LOCAL clause has the same semantic definitions as those for the result clause.

7.9.1.5.2 Local variables

Local variables are variables created within the scope of a statement, which only persist for the duration of that statement. Local variables are only visible to the statement that defines them (Syntax 7.33).

```
local_clause ::= LOCAL ( local_sequence )
local_sequence ::= [ named_local_expression_list
    | conditional_logic { ; named_local_expression_list
    | conditional_logic } ]
named_local_expression_list ::=
    ( ( aggregate_type : assignment ) | ( : statement_reference ) ) { ; ( (
    aggregate_type : assignment ) | ( : statement_reference ) ) } [ ; ]
```

Syntax 7.33—local_clause

7.9.1.5.3 Local variable definition

Local variables are defined by declaring a type within a LOCAL clause and associating it with an identifier. Variables defined in a LOCAL clause have a lifetime limited to the duration of this statement and cannot be redefined in a RESULT clause. Local variables may have their values changed within the associated result or default clauses.

7.9.1.5.4 Local clause placement

Local clauses can be placed as follows (Syntax 7.34):

- The first clause within a DEFAULT clause
- Immediately preceding the first RESULT clause, outer WHEN, or REPEAT within a CALC, an EXPOSE, or an ASSIGN statement
- Immediately preceding the first occurrence of an EARLY or LATE clause within a DELAY or SLEW statement
- Immediately preceding the BIAS clause of a check statement

```
default_clause ::= [ DEFAULT
    ( [ local_clause ; ] default_sequence ) ]
calculation_body ::= [ passed_clause ] [ local_clause ]
    conditional_result
delay_statement ::= [ EXPORT ] DELAY ( name ) delay_slew_postfix_modifier
    :
    [ passed_clause ] [ local_clause ]
    conditional_time ;
check_statement ::= [ EXPORT ] CHECK ( name ) std_postfix_modifier :
    [passed_clause] [local_clause] conditional_bias ;
```

Syntax 7.34—default_clause

7.9.1.6 DEFAULT clause

The syntax for the **DEFAULT** clause is given in Syntax 7.35.

```
default_clause ::= [ DEFAULT ( [local_clause] default_sequence ) ]
default_sequence ::= named_expression_list
    [ ; default_expression ]
named_expression_list ::= variable_name = expression
    { ; variable_name = expression }
default_expression ::= expression
```

Syntax 7.35—default_clause (result variable)

The **DEFAULT** clause specifies the values to be returned if a statement fails to complete successfully. If a **DEFAULT** clause is used and the statement has a **RESULT** clause, the **DEFAULT** clause shall specify and assign a value to every variable in that **RESULT** clause. Named variables may appear in a different order than listed in the **RESULT** clause, except the unnamed (default) variable, if any, shall appear last in the **DEFAULT** clause.

7.9.2 Modifiers

This subclause describes how modifiers are used in DCL.

7.9.2.1 Statement purity

A statement is pure if it always returns the same result value(s) given the same passed parameter(s). Otherwise, a statement is impure. Statements can be explicitly made pure or impure by specifying the **PURE** or **IMPURE** statement modifier, respectively.

If no purity modifiers are specified for a statement, the statement's purity shall be determined as follows:

- **ASSIGN**, **EXPOSE**, **LOAD_TABLE**, **UNLOAD_TABLE**, **ADD_ROW**, and **DELETE_ROW** statements shall be considered to be **IMPURE**.
- Statements executed by means of a statement pointer shall be considered to be impure.
- All other statements shall be considered to be **PURE** unless those statements reference an **IMPURE** statement without the **PURE** operator or are declared with a **VAR** array parameter.
- Variables created with the **NEW** operator shall be considered to be impure.

NOTE 1—The behavior and results of asserting an impure statement as pure are undefined.

NOTE 2—A pure statement is a hint to the DCL compiler that the results of a statement can be saved and reused as long as the input parameters are the same.

7.9.2.2 Statement consistency

Consistency of a DCL statement is only meaningful within a MODELPROC statement. A statement is said to be consistent if, given the same passed parameter(s), it shall return the same result value(s) for all instances of the same cell modeled by a given MODELPROC. Otherwise, a statement is inconsistent. Statements can be explicitly made consistent or inconsistent by specifying the CONSISTENT or INCONSISTENT statement modifier, respectively.

If no consistency modifiers are specified for a statement, the statement's consistency shall be determined as follows.

The following statements are considered to be INCONSISTENT:

- TABLEDEF statements with the DYNAMIC option
- LOAD_TABLE statements
- UNLOAD_TABLE statements
- ADD_ROW statements
- DELETE_ROW statements
- EXTERNAL statements
- ASSIGN statements
- INTERNAL statements
- Statements with the IMPORT prototype modifier

All other statements are considered to be CONSISTENT unless the statements:

- Are impure statements
- Reference an inconsistent statement
- Reference embedded C code
- Executed by means of a statement pointer

It is an error to mark LOAD_TABLE, UNLOAD_TABLE, ADD_ROW, and DELETE_ROW statements as CONSISTENT. They are always INCONSISTENT. It is an error to use CONSISTENT with IMPURE. IMPURECONSISTENT is self-contradictory.

IMPORTPURE without CONSISTENT shall be understood as IMPORTPUREINCONSISTENT. IMPORTCONSISTENT without PURE shall be understood as IMPORTPURECONSISTENT. FORWARD PURE without CONSISTENT shall be understood as FORWARDPURECONSISTENT. FORWARD CONSISTENT without PURE shall be understood as FORWARDPURECONSISTENT.

NOTE 1—The behavior and results of asserting an inconsistent statement as consistent are undefined.

NOTE 2—A consistent statement is a hint to the DCL compiler that the results of a statement can be saved and reused for all cells modeled by this MODELPROC statement.

7.9.2.3 Locking modifiers

Locking modifiers control the serialization of the execution of the associated statement. The locking modifiers serialize the access to the identified statement's execution, and it is not associated with any specific instance of data. Locking modifiers can serialize at the granularity of either the plane or the space. Serializing at the granularity of the plane allows only one plane within a space to be executing this

statement at any any given instance in time. Serializing at the space level allows only one plane from one space to execute within this statement at any given instance in time.

7.9.2.3.1 AUTOLOCK

AUTOLOCK causes the serialization to begin before the first expression in the statement is evaluated and control is released after the last expression in the statement has been evaluated.

7.9.2.3.2 LOCK

LOCK causes the expressions contained within the scope of the LOCK clause to be serialized. Serialization begins before the first expression contained within the lock clause is evaluated, and serialization is released when the lock clause is exited.

7.9.2.4 Context modifiers

Context modifiers control the accessibility of data. These modifiers only apply to either assign statements or tabledef statements. These modifiers indicate the degree of data reuse with these statements. By default, that is, without any context modification, the data contained within these statements are unique per context, and each context contains its own separate versions of the data. When a common plane is used, the data contained within the statement have the same values for any plane on a given space. When a common space is used, the data values are identical for all planes on all spaces.

7.9.2.4.1 COMMON

COMMON modifiers are used for storage statements such as ASSIGN and TABLEDEF. COMMON identifies whether the storage is unique to a context or is accessible to more than one context. COMMON has an argument that can have either the value of SPACE or PLANE. Plane indicates that there shall be a single instance of the data for each space, and access to the stored values is serialized per plane. SPACE indicates there is only a single instance of the data for all spaces and all planes. All accesses to the data are serialized and no concurrent access is allowed.

Data types associated with statements that have the COMMON attribute are assumed to be SYNC.

7.9.2.5 Access modifiers

Access modifiers control the serialization to structures.

7.9.2.5.1 SHARED

Types marked with the shared option shall be shared types.

7.9.2.5.2 SYNC

Types marked with the sync option shall be sync types.

7.9.3 Prototypes

This subclause describes how prototypes are used in DCL. Prototypes represent the structure of a statement that is not currently defined. The prototype statement gives the structure of the statement, where this statement might be found when the module is executed, and whether or not the presence of the definition is required.

7.9.3.1 Prototype modifiers

The syntax for prototype modifiers is given in Syntax 7.36. The optional modifier shall not be used with the assign statement.

```

prototype_modifier ::= IMPORT | FORWARD
std_postfix_modifier ::= [optimize_ctl_postfix_modifier]
                        [model_ctl_postfix_modifier]
                        [link_control_postfix_modifier]
                        [lock_postfix_modifier]
                        [common_postfix_modifier] [( SYNC | SHARED )]
optimize_ctl_postfix_modifier ::= PURE | IMPURE
model_ctl_postfix_modifier ::= CONSISTENT | INCONSISTENT
tabledef_ctl_modifier ::= DYNAMIC | OVERRIDE
table_message_modifier ::= SUPPRESS
link_control_postfix_modifier ::= OPTIONAL
lock_postfix_modifier ::= ( autolock_modifier | lock_modifier )
autolock_modifier ::= AUTOLOCK ( ( SPACE | PLANE ) )
lock_modifier ::= LOCK ( ( SPACE | PLANE ) )
common_postfix_modifier ::= COMMON _ ( ( SPACE | PLANE ) )

```

Syntax 7.36—prototype_modifier

The syntax for common prototype modifiers is given in Syntax 7.37.

```

common_prototype ::= prototype_modifier ASSIGN | CALC
                  | EXPOSE | INTERNAL | IMPORT | EXTERNAL
common_statement_prototype ::= common_prototype ( name )
std_postfix_modifier : [ passed_clause ] result_prototype ;

```

Syntax 7.37—common_prototype

7.9.3.2 TABLEDEF prototype

The syntax for TABLEDEF prototype modifiers is given in Syntax 7.38.

```

tabledef_prototype ::= tabledef_prototype_preamble
                      tabledef_prototype_clauses ;
tabledef_prototype_clauses ::=
    [passed_clause] qualifiers_clause data_clause [key_clause]
    | [passed_clause] qualifiers_clause [key_clause] data_clause
    | [passed_clause] [key_clause] qualifiers_clause data_clause
tabledef_prototype_preamble ::= prototype_modifier
    TABLEDEF ( name ) std_postfix_modifier [tabledef_modifiers] :
tabledef_modifiers ::= [tabledef_ctl_modifier] [table_message_modifier] [
    descriptor_clause]
descriptor_clause ::= DESCRIPTOR ( [ aggregate_modifiers ] name )
aggregate_modifiers ::= SYNC | SHARED
qualifiers_clause ::= QUALIFIERS ( qualifier_list )
qualifier_list ::= qualifier_name {, qualifier_name}
qualifier_name ::= assign_variable_reference | store_variable_reference |
    passed_variable_name | predefined_variable_name
data_clause ::= DATA ( table_data_sequence )
table_data_sequence ::= default_data_sequence | named_data_sequence [ ;
    default_data_sequence ]
default_data_sequence ::= table_type

```

```
table_type ::= base_table_type [ dimension_list ]
base_table_type ::= mathematical_type | STRING
named_data_sequence ::= table_type : variable_name_list
    { ; table_type : variable_name_list }
key_clause ::= KEY ( scalar )
```

Syntax 7.38—tabledef_prototype

It shall be considered an error to mark a DYNAMIC table as CONSISTENT unless the DYNAMIC table, once created, never changes. In this case, the DYNAMIC table can be marked PURECONSISTENT.

7.9.3.3 LOAD_TABLE, UNLOAD_TABLE and WRITE_TABLE prototypes

The syntax for LOAD_TABLE and UNLOAD_TABLE prototype modifiers is given in Syntax 7.39.

```
load_table_prototype ::= IMPORT load_unload_type ( name )
    std_postfix_modifier : [passed_clause]
    TABLEDEF ( tabledef_statement_name ) ;
load_unload_type ::= LOAD_TABLE | UNLOAD_TABLE | WRITE_TABLE
```

Syntax 7.39—load_table_prototype

7.9.3.4 ADD_ROW and DELETE_ROW prototypes

The syntax for the add and delete row prototype modifiers is given in Syntax 7.40.

```
add_row_prototype ::= prototype_modifier add_delete_type
    ( name ) std_postfix_modifier : TABLEDEF ( tabledef_statement_name ) ;
add_delete_type ::= ADD_ROW | DELETE_ROW
```

Syntax 7.40—add_row_prototype

7.9.3.5 DELAY and SLEW prototypes

The syntax for the DELAY and SLEW prototype modifiers is given in Syntax 7.41.

```
delay_prototype ::= prototype_modifier delay_slew_type ( name
    ) std_postfix_modifier : [passed_clause] ;
delay_slew_type ::= DELAY | SLEW
```

Syntax 7.41—delay_prototype

7.9.3.6 CHECK prototype

The syntax for the CHECK prototype modifier is given in Syntax 7.42.

```
check_prototype ::= prototype_modifier CHECK ( name )
    std_postfix_modifier : [passed_clause] ;
```

Syntax 7.42—check_prototype

7.9.3.7 SUBMODEL prototype

The syntax for the SUBMODEL prototype modifiers is given in Syntax 7.43.

```
submodel_prototype ::= prototype_modifier SUBMODEL ( name ) :  
    [passed_clause] [result_prototype] END ;
```

Syntax 7.43—submodel_prototype

7.9.4 Statement failure

DCL statements can fail (i.e., not successfully complete the desired calculation). If a statement is about to fail with error severity less than 3 and has an associated DEFAULT clause, then that clause shall be evaluated. If the DEFAULT clause evaluation succeeds, its values shall be returned by the statement. If the statement is about to fail and either it has no DEFAULT clause or it failed during the evaluation of the DEFAULT clause, then a nonzero (“error”) code shall be returned.

The error code returned to the application by a nested set of failing DCL statements shall be the code associated with the most deeply nested failing statement.

NOTE 1—See 10.9.3.

NOTE 2—If in DCL statement S1 there is a reference to DCL statement S2 and the reference to S2 fails, if S1 has a DEFAULT clause, then that clause is evaluated. Otherwise, statement S1 fails.

7.9.5 Type definition statements

DCL uses statements to define structure types. All DCL statements that may be referenced define a type that matches that statement.

7.9.5.1 TYPEDEF

A TYPEDEF statement is used to define structures. The TYPEDEF statement shall not be referenced in an expression as shown in Syntax 7.44.

```
typedef ::= TYPEDEF ( name ) [typedef_options] :  
    result_prototype ;  
typedef_options ::= SYNC | SHARED  
result_type_list ::= result_default_type | type_named_list  
    [ ; type_default_type ]  
type_named_list ::= type_named_type { ; type_named_type }  
typedef_named_type ::= aggregate_type_definition [TRANSIENT] :  
    variable_name_list
```

Syntax 7.44—typedef

7.9.5.1.1 TYPEDEF RESULT clause

The result clause of the typedef statement is a result clause prototype with the following differences:

- TRANSIENT attribute shall not be used on structure types that have the field var attribute.
- Structures defined may be used as member of the result clause of the method statement or the external statement. All other statements having a result clause or a data clause shall not return structures that are or contain transient structures as any level of nesting.
- Structure definitions that contain the transient attribute may be used at types in local and passed clauses.

This can help facilitate communications with other programming languages.

7.9.6 Interfacing statements

Interfacing statements specify statement names, arguments, and return values from the perspective of either the application or the DPCM. The statement names defined by this standard are enumerated in 7.9.6.1, 7.9.6.2, and 7.9.6.3.

7.9.6.1 EXPOSE statement

The syntax for the EXPOSE statement is given in Syntax 7.45.

```
expose_statement ::= [EXPORT] EXPOSE ( name ) expose_modifiers :  
    [passed_clause] [local_clause] [conditional_result] ;  
expose_modifiers ::= {method_post_fix_modifier  
    | FIRST | LEADING | TRAILING | LAST}
```

Syntax 7.45—expose_statement

The EXPOSE statement makes the exposed name visible to the application.

The default linkage for EXPOSE shall be EXPORT; it cannot be made static. This means an EXPOSE statement can be as follows:

- Imported using the IMPORT reserved word
- Exported (by default or by the EXPORT reserved word explicitly)
- Is IMPURE INCONSISTENT by default

There can be more than one exported EXPOSE statement of the same name within a system of subrules within a TECH_FAMILY. In this situation, all such statements shall have the same argument signatures in their PASSED and RESULT clauses. More than one statement may be executed when the exposed statement is referenced.

By default, each EXPOSE statement is appended to the expose chain in the order the rules were loaded. In contrast, when more than one MODELPROC of the same name is read, the last MODELPROC loaded is the MODELPROC used. A conflict can exist when a calculation based on the last MODELPROC loaded requires information from an EXPOSE call obtained from the expose chain sequence. If the expose chain is not altered, then information for the revised MODELPROC could be mixed with the data associated with the first MODELPROC loaded. To prevent this from happening, the expose chain placement control modifiers FIRST, LAST, LEADING, and TRAILING control the order in which the statements are placed on the expose chain for a given statement. Each EXPOSE statement shall contain zero or one placement control modifiers. The EXPOSE statement syntax (see Syntax 7.45) shows the syntax for these modifiers.

7.9.6.1.1 FIRST modifier

The FIRST modifier causes the EXPOSE statement to be placed first on the chain of EXPOSE statements having the same name and contained within the same tech_family. There shall only be one EXPOSE statement within a tech_family with the FIRST placement control modifier.

7.9.6.1.2 LEADING modifier

The LEADING modifier causes the EXPOSE statement to be placed either first or second on the list of EXPOSE statements having the same name and contained within the same tech_family. The EXPOSE statement with the leading modifier is placed first on the chain of EXPOSE statements only if no other EXPOSE statement possessing the FIRST modifier already on the chain already exists. Otherwise, the

EXPOSE statement with the LEADING modifier is placed second on the chain, just after the statement containing the FIRST placement control modifier.

7.9.6.1.3 TRAILING modifier

The TRAILING modifier causes the EXPOSE statement to be placed either last or next to last on the chain of EXPOSE statements having the same name and contained within the same tech_family. The EXPOSE is placed last on the chain if no other EXPOSE statement possessing the last modifier already exists on the chain. Otherwise, the EXPOSE statement is placed next to last on the chain, just before the statement with the LAST placement control modifier.

7.9.6.1.4 LAST modifier

The LAST modifier causes the EXPOSE statement to be placed last on the chain of EXPOSE statements having the same name and contained within the same tech_family. There shall only be one EXPOSE statement within a tech_family with the LAST placement control modifier.

7.9.6.2 EXTERNAL statement

The syntax for the EXTERNAL statement is given in Syntax 7.46.

```
external_statement ::= [EXPORT] EXTERNAL ( name ) std_postfix_modifier :
    [passed_clause] [result_prototype] [default_clause] [proxy_clause] ;
proxy_clause ::= PROXY ( result_type_statement members )
```

Syntax 7.46—external_statement

The EXTERNAL statement makes the referenced name visible to the DPCM.

All EXTERNAL statements for a given statement name shall have the same argument signature for their PASSED and RESULT clauses. This requirement regarding argument signatures applies for all TECH_FAMILY(ies).

7.9.6.2.1 PROXY

The PROXY clause identifies a statement that shall be used in place of the current statement should the application or the run time fail to provide one. The statement identified in the proxy clause shall have the same passed and result type sequences as the containing external statement.

NOTE—Coding optional causes the run time to supply a default function if the application fails to provide one. The default function supplied by the run time prevents the proxy from taking effect. When using a proxy clause, do not use the optional modifier.

7.9.6.3 INTERNAL statement

The syntax for the INTERNAL statement is given in Syntax 7.47.

```
internal_statement ::= [EXPORT] INTERNAL ( name ) method_postfix_modifier
    : [passed_clause] [result_prototype] %{ C_CODE }%;
```

Syntax 7.47—internal_statement

The INTERNAL statement declares a DCL-language interface to code written in the C language.

7.9.7 DCL to C communication

The passed argument list to C code shall consist of the following:

- The Standard Structure pointer as the first argument and named *std_struct*.
- If the statement contains a result prototype, the next argument shall be the address of the area where the results are to be placed. The result area shall be a C struct containing elements of the corresponding type and order for each variable defined in the result prototype. The name of the return area pointer shall be *dcm_rtn*.
- If there is no result prototype, the *dcm_rtn* parameter shall not be present.
- Any additional passed parameters, if any, shall match in quantity, type, and order as expressed in the passed clause.

The return value of the statement shall be the C type *int* and shall represent the return code. Successful completion shall return with value 0. Unsuccessful completion shall return with an error code.

The internal statement maintains the reference counts for aggregate types when the function is called and again when it returns. To allow the proper manipulation of the reference counts, the return operator shall not be used in the C function body. Instead, use well-structured programming practices or create a label at the end of the C function body and use a C *goto* to jump to the label.

7.9.7.1 Built-in label

Within internal statements, the C function body may use the *goto* operator in C to jump to the predefined label *dcmStmtExit*. The label *dcmStmtExit* marks the end of the C function body. Within the C function body, a *goto dcmStmtExit* shall jump to the end of the C function body.

7.9.7.2 dcm_rc

Internal statements have the predefined return variable of type *int*. This variable returns the error code to the caller. This variable is initialized to the value of zero before the C function body starts. Error returned C function body shall be done by assigning the error code value to the predefined variable *dcm_rc* before jumping to *dcmStmtExit* or reaching the end of the C function body.

7.9.8 Constant statement

A constant statement allows the definition of simple constants. These constants are embedded directly in the text of the executable module.

For example, a constant statement can be used to embed a string in the executable module that other module examination programs can search for (Syntax 7.48).

```
constant_statement ::= CONSTANT ( name ) : RESULT (
    simple_assignment_list ) ;
simple_assignment_list ::= simple_assignment
    { ; simple_assignment }
    | simple_assignment { ; simple_assignment } ; default_constant |
    default_assignment
simple_assignment ::= (DOUBLE | FLOAT) : name = number_constant
    | STRING : name = string_constant
default_constant ::= (DOUBLE | FLOAT) : number_constant
    | STRING : string_constant
```

Syntax 7.48—constant_statement

7.9.9 Calculation statements

The syntax for the calculation is given in Syntax 7.49.

```
calculation_body ::= [passed_clause] [local_clause] [conditional_result]
```

Syntax 7.49—calculation_body

7.9.9.1 CALC statement

The syntax for the CALC statement is given in Syntax 7.50.

```
calc_statement ::= [EXPORT] CALC ( name ) method_postfix_modifier :  
    calculation_body ;  
method_postfix_modifier ::= std_postfix_modifier  
    [DEFAULT ( method_name_list )]  
method_name_list ::= method_statement_name { , method_statement_name }
```

Syntax 7.50—calc_statement

The CALC statement is DCL's primary numerical calculation statement. It defines a statement that can be called from other DCL statements.

7.9.9.2 ASSIGN statement

The syntax for the ASSIGN statement is given in Syntax 7.51.

```
assign_statement ::= [EXPORT] ASSIGN ( name ) method_postfix_modifier :  
    [passed_clause] [local_clause] [conditional_result] ;
```

Syntax 7.51—assign_statement

An ASSIGN statement, which is similar to a CALC statement, shall evaluate and return values specified in a RESULT clause. The ASSIGN statement, unlike the CALC statement, shall allocate storage for variables in the RESULT clause and copy the evaluation results into that storage before returning.

An ASSIGN variable has the same scope (visibility) as the defining ASSIGN statement. An ASSIGN statement shall not reference its own ASSIGN variables.

NOTE—Using the ASSIGN statement can result in side effects if the calculation environment is recursive or reentrant.

7.9.9.3 DELAY statement

The DELAY statement (Syntax 7.52) calculates segment delays for a path.

```
delay_statement ::= [EXPORT] DELAY ( name ) delay_slew_postfix_modifier :  
    [passed_clause] conditional_time ;  
delay_slew_postfix_modifier ::= std_postfix_modifier [DEFAULT]  
conditional_time ::= early_late_sequence | delay_slew_when_sequence ,  
    OTHERWISE early_late_sequence  
early_late_sequence ::=  
    EARLY ( float_expression ) LATE ( float_expression )  
    | LATE ( float_expression ) EARLY ( float_expression )  
delay_slew_when_sequence ::= WHEN ( logical_expression )  
    early_late_sequence { , WHEN ( logical_expression )  
    early_late_sequence }
```

Syntax 7.52—delay_statement

7.9.9.3.1 EARLY and LATE clauses and result variables

The DELAY statement does not have an explicit RESULT clause; rather, the statement has two required clauses, EARLY and LATE, which can appear in any order. The EARLY and LATE clauses define their respective result variables EARLY and LATE. The DELAY statement returns these values as though it had the RESULT clause

```
RESULT (FLOAT: EARLY, LATE)
```

7.9.9.3.2 DEFAULT modifier for DELAY statements

The DEFAULT modifier identifies the DELAY statement to be used in situations where no other DELAY statement has been specified.

The DEFAULT modifier has the following restrictions:

- Only one DELAY statement may contain the DEFAULT modifier within the scope of a TECH_FAMILY.
- No passed parameters.
- The DEFAULTDELAY statement shall not have passed parameters. Because the DPCM does not model the segment, it shall not recognize the proper parameters to pass the statement.
- The DEFAULTDELAY statement shall not reference STORE variables.

7.9.9.4 SLEW statement

The SLEW statement (Syntax 7.53) calculates transition times for a path.

```
slew_statement ::= [EXPORT] SLEW ( name ) delay_slew_postfix_modifier :  
[passed_clause] conditional_time ;
```

Syntax 7.53—slew_statement

7.9.9.4.1 EARLY and LATE clauses and result fields

The SLEW statement does not have an explicit RESULT clause; rather, the statement has two required clauses, EARLY and LATE, which can appear in any order. The EARLY and LATE clauses define their respective result variables EARLY and LATE. The SLEW statement returns these values as though it had the RESULT clause

```
RESULT (FLOAT: EARLY, LATE)
```

7.9.9.4.2 DEFAULT modifier for slew statements

The DEFAULT modifier identifies the SLEW statement to be used in situations in which no other SLEW statement has been specified.

The DEFAULT modifier has the following restrictions:

- Only one SLEW statement may contain the DEFAULT modifier within the scope of a TECH_FAMILY.
- No passed parameters.

- The DEFAULTSLEW statement shall not have passed parameters. Because the DPCM does not model the segment, it shall not recognize the proper parameters to pass the statement.
- The DEFAULTSLEW statement shall not reference STORE variables.

7.9.9.5 CHECK

The syntax for the CHECK statement is given in Syntax 7.54.

```
check_statement ::= [EXPORT] CHECK ( name ) std_postfix_modifier :
    [passed_clause] conditional_bias ;
conditional_bias ::= BIAS ( expression ) check_when_sequence , OTHERWISE
    BIAS ( expression )
check_when_sequence ::=
    WHEN ( logical_expression ) BIAS ( expression )
    { , WHEN ( logical_expression ) BIAS ( expression ) }
```

Syntax 7.54—check_statement

The CHECK statement computes the allowable difference in arrival times based on the comparison between a signal's (data) arrival time and a reference (clock), which shall be present for a circuit to function.

The CHECK statement does not have an explicit RESULT clause; rather, the statement has a required clause, BIAS. The CHECK statement returns the value of this BIAS clause as if it had the RESULT clause.

RESULT (float: BIAS)

The BIAS clause computes the allowable difference in arrival times between a signal and reference.

7.9.10 METHOD statement

The syntax for the METHOD statement is given in Syntax 7.55.

```
method_statement ::= METHOD ( name ) std_postfix_modifier :
    result_prototype ;
```

Syntax 7.55—check_statement

DCL supports access to multiple statements, which are referenced through a common name (the METHOD *name*) and differentiated by the PATH_DATA or CELL_DATA scoping operator associated with particular PINs, PATHs, or CELLS. The METHOD statement shall declare a common result prototype for all associated statements. These associated statements are called “action statements.” An action statement can be a CALC, ASSIGN, EXPOSE, INTERNAL, or EXTERNAL statement. Within a MODELPROC or SUBMODEL, the METHODS clause shall associate a particular action statement with a CELL, PIN, or PATH.

If no specific action statement is associated with a method, the DEFAULT action statement (if defined) is executed. A DEFAULT action statement is defined via the DEFAULT modifier on a CALC, ASSIGN, EXPOSE, INTERNAL, or EXTERNAL statement. It shall be an error to reference a METHOD statement for which there is no associated action statement. The scope of a METHOD statement name shall be global [i.e., across all TECH_FAMILY(ies)].

7.9.10.1 Default action statement

A DEFAULT action statement may be defined for any METHOD statement and is subject to the following restrictions:

- A DEFAULT action statement shall not have any passed parameters.
- There shall only be zero or one default action statements registered to each method within a TECH_FAMILY.

7.9.10.2 Selection of action statement

At all times during the execution of DCL statements, a context shall be defined by the contents of the Standard Structure.

When a METHOD is referenced, the following rules shall specify which action statement is executed:

- The associated CELL, PIN, or PATH is first determined.
- If the METHOD reference uses the scope operator CELL_DATA::, then the action statement is assumed to be associated with the cell identified by the cellData field of the Standard Structure. If the METHOD reference uses the scope operator PATH_DATA:: (or if no scope operator was used), then the action statement is assumed to be associated with the PIN or PATH identified by the pathData field of the Standard Structure.
- If an action statement was associated with the identified CELL, PIN, or PATH, that action statement shall be executed.
- If no such action statement was found, or the supplied pathData or cellData handle is zero, and a DEFAULT action statement was declared, then that DEFAULT statement shall be executed.
- If no action statement was found and no default action statement was declared, then an error shall be propagated back to the calling statement.

7.10 Predefined types

To enable the language to manipulate objects created by some built-in functions or basic constructs of the language, these types shall be defined as part of the language.

7.10.1 ACTIVITY_HISTORY_TYPE

The ACTIVITY_HISTORY_TYPE contains a linked list of HISTORY_TYPE objects shown in Table 15.

Table 15—Type definition for ACTIVE_HISTORY_TYPE

```
typedef(ACTIVE_HISTORY_TYPE):
    result(ACTIVE_HISTORY_TYPE: next;
          HISTORY_TYPE: refobj;
          void: reserved;
          int: activityCode; );
```

7.10.1.1 next

next creates a linked list of activities that have occurred on the referenced object (refobj).

7.10.1.2 refobj

refobj is the structure representing the history associated with this object.

7.10.1.3 reserved

Reserved is undefined and shall not be referenced. It is reserved for run-time library support only.

7.10.1.4 activityCode

activityCode indicates what operation was performed on this object, as shown in Table 16.

Table 16—Permitted activityCode values

Value	Meaning
0	Indicates that refobj represents the history for the root rule
1	Indicates that the refobj represents the history for the main program rule
2	Indicates that the refobj represents the history for a C program module
3	Indicates that the refobj represents the history for a subrule that was loaded
4	Indicates that the refobj represents the history for a table that was loaded
5	Indicates that the refobj represents the history for a table that was loaded but the data were not because they were deferred
6	Indicates that the refobj represents the history for a table that was loaded with the replace option
7	Indicates that the refobj represents the history for a table that was loaded with the override option
8	Indicates that the refobj represents the history for a table's data that were loaded because it was deferred, but the table was since searched forcing the table data to be placed into the table's memory image
9	Indicates a dynamic table was appended
10	Reserved for future use
11	Indicates that the refobj represents a load operation by this subrule

7.10.2 HISTORY_TYPE

HISTORY_TYPE holds the information on individual history events such as table loading information (Table 17).

Table 17—Type definition for HISTORY_TYPE

```
typedef(HISTORY_TYPE) :
    result(string[*]: info;
        ACTIVITY_HISTORY_TYPE: *activity;
        void: reserved;
        int: kind; );
```

7.10.2.1 info

The info array contains a list of pertinent information about the table or subrule loaded. The sequence of messages is determined by the kind of history this instance is. When kind applies to a rule the sequence of messages are shown in Table 18.

Table 18—Rule history info message types

Index value	Type of string message
0	Name
1	Load path
2	Tech_family name
3	Time stamp
4	Control_parm
5	Rule source name

When kind represents a table, the permitted values are shown in Table 19.

Table 19—Table History inform message types

Index value	Type of string message
0	Name
1	Load path
2	Tech_family name
3	Time stamp

7.10.2.2 activity

The activity structure contains what other actions were spawned as a result of this action.

7.10.2.3 kind

The kind field indicates whether the history applies to a rule or a table as shown in Table 20.

Table 20—Permitted kind values

Value	Meaning
0	The history applies to a subrule
1	The history applies to a table

7.10.3 LOAD_HISTORY_TYPE

LOAD_HISTORY_TYPE contains the history for both rule and tables loaded as shown in Table 21.

Table 21—LOAD_HISTORY TYPE

```
typedef (LOAD_HISTORY_TYPE):
    result(HISTORY_TYPE[*]: ruleHistory, tableHistory;
    void: reserved1, reserved2, reserved3, reserved4;
    string: techName; );
```

7.10.3.1 ruleHistory

ruleHistory is an array containing the history records for all the subrules loaded.

7.10.3.2 reserved1, reserved2, reserved3, reserved4

The reserved fields are undefined and shall not be modified.

7.10.3.3 techName

techName is the tech_family name of the root rule.

7.10.4 CELL_LIST_TYPE

CELL_LIST_TYPE (Table 22) contains the cell name, cell qualifier name, and model domain name for a cell as it appears in the model statement. This structure also contains two undefined fields rsvd1 and rsvd2 that shall not be modified.

Table 22—CELL_LIST_TYPE

```
typedef (CELL_LIST_TYPE):  
    result(string: cellName, cellQualName, modelDomainName;  
          void: rsvd1, rsvd2; );
```

7.10.5 TECH_TYPE

TECH_TYPE (Table 23) represents the internal representation of the a specific technology. It is returned as a result of calling several different default library functions. These structures are created by the run-time environment and cannot be copied or created using the new operator.

Table 23—TECH_TYPE

```
typedef (TECH_TYPE):  
    result(string: name;  
          int: "DEFAULT", dcmInfo;  
          void: reserved; );
```

7.10.5.1 TECH_TYPE field: name

name is the tech_family name this TECH_TYPE represents.

7.10.5.2 TECH_TYPE field: DEFAULT

DEFAULT result contains the tech_family index used by the run time. This field shall not be modified.

7.10.5.3 TECH_TYPE fields: dcmInfo and reserved

dcmInfo and reserved are undefined and shall not be modified.

7.10.6 DELAY_REC_TYPE

DELAY_REC_TYPE (Table 24) contains two float fields, one for the early delay and the other for the late delay.

Table 24—DELAY_REC_TYPE

```
typedef (DELAY_REC_TYPE):  
    result(float var: early, late);
```

7.10.7 SLEW_REC_TYPE

SLEW_REC_TYPE (Table 25) contains two float fields, one for the early slew and the other for the late slew.

Table 25—SLEW_REC_TYPE

```
typedef(SLEW_REC_TYPE):
    result(float var: early, late);
```

7.10.8 CHECK_REC_TYPE

CHECK_REC_TYPE (Table 26) contains a single float field representing a bias calculation result.

Table 26—CHECK_REC_TYPE

```
typedef(CHECK_REC_TYPE):
    result(float var: bias);
```

7.10.9 CCDB_TYPE

CCDB_TYPE (Table 27) is a type definition used within the Standard Structure. Its use is reserved for the run-time environment and shall not be created, copied, or modified. Its definition is only present to aid in the description of the Standard Structure.

Table 27—CCDB_TYPE

```
typedef(CCDB_TYPE):
    result(int: sfiCount;
           void: sfi, destructor, reserved0, anchor;
           short: ci, flags;
           int: methodsIndex;
           void: reserved1, reserved2, reserved3, reserved4; );
```

7.10.10 CELL_DATA_TYPE

CELL_DATA_TYPE (Table 28) is generated by the run-time environment during modeling. Typically, they are used during calculations after the cell has been modeled. This definition is to aid in the description of the Standard Structure. The CELL_DATA_TYPE shall not be modified, copied, or created using the new operator.

Table 28—CELL_DATA_TYPE

```
typedef(CELL_DATA_TYPE):
    result(void: recallData;
           CCDB_TYPE: ccdb;
           int: usageCount;
           void: reserved1;
           short: flags;
           void: cause, reserved3, reserved4; );
```

7.10.11 PCDB_TYPE

PCDB_TYPE (Table 29) is a type definition used within the Standard Structure. Its use is reserved for the run-time environment and shall not be created, copied, or modified. Its definition is only present to aid in the description of the Standard Structure.

Table 29—CCDB_TYPE

```
typedef(PCDB_TYPE):
result(string: clkflg, objectType;
    int: delayAdj;
    void: delay, slew;
    int: ci, sfiCount;
    void: sfi, destructor, anchor, reserved0, reserved1, reserved2,
        reserved3;
    int: methodsIndex; );
```

7.10.12 PIN_ASSOCIATION

PIN_ASSOCIATION (Table 30) is used to relate an application pin to an arbitrary DCL library structure.

Table 30—PIN_ASSOCIATION

```
typedef(PIN_ASSOCIATION):
    result(pin var: pinHandle;
        var abstract var: pinInfo; );
```

7.10.13 PATH_DATA_TYPE

PATH_DATA_TYPE (Table 31) is generated by the run-time environment during modeling. Typically, it is used during calculations after the cell has been modeled. This definition is to aid in the description of the Standard Structure. The PATH_DATA_TYPE shall not be modified, copied, or created using the new operator.

Table 31—PATH_DATA_TYPE

```
typedef(PATH_DATA_TYPE):
    result(string: path;
        void: recallData;
        PCDB_TYPE: pcdB;
        int: usageCount;
        short: flags, cycleAdj, corrind, modifiers,
            msbStrandSource, lsbStrandSource,
            msbStrandSink, lsbStrandSink; );
```

7.10.14 STD_STRUCT

The STD_STRUCT (Table 32) is the first implicit argument to all DCL statements. The STD_STRUCT is made up of fields that can be modified and those that shall not be modified. To exercise a library module effectively as an application, Standard Structures need to be created and modified.

Table 32—STD_STRUCT

```
typedef(STD_STRUCT):
  result(int: dcminfo;
    void: states;
    string var: cell, cellQual, modelDomain, ctl, model_name,
      instantiated, expanded;
    pin var: block;
    void: inputPins, outputPins, nodes;
    pin var: fromPoint, toPoint;
    int var: inputPinCount, outputPinCount, nodeCount,
      sourceEdge, sinkEdge, sourceMode;
    int var: sinkMode, calcMode;
    var SLEW_REC_TYPE : slew;
    PATH_DATA_TYPE var: pathData;
    void var: applicationInfo;
    CELL_DATA_TYPE var: cellData;
    pin var: utilityHandle;
    int var: processVariation;
    var PIN_ASSOCIATION var: fromPointPinAssociation,
      toPointPinAssociation;
    void: reserved2, reserved3, reserved4, reserved5,
      reserved6);
```

7.11 Predefined variables

DCL contains variables whose names and types have been predefined. Predefined variable names are keywords that are available in both uppercase or lowercase variants.

7.11.1 ARGV

ARGV (Table 33) is an array of strings containing the arguments as they are passed to the application. Each string represents an argument in the list supplied to the application. The order of the strings in the array is the order the arguments were passed to the application. ARGV only contains valid information if the application is written in DCL using the TECH_FAMILY MAIN option. Applications written in other languages such as C the argv array shall contain no elements.

Table 33—ARGV

```
string[*]: ARGV
```

7.11.2 CONTROL_PARM

CONTROL_PARM (Table 34) contains the control parameter string with which the rule was loaded. When a subrule is loaded either by the application or another subrule.

Table 34—CONTROL_PARM

```
string: CONTROL_PARM
```

7.12 Built-in function calls

A built-in function is a function built as part of the compiler. Their names are reserved words. There are several built-in functions to perform a variety of operations. Most provide access to information that could not otherwise be accessed.

7.12.1 ABS

ABS (Syntax 7.56) returns the absolute value associated with its argument. When the argument type is int, short, or char, the result's type is int. When the argument type is float or double, the result's type is double. No other types shall be allowed as arguments.

```
expression ::= expression
            | ABS ( integer_expression | double_expression )
```

Syntax 7.56—ABS

7.12.2 Complex number components

To access the individual components of a complex number, there are two built-in functions, `imag_part` and `real_part`.

7.12.2.1 IMAG_PART

IMAG_PART (Syntax 7.57) takes as an argument a complex number and returns the imaginary component. The type of the result is double.

```
expression ::= expression
            | IMAG_PART ( complex_type_expression )
```

Syntax 7.57—IMAG_PART

7.12.2.2 REAL_PART

REAL_PART (Syntax 7.58) takes as an argument a complex number and returns the real component. The type of the result is double.

```
expression ::= expression
            | REAL_PART ( complex_type_expression )
```

Syntax 7.58—REAL_PART

7.12.3 EXPAND

Expand (Syntax 7.59) takes as an argument a pin range and expands the pin range into a list of pins. The expanded pin range list is a one-dimensional array of strings in which each string contains the name of one of the pins in the pin range. The strings are constant, and it shall be an error to free them.

```
expression ::= expression | EXPAND ( pin_range )
```

Syntax 7.59—EXPAND

7.12.4 Array functions

7.12.4.1 IS_EMPTY

IS_EMPTY (Syntax 7.60) takes an array as an argument and returns whether the array contains any elements. The result type is int where the value of zero indicates the array has elements and a value of one indicates the array contains no elements.

```
expression ::= expression | IS_EMPTY ( array_expression )
```

Syntax 7.60—IS_EMPTY

7.12.4.2 NUM_DIMENSIONS

NUM_DIMENSIONS (Syntax 7.61) takes as an argument that is an array any type and returns the number of dimensions contained in that array as an int.

```
expression ::= expression  
| NUM_DIMENSIONS ( array_expression )
```

Syntax 7.61—NUM_DIMENSIONS

7.12.4.3 NUM_ELEMENTS

NUM_ELEMENTS (Syntax 7.62) returns the number of elements contained in the specified dimension of the array. The function takes two arguments: The first is the array to query, and the second is the dimension of interest. The first argument is an array of any type, and the second is of type int.

```
expression ::= expression  
| NUM_ELEMENTS ( array_expression , int_expression )
```

Syntax 7.62—NUM_ELEMENTS

7.12.5 Messaging functions

This subclause contains the built-in functions contained within the language.

7.12.5.1 ISSUE_MESSAGE

ISSUE_MESSAGE (Syntax 7.63) creates an error whose level is that of the severity argument and issues a text message. The text message is composed with similar syntax and semantics as the ANSI C printf function.

```
expression ::= expression  
| ISSUE_MESSAGE ( int_expression , int_expression , string_expression  
{ , expression } )
```

Syntax 7.63—ISSUE_MESSAGE

7.12.5.1.1 Arguments

The arguments are as follows:

- The first argument to ISSUE_MESSAGE is the message number. The first 10 000 message numbers are reserved for the system. If the first argument has a value of 10 000 or less, then a value of 10 000 shall be added to the first argument before processing the message.
- The second argument shall be the message severity. The value of the second argument shall conform to the system error severity values. There is a set of predefined constants associated with error severities in which each predefined constant represents the value associated with its severity.
- **INFORM**
INFORM represents the value associated with the severity of an informative message. The error code returned by issue_message when an inform message is issued is zero and no error exit is taken.

- **WARNING**
WARNING represents the value associated with the message severity of a warning message. The error code returned by `issue_message` when an warning message is issued is zero and no error exit is taken.
- **ERROR**
ERROR represent the value associated with the message severity of an error message. The error code returned by `issue_message` when an error message is issued is the error number with the error severity inserted as the high-order byte.
- **SEVERE**
SEVERE represents the value associated with the message severity of severe. The error code returned by `issue_message` when a severe error message is issued is the error number with the severe severity inserted as the high-order byte.
- **TERM**
TERM represents the value associated with the message severity of terminate. The error code returned by `issue_message` when a terminate error message is issued is the error number with the terminate severity inserted as the high-order byte.
- The third argument is the format string. It shall conform to the syntax and semantics of the ANSI C `printf` format string.
- Any additional parameters shall conform to the format operators used in the format string and appear in the order the format operators appeared in the format string.

7.12.5.1.2 Result

The result is default field if the integer type and whose value is the calculated message number. If the severity argument is either **INFORM** or **WARNING**, then no error is generated and a value of zero is returned as message number.

7.12.5.2 PRINT_VALUE

PRINT_VALUE (Syntax 7.64) prints the value of its argument. Printing is conditional on the debug setting. If debug is not set to **OFF**, then the values shall be printed; otherwise, no action is taken. **PRINT_VALUE** has no result.

```
expression ::= expression | PRINT_VALUE ( expression )
```

Syntax 7.64—PRINT_VALUE

7.12.5.3 SOURCE_STRANDS_MSB

SOURCE_STRANDS_MSB (Syntax 7.65) returns the value of the most significant strand at the source end of the bus. This is a value carried in the Standard Structure that shall only contain valid information during those calls identified by the application interface clause of this specification.

```
expression ::= expression | SOURCE_STRANDS_MSB
```

Syntax 7.65—SOURCE_STRANDS_MSB

7.12.5.4 SOURCE_STRANDS_LSB

SOURCE_STRANDS_LSB (Syntax 7.66) returns the value of the least significant strand at the source end of the bus. This is a value carried in the Standard Structure that shall only contain valid information during those calls identified by the application interface clause of this specification.


```
expression ::= expression | SOURCE_STRANDS_LSB
```

Syntax 7.66—SOURCE_STRANDS_LSB

7.12.5.5 SINK_STRANDS_MSB

SINK_STRANDS_MSB (Syntax 7.67) returns the value of the most significant strand at the sink end of the bus. This is a value carried in the Standard Structure that shall only contain valid information during those calls identified by the application interface clause of this specification.

```
expression ::= expression | SINK_STRANDS_MSB
```

Syntax 7.67—SINK_STRANDS_MSB

7.12.5.6 SINK_STRANDS_LSB

SINK_STRANDS_LSB (Syntax 7.68) returns the value of the most significant strand at the sink end of the bus. This is a value carried in the Standard Structure that shall only contain valid information during those calls identified by the application interface clause of this specification.

```
expression ::= expression | SINK_STRANDS_LSB
```

Syntax 7.68—SINK_STRANDS_LSB

7.13 Tables

A DCL table is a collection of one-dimensional vector(s) of data. Each data vector (called a row) shall have the same structure (i.e., the same number of data values, which are also called “fields”) and the same sequence of DCL data types. Each table row shall be associated with a set of qualifiers, which are used to select the desired row during table search operations. Table data can be created at compile time (static tables) or at run time (dynamic tables). Table data can be read from or written to a mass storage device.

Tables shall be defined by a TABLEDEF statement together with one or more TABLE statements. The TABLEDEF statement defines the name of the table, data format, and search criteria; the TABLE statement groups data for compiled tables in a collection of rows.

NOTE—The order of the data rows in a table has no effect on the searching for a matching row (see 7.13.3.7).

7.13.1 TABLEDEF statement

The syntax for the TABLEDEF statement is given in Syntax 7.69.

```
tabledef_statement ::= tabledef_preamble [passed_clause] tabledef_clauses
;
tabledef_preamble ::= [EXPORT] TABLEDEF ( name ) std_postfix_modifier
tabledef_modifiers :
tabledef_clauses ::=
    qualifiers_clause data_clause default_clause [key_clause]
| qualifiers_clause data_clause [key_clause] default_clause
| qualifiers_clause [key_clause] data_clause default_clause
| [key_clause] qualifiers_clause data_clause default_clause
```

Syntax 7.69—tabledef_statement

The TABLEDEF statement shall define the name of the table, input parameters, options, returned data format, and the variable references used to match the table row qualifiers. When the TABLEDEF statement is called, the associated table is searched for the row that matches the qualifiers, and if found, it returns the data contained in that row. If a row matching the qualifiers is not found, then an error is returned.

7.13.1.1 QUALIFIERS clause

The QUALIFIERS clause shall define the variables whose values are used to match the qualifier data associated with each table row during table search operations. The number of the variables listed in this clause shall match the number of qualifiers specified for each row in the TABLE statement of this table, with the exception of the PROTOTYPE_RECORD row and the DEFAULT row. The order of the variables in this clause is significant, as the variable's value is compared to the qualifier in the same position from the TABLE statement (see 7.13.3.7).

All qualifier variables shall be of type STRING or are converted to STRING using the following rules:

- Variables of type PIN shall be converted to the type STRING using DCL's implicit conversions (see 7.4.5.3.3.1).
- Variables of type INT shall be converted to type STRING as generated by sprintf() using a format of %d (Section 7.9.6.1 of ISO/IEC 9899:1990).
- Variables of type NUMBER, FLOAT and DOUBLE shall be converted to type string as generated by sprintf using the format of %.0+ (see Section 7.9.6.1 of ISO/IEC 9899:1990).
- PINLIST, VOID, aggregate types and array types shall not be used in the QUALIFIERS clause.

7.13.1.2 DATA clause

The data clause shall define the name and data type of each field returned by a table search operation. For scalar data types (INT, FLOAT, STRING, etc.), these variables shall correspond one to one to the data values in each row of the TABLE statement. For array data types, these variables shall correspond one to one with data fields enclosed within square brackets.

The pointer types of *TECH_TYPE*, PIN, PINLIST, and VOID shall not be allowed in the DATA clause of a TABLEDEF statement.

7.13.1.3 KEY clause

The optional key clause shall define a decryption key as a method of keeping table data private. If a KEY is used in the TABLEDEF statement that defines the table at compile time, the same KEY shall be used in the TABLEDEF statement that defines and loads the table at run time.

The KEY clause shall be valid only for static tables.

7.13.1.4 OVERRIDE modifier

The contents of a table can be built up as the result of merging together information from multiple TABLE statements with the same table name. This merging is always allowed within a single compilation unit, but it is only allowed across compilation units if the OVERRIDE modifier is specified on the IMPORTED TABLEDEF statement prototype (see 7.13.3.9).

The OVERRIDE modifier shall be valid only for static tables. Table merging shall occur only among subrules of the same *TECH_FAMILY*.

7.13.1.5 SUPPRESS modifier

The suppress modifier prevents the table search functions associated with the tabledef from issuing error messages when a row that is being searched for is not in the table.

7.13.1.6 DESCRIPTOR modifier

Tabledefs without the DESCRIPTOR defines a single instance of a table. This instance can either be static, which is loaded when the table is first searched, or be dynamic and read in when the DPCM requests it. Tabledefs with the descriptor modifier define a general table description that can be used for any number of table instances.

A table DESCRIPTOR is a type defined by the tabledef and is known by the descriptor name. Tabledef descriptors are allocated through use of the operator new. All the statements associated with a tabledef containing a descriptor clause shall have as an implicit first argument a structure var tabledef descriptor. The association to a specific tabledef is created by statements (LOAD_TABLE, UNLOAD_TABLE, WRITE_TABLE, LOAD_TABLE, ADD_ROW, and DELETE_ROW) that have tabledef clauses identifying the specific tabledef statement it is to be associated with.

A new table instance is created when a call is made to LOAD_TABLE with a table descriptor that does not have a table in memory associated with it. A table instance is removed when a call to UNLOAD_TABLE is made using a table descriptor. A specific instance of a table may be modified by calls to ADD_ROW, DELETE_ROW, and LOAD_TABLE using the descriptor.

7.13.1.7 DYNAMIC modifier

The DYNAMIC modifier identifies the tables associated with this table may be modified using ADD_ROW and DELETE_ROW.

7.13.1.8 DEFAULT clause

The optional DEFAULT clause shall define a set of values to return if a qualifier search fails. The DEFAULT clause shall be specified only for static tables or EXTERNAL TABLEDEF statements. The DEFAULT clause in the TABLEDEF statement shall be overridden by the DEFAULT row (see 7.13.3.3).

7.13.2 Table visibility rules

A TABLEDEF statement may be static, EXPORTed, IMPORTed, or IMPORTed with an OVERRIDE modifier. The TABLE corresponding to a TABLEDEF statement shall exist in the same subrule when the defining TABLEDEF is static, EXPORTed, or IMPORTed with an OVERRIDE statement.

The following rules also apply to all TABLEDEF:

- A table defined by a static TABLEDEF shall be visible only within its compilation unit.
- A table defined by an EXPORTTABLEDEF shall be visible within its compilation unit and within any other compilation unit the same TECH_FAMILY and an IMPORT of that TABLEDEF.
- A compilation unit that contains a static TABLEDEF, an EXPORTTABLEDEF, or an IMPORTed TABLEDEF that specifies the OVERRIDE modifier shall contain at least one TABLE statement with the same name as the TABLEDEF.
- A compilation unit that IMPORTs a TABLEDEF and does not specify the OVERRIDE modifier shall not contain any TABLE statements with the same name as the TABLEDEF.

7.13.3 TABLE statement

Tables represent the data associated with a tabledef. The data in a table shall conform to the definition established in the associated tabledef. For static compiled tables, the tabledef shall not have the modifiers DYNAMIC or DESCRIPTOR and shall have the same name as the TABLEDEF that defined its organization.

Tables associated with a tabledef with the DYNAMIC modifier can be stored on disk in either a binary format or an ASCII format. The ASCII format table shall consist of a collection of table records that have the same syntax and semantics as those contained the TABLE statement (Syntax 7.70).

```

table_statement ::= TABLE ( name ) [table_postfix_modifier] :
    prototype_default_records table_records END ;
table_postfix_modifier ::= COMPRESSED
prototype_default_records ::= [prototype_record] [default_record]
prototype_record ::= PROTOTYPE_RECORD : table_data_fields ;
default_record ::= DEFAULT : table_data_fields ;
table_data_fields ::= table_data_element { [ , ] table_data_element }
table_array_dimension ::= [ [ table_data_fields ] ]
table_multi-dimensional_array ::=
    [ [ table_multi-dimensional_array ]
    { [ , ] [ table_multi-dimensional_array ] }
    | [ [ table_array_dimension ]
    { [ , ] [ table_array_dimension ] } ]
table_data_element ::= statement_name | constant | table_structure |
    complex_number | table_array
constant ::= string_literal | integer_constant | floating_point_constant
table_structure ::= statement_name :
    { table_data_element { , table_data_element } }
complex_number ::= ( real_part , imaginary_part )
real_part ::= floating_point_constant
imaginary_part ::= floating_point_constant
table_records ::= table_record { table_record }
table_record ::= table_qualifier_list : table_data_fields ;
table_qualifier_list ::= table_qualifier { , table_qualifier }
table_qualifier ::= double_quoted_literal_string

```

Syntax 7.70—table_statement

For variables of type array declared in the data clause of the TABLEDEF statement, the data values to be returned in the array shall be enclosed within square brackets. For a multidimensional array, the values for each dimension shall be enclosed within nested square brackets. The number of values contained within each set of nested square brackets for the same array shall contain the same number of elements.

7.13.3.1 Static tables

Static tables are compiled and stored in memory image. The static tables are read only and shall not be modified. The var type modifier shall not be used in static tables.

The TABLE statement shall define data values for static tables. Each TABLE statement requires a corresponding TABLEDEF statement. The first TABLE statement encountered in the scope of the static or EXTERNAL TABLEDEF statement shall be considered the original TABLE statement for this table.

The TABLE statement groups data into rows. Each row consists of a set of qualifiers followed by a set of data fields. The qualifiers of each row correspond to the variables specified in the QUALIFIERS clause in the associated TABLEDEF statement. The data fields correspond to the DATA clause in the associated

TABLEDEF statement. The number and type of the qualifiers and data fields shall match the qualifiers and data fields specified in the associated TABLEDEF statement.

Within a single compilation unit there may exist more than one TABLE statement of the same name. The tables are appended as they lexically appear in the source with the following restrictions:

- The DEFAULT record (if present) shall appear in the first table in the source and follows the rules for default rows.
- The PROTOTYPE_RECORD shall appear in the first table encountered and shall follow all the rules for prototype_records.
- It shall be considered an error to have two or more data rows in one or more tables associated with the same TABLEDEF that have identical qualifier sequences.
- All rows in each table associated with a TABLEDEF shall conform to the DATA and QUALIFIER clauses defined in that TABLEDEF.

Once a table is loaded, independent of whether it originated from one TABLE statement or many, each row of a table shall be uniquely identified by its qualifiers (see 7.13.1.2). Hence, the table search mechanism is independent of the row order (e.g., the order of the table rows shall never determine which row is returned).

7.13.3.2 PROTOTYPE_RECORD row

The original TABLE statement of a static table may contain the PROTOTYPE_RECORD row as its first data row. In this case, the qualifier for this row is the keyword PROTOTYPE_RECORD. Only one PROTOTYPE_RECORD row shall be allowed per table.

If a PROTOTYPE_RECORD row is present, then any subsequent TABLE rows that do not have the full number of data fields shall be filled out by copying the appropriate number of trailing fields from the PROTOTYPE_RECORD row.

7.13.3.3 DEFAULT row

The original TABLE statement for a table may contain the DEFAULT row as its first data row (or as its second data row if the PROTOTYPE_RECORD row is present). The qualifier for the DEFAULT row is the keyword DEFAULT. Only one DEFAULT row shall be allowed per table. If a DEFAULT row is defined for a table, then that row's data fields shall be returned if a table search operation does not match any of the other row qualifiers.

If a TABLEDEF statement has a DEFAULT clause and the corresponding TABLE has a DEFAULT row, the TABLEDEF DEFAULT clause shall never be exercised because the qualifier search can never fail (except in the case of a reference to a DYNAMIC table that has not been loaded in memory).

If a DEFAULT row is present but no PROTOTYPE_RECORD is present, any subsequent TABLE rows that do not have the full number of data fields shall be filled out by copying the values from the corresponding fields of the DEFAULT row.

7.13.3.4 Default operator as table row qualifier

The default operator* may be used for one or more qualifiers in a static table row.

It shall be an error to use the default operator * in a dynamic table row.

7.13.3.5 Default operator in a table reference

It shall be an error to pass the default operator * as the value to be matched when referencing a TABLE.

7.13.3.6 String prefix operator

The string prefix operator * (see 7.6.1) can prefix a nonempty string used as a qualifier component in either a table row or a table reference.

7.13.3.7 Qualifier matching

The search for matching qualifiers in a table reference for any given row shall proceed from the first qualifier component to the last qualifier component in left-to-right order. The default operator * if present, shall be matched only if an exact match between the given qualifier and each qualifier component fails. If no qualifier matches are found and the default row is specified, values from the default row shall be returned.

The order of the rows in the table shall not affect the matching process. Matching shall be undefined if multiple table row qualifiers use the string prefix operator and can potentially match the given qualifier string.

7.13.3.8 COMPRESSED modifier

The compressed modifier in the table statement shall be a hint to the DCL compiler that table storage space may be saved by removing values from data rows that duplicate values specified in the PROTOTYPE_RECORD row (or the DEFAULT row, if there is no PROTOTYPE_RECORD row).

The COMPRESSED modifier shall be used only if the table has either a PROTOTYPE_RECORD or a DEFAULT row. The COMPRESSED modifier shall only be allowed on the original TABLE statement.

7.13.3.9 Duplicate table rows

A duplicate row is defined as one in which all the qualifiers match those in an existing table row for a given table name. A qualifier that contains the string prefix operator matches another qualifier if it matches the same strings as the other qualifier.

For static tables, duplicate table rows shall not be allowed within a single compilation unit. Duplicate table rows shall be allowed in TABLE statements that exist in separate compilation units (see 7.13.1.4). When duplicate table rows are found in separately compiled table statements, the latter used the OVERRIDE modifier; the TABLE statement's data that are loaded later shall supersede the existing table data.

7.13.3.10 Dynamic tables

Dynamic tables are loaded into memory at run time (using the LOAD_TABLE statement) on request of the DPCM. These tables can be modified once they are loaded into memory using the ADD_ROW and DELETE_ROW statements. These tables can also be unloaded from memory (using the UNLOAD_TABLE statement) or written to disk (using the WRITE_TABLE statement) on request of the DPCM.

A table shall be designated as dynamic with the DYNAMIC modifier on the appropriate TABLEDEF statement.

7.13.3.10.1 Dynamic table syntax

Dynamic table data shall be read from a file and shall have the same syntax and semantics as the TABLE statement for static tables, with the following exceptions:

- Dynamic table data shall consist only of the table row data. Specifically, there shall be no TABLE keyword, table name, modifiers, or colon (:) at the beginning of the table description. In addition, there shall be no END keyword or trailing semicolon (;) at the end of the table description.
- There shall be no other data in the file except the table row data.
- The default operator (*) shall not be used as a table row qualifier.
- The table shall not contain a PROTOTYPE_RECORD.

7.13.3.10.2 Limitations

The OVERRIDE modifier and the KEY clause shall be considered illegal in a dynamic TABLEDEF statement.

7.13.3.10.3 Dynamic table manipulation

This subclause details dynamic table manipulation within DCL.

7.13.4 LOAD_TABLE statement

The syntax for the LOAD_TABLE statement is given in Syntax 7.71.

```
load_table_statement ::= [EXPORT] LOAD TABLE ( name )
                        std_postfix_modifier [opt_replace] :
                        [passed_clause] TABLEDEF ( name_of_tabledef )
                        [opt_file_filter_paths] [opt_integer_default] ;
opt_replace ::= REPLACE
opt_file_filter_paths ::= opt_file_filter_path {opt_file_filter_path}
opt_file_filter_path ::= ( FILE | FILTER | PATH | SUFFIX ) ( string_exp )
opt_integer_default ::= DEFAULT
                    ( [local_clause] integer_expression )
```

Syntax 7.71—load_table_statement

The contents of a dynamic table shall be loaded into memory as defined by the LOAD_TABLE statement. LOAD_TABLE can specify the data be loaded directly from a file or run through a filter program. In either case, when the data are received by the statement, they shall conform to the QUALIFIER and DATA clauses of the associated TABLEDEF statement (see 7.13.1).

LOAD_TABLE shall not be declared CONSISTENT.

7.13.4.1 Restrictions

There shall be one FILE clause. At most, there shall be one instance of each FILE, FILTER, PATH, and SUFFIX clause. The combination of FILE, FILTER, and/or PATH statements shall point to either a file containing legal table row statements or a program that produces legal table row statements. Each expression contained within the FILE, FILTER, and PATH clauses shall be of type string.

7.13.4.2 TABLEDEF clause

The TABLEDEF clause shall reference the name of a TABLEDEF statement that is visible in the current scope. This TABLEDEF statement shall have the DYNAMIC modifier.

7.13.4.3 Result value

The LOAD_TABLE statement has an implicit result (there is no RESULT clause) whose data type is INTEGER. This implicit result shall be set to zero (0) if the statement is successful in reading the table data file; otherwise, it shall be set to a nonzero value that refers to an error code.

7.13.4.4 FILE clause

The FILE clause shall designate the file name containing the table rows. The extension `.table` shall be appended to the file name, so the actual table data file in the file system needs to have this extension or the DPCM shall not be able to find it. If the file has zero length, then an empty table shall be created. If the FILTER clause is not present, the FILE shall be read in as is.

7.13.4.5 SUFFIX clause

The SUFFIX clause overrides the default extension of `.table`. The string expression argument replaces the default extension of `.table` with the string supplied.

Example

`load_table(t): tabledef(td) file('myTable') suffix("");` loads a table with definition defined by `td` and with the name “myTable” using the environment variable `DCMTABLEPATH`.

7.13.4.6 FILTER clause

The FILTER clause shall contain a STRING that is passed to a shell and whose execution is expected to read information from `stdin` and generate table row statements on `stdout` in the format required.

If the FILTER clause is used in conjunction with the FILE clause, the `stdin` shall read from the file named with “`.table`” appended.

7.13.4.7 PATH clause

The PATH clause shall contain a STRING that designates the name of an environment variable (UNIX) or a user variable (Windows NT) used to search for the FILE. If the PATH clause is not present, the file shall either be located in the current directory or the FILE clause shall include the complete path name to that file. The value of the environment or user variable shall contain a colon delimited list of file system directory names.

meta-variables can also be used in the PATH clause (see 7.15.6).

7.13.4.8 DEFAULT clause

A LOAD_TABLE statement shall fail when the file name or filter program is not found or if the filter program returns a nonzero code. (Zero records read shall not cause an error.) If the LOAD_TABLE statement fails and there is no DEFAULT clause, a nonzero return code shall be sent back to the DPCM. If there is a DEFAULT clause, then that clause (see 7.13.1.8) shall be executed and the result of that clause shall be returned to the DPCM.

7.13.4.9 REPLACE modifier

If the REPLACE modifier is specified and a duplicate table row is encountered, older table data shall be replaced by newer data. If a duplicate table row is encountered during a dynamic table load and the REPLACE modifier is absent, it shall be considered an error.

7.13.4.10 Descriptor

LOAD_TABLE statements that are associated with a TABLEDEF that had a DESCRIPTOR modifier have as their first argument a table descriptor. The LOAD_TABLE checks the table descriptor to determine if the descriptor contains a table already loaded in memory. If a table already exists, the new table is appended to the existing; otherwise a new table image is created and the table is loaded into it.

7.13.5 UNLOAD_TABLE statement

The syntax for the UNLOAD_TABLE statement is given in Syntax 7.72.

```
unload_table_statement ::= [EXPORT] UNLOAD_TABLE ( name )
    std_postfix_modifier unload_table_options : [passed_clause] TABLEDEF
    ( name_of_tabledef ) [opt_file_filter_path] [DEFAULT (
    [local_clause] integer_expression ) ] ;
unload_table_options ::= { unload_table_option }
unload_table_option ::= APPENDABLE | BINARY | FREE_SPACE | INTERNAL
```

Syntax 7.72—unload_table_statement

The following conditions apply:

- The UNLOAD_TABLE statement shall write an in-memory DYNAMIC table out to the file specified in the form expected by the LOAD_TABLE statement and shall delete the table from memory. The file format of the output shall be table records written in ASCII characters.
- The UNLOAD_TABLE has the same fail conditions as the LOAD_TABLE statement (see 7.13.1.8). The DEFAULT clause, if present, shall be executed if this statement fails.
- The syntax and semantics of the TABLEDEF, FILE, FILTER, and PATH clauses shall be the same as those described in the LOAD_TABLE statement (see 7.13.4), except the file specified shall be the output file and the filter specified, if any, shall filter the data written to this file.
- The UNLOAD_TABLE statement has as an implicit result (there is no RESULT clause) whose data type shall be INTEGER. This implicit result shall be set to zero (0) if the statement is successful in writing the file; otherwise it shall be set to a nonzero value.
- The expression within the DEFAULT clause, if present, shall evaluate to an integer.
- The UNLOAD_TABLE shall not be declared CONSISTENT.
- The UNLOAD_TABLE statement without FILE and FILTER shall empty the table without writing any data to a file.

7.13.5.1 Descriptor

UNLOAD_TABLE statements that are associated with a tabledef that had a DESCRIPTOR modifier have as their first argument a table descriptor. The UNLOAD_TABLE uses the descriptor to determine which table to unload and updates the descriptor to indicate there is no longer a table image in memory.

7.13.5.2 APPENDABLE modifier

The APPENDABLE modifier causes the table to be appended to the end of a file. If no file of the specified name exists, a new one shall be created. Without the appendable modifier, a new file shall be created. If a of the same name already exists, it shall be overwritten with the table data.

7.13.5.3 BINARY modifier

The BINARY modifier shall cause the table to being unloaded to be written in memory image. Without the BINARY modifier, the table shall be written in ASCII format.

7.13.5.4 FREE_SPACE modifier

The FREE_SPACE modifier shall cause the data members for type string stored in the table that is being unloaded to be returned to the system.

7.13.5.5 INTERNAL modifier

When the unload_table statement unloads a table containing statement types, the module containing the statement pointed to is recorded as well as the statement name. This allows the linker to reestablish the statement reference that was in the table when it is written. In some situations, this behavior is not what is desired. The INTERNAL modifier causes the unload to write all statement pointers as though they were contained in the module containing the UNLOAD_TABLE statement. The INTERNAL modifier is useful in situations when the table that is being unloaded is intended to be loaded by a different system of rules. The module that does the loading shall contain executable statements of the same name and parameter sequences as those contained in the table.

7.13.6 WRITE_TABLE statement

The WRITE_TABLE performs a similar operation and has the same syntax and semantics as the UNLOAD_TABLE statement. WRITE_TABLE leaves the memory image of the table intact, whereas the JNLOAD_TABLE removes it.

The syntax for the WRITE_TABLE statement is given in Syntax 7.73.

```
write_table_statement ::= [EXPORT] WRITE_TABLE ( name )
                        std_postfix_modifier : [passed_clause]
                        TABLEDEF ( name_of_tabledef ) [opt_file_filter_path] [DEFAULT (
                        [local_clause] integer_expression ) ] ;
```

Syntax 7.73—unload_table_statement

7.13.6.1 Descriptor

WRITE_TABLE statements that are associated with a tabledef that had a descriptor modifier have as their first argument a table descriptor. The write_table statement uses the table descriptor to determine which table to write.

7.13.7 ADD_ROW statement

The syntax for the ADD_ROW statement is given in Syntax 7.74.

```
add_row_statement ::= [EXPORT] ADD_ROW ( name ) std_postfix_modifier
                    [opt_replace] :
                    TABLEDEF ( name_of_tabledef ) [DEFAULT ( expression ) ] ;
```

Syntax 7.74—add_row_statement

The ADD_ROW statement declares *name* as a function that can be called to add a single row to a dynamic table. The ADD_ROW statement shall not be declare CONSISTENT.

7.13.7.1 TABLEDEF clause

The TABLEDEF clause shall identify the name of a TABLEDEF statement in the current scope that was declared with the DYNAMIC modifier. The row to be added shall match the qualifier sequence and data format of the identified TABLEDEF.

7.13.7.2 Passed parameters

Although there is no PASSED clause in the definition of the ADD_ROW statement, passed arguments to the declared statement shall be required. If the add_row statement is associated with a tabledef that has a descriptor modifier, the first argument shall be a table descriptor and the remaining argument shall be the number, order, and types of these arguments that are exactly those specified by the concatenation of the variables in the QUALIFIERS and DATA clauses of the associated TABLEDEF statement.

7.13.7.3 DEFAULT clause

The DEFAULT clause allows another action to be taken should the statement declared by the ADD_ROW statement fail. A failure shall occur if the row to be added is a duplicate of an existing row (i.e., has the same qualifiers) and the REPLACE modifier was not specified.

7.13.7.4 REPLACE modifier

Unless the REPLACE modifier is specified, it shall be an error to add a duplicate row. If the REPLACE modifier is specified and a duplicate table row is encountered, older table data shall be replaced by newer data.

7.13.7.5 Result value

The ADD_ROW statement has as an implicit result (there is no RESULT clause) whose data type is INTEGER. This implicit result shall be set to zero (0) if the statement is successful in adding the row; otherwise it shall be set to a nonzero value.

7.13.7.5.1 Descriptor

For add_row statements that are associated with a tabledef that contains a descriptor modifier, use the descriptor to determine to which table the row is added. It shall be an error to call add_row supplying a descriptor that does not have a memory image associated with it.

7.13.8 DELETE_ROW statement

The syntax for the DELETE_ROW statement is given in Syntax 7.75.

```
delete_row_statement ::= [EXPORT] DELETE_ROW ( name ) std_postfix_modifier
                        [FREE_SPACE] :
                        TABLEDEF ( tabledef_statement_name ) [default_clause] ;
```

Syntax 7.75—delete_row_statement

The DELETE_ROW statement shall delete a single row from a dynamic table. The DELETE_ROW statement shall not be declared CONSISTENT.

7.13.8.1 TABLEDEF clause

The TABLEDEF clause shall identify the name of a TABLEDEF statement and its corresponding TABLE, from which the row is to be deleted. The TABLEDEF clause shall identify a TABLEDEF in the current scope that was declared with the DYNAMIC modifier.

7.13.8.2 Passed parameters

Although there is no passed clause in the definition of the DELETE_ROW statement, passed arguments to the declared statement shall be required. The number, order, and types of these arguments shall be a descriptor if the delete_row statement is associated with a tabledef that contains a descriptor modifier plus exactly those specified by the QUALIFIERS clause of the associated TABLEDEF statement.

A call to the statement declared by DELETE_ROW shall attempt to match the passed qualifier values against the qualifiers associated with each table row. If a match is found, the matching row shall be deleted from the table.

7.13.8.3 Result value

The DELETE_ROW statement has as an implicit result (there is no result clause) whose result variable order, types, and names shall be the same as those specified in the DATA clause of the TABLEDEF statement for this table. When the reference to a DELETE_ROW statement is successful, this reference shall return the data values for the deleted row.

7.13.8.4 DEFAULT clause

The DEFAULT clause (if present) shall be invoked if the parameters passed in a statement reference fails to match all the qualifiers for any of the table rows. The DEFAULT clause defines the values to be returned when the DELETE_ROW statement reference fails. A failure shall occur if a matching row cannot be found and there is no default clause.

The variable order, types, and names referenced in this clause shall be the same as those specified in the DATA clause of the TABLEDEF statement for this table.

7.13.8.4.1 Descriptor

For delete_row statements that are associated with a tabledef that contains a descriptor modifier, use the descriptor to determine from which table the row is to be deleted. It shall be an error to call delete_row supplying a descriptor that does not have a memory image associated with it.

7.14 Built-in library functions

Default library functions are functions that the language automatically makes available without the need to include additional modules or header files. The names of these functions are not reserved words and shall be used as shown. No prototypes or definitions for these functions shall be present in the source as they are understood by the compiler. Prototypes shown in this subclause are for descriptive reasons only.

7.14.1 Numeric conversion functions

The following built-in functions discussed in this subclause convert numeric values.

7.14.1.1 floor

floor converts a double-precision floating point value to the nearest whole number whose value is less than or equal to the argument (Table 35).

Table 35—Library function floor

```
import calc(floor):
    passed(double: valueToBeConverted)
    result(double);
```

7.14.1.2 ifloor

ifloor converts a double-recision floating point value to the nearest integer value greater than or equal to the argument (Table 36).

Table 36—Library function ifloor

```
import calc(ifloor):
    passed(double: valueToBeConverted)
    result(int);
```

7.14.1.3 ceil

Ceil converts a double-precision floating point value to the nearest whole number whose value is greater than or equal to the argument (Table 37).

Table 37—Library function ceil

```
import calc(ceil):
    passed(double: valueToBeConverted)
    result(double);
```

7.14.1.4 iceil

iceil converts a double-precision floating point value to the nearest integer value greater than or equal to the argument (Table 38).

Table 38—Library Function iceil

```
import calc(iceil):
    passed(double: valueToBeConverted)
    result(int);
```

7.14.1.5 rint

rint rounds a double-precision floating point value to the nearest whole number (Table 39).

Table 39—Library function rint

```
import calc(rint):
    passed(double: valueToBeConverted)
    result(double);
```

7.14.1.6 round

round rounds a double-precision floating point value to the nearest integer value (Table 40).

Table 40—Library function round

```
import calc(round):
  passed(double: valueToBeConverted)
  result(int);
```

7.14.1.7 trunc

trunc converts a double-precision floating point value to a whole number by setting all the digits to the right of the decimal point to zero (Table 41).

Table 41—Library function trunc

```
import calc(trunc):
  passed(double: valueToBeConverted)
  result(double);
```

7.14.1.8 itrunc

trunc converts a double-precision floating point value to an integer value by first setting all the digits to the right of the decimal point to zero and then doing the converting (Table 42).

Table 42—Library function itrunc

```
import calc(itrunc):
  passed(double: valueToBeConverted)
  result(int);
```

7.14.2 Tech_family functions

Tech_family functions give access to the tech_family for the purposes of querying or to setting its value in a Standard Structure. These functions return results pertaining to the space the function was called on.

7.14.2.1 map_tech_family

map_tech_family creates a TECH_TYPE that is associated with the tech_family name of the argument. If the technology is not present, a value of nil is returned (Table 43).

Table 43—Library function map_tech_family

```
import calc(map_tech_family):
  passed(string: tech_family_name)
  result(TECH_TYPE: TT);
```

7.14.2.2 current_tech_type

current_tech_type returns the TECH_TYPE associated with the Standard Structure that has established the context (Table 44).

Table 44—Library function current_tech_type

```
import calc(current_tech_type):
  result(TECH_TYPE: TT);
```

7.14.2.3 subrule_tech_type

subrule_tech_type returns the TECH_TYPE associated with the subrule's TECH_FAMILY statement (Table 45).

Table 45—Library function subrule_tech_type

```
import calc(subrule_tech_type):  
    result(TECH_TYPE: TT);
```

7.14.2.4 is_expose_in_tech

is_expose_in_tech returns an integer value indicating whether the named expose is present in the specified tech_family. A value of one indicates the expose is present in the tech_family contained with the TECH_TYPE specified in the second argument (Table 46).

Table 46—Library function subrule_tech_type

```
import calc(subrule_tech_type):  
    passed(string: expose_name; TECH_TYPE: TT)  
    result(int);
```

7.14.2.5 get_technology_list

get_technology_list returns an array of TECH_TYPES where each element of the array is a TECH_TYPE corresponding to a specific tech_family present in the system of library modules (Table 47).

Table 47—Library function get_technology_list

```
import calc(get_technology_list):  
    result(TECH_TYPE[*]: TT);
```

7.14.3 Trigonometric functions

7.14.3.1 cos

cos performs the cosine function. The cos function accepts an argument in radians and converts that to the cosine value between -1 and 1 (Table 48).

Table 48—Library function cos

```
import calc(cos):  
    passed(double:radians)  
    result(double);
```

7.14.3.2 sin

sin function performs the sine function. The sin function accepts an argument in radians and converts that to the sine value between -1 and 1 (Table 49).

Table 49—Library function sin

```
import calc(sin):  
    passed(double:radians)  
    result(double);
```

7.14.3.3 tan

tan performs the tangent function. The tan function accepts an argument in radians and converts that to the tangent value between $-\infty$ and $+\infty$ (Table 50).

Table 50—Library function tan

```
import calc(tan):
    passed(double:radians)
    result(double);
```

7.14.4 Context manipulation functions

Context manipulation functions query, control, or modify the contexts being used.

7.14.4.1 new_plane

new_plane creates a new plane on the same space as that contained in the Standard Structure argument. The plane may be given a name by passing a string containing the desired name as the second argument. If the second argument value is nil, then the plane name is the number of the plane being created. The last argument controls whether errors and their associated messages should be issued during the plane creation process. If the last argument value is not zero, all errors shall be suppressed; otherwise, error and messages shall be issued (Table 51).

Table 51—Library function new_plane

```
import calc(new_plane):
    passed(STD_STRUCT: std; string: planeName; int: suppressErrors)
    result(var sync STD_STRUCT: std_struct);
```

7.14.4.2 get_plane_name

get_plane_name gets the name of the plane associated with the Standard Structure argument (Table 52).

Table 52—Library function get_plane_name

```
import calc(get_plane_name):
    passed(STD_STRUCT: std)
    result(string);
```

7.14.4.3 get_space_name

get_space_name returns the name of the space associated with the Standard Structure argument (Table 53).

Table 53—Library function get_space_name

```
import calc(get_space_name):
    passed(STD_STRUCT: std)
    result(string);
```

7.14.4.4 get_max_spaces

get_max_spaces returns the system's setting for the maximum number of spaces allowed (Table 54).

Table 54—Library function `get_max_spaces`

```
import calc(get_max_spaces):
    result(int);
```

7.14.4.5 `get_max_planes`

`get_max_planes` returns the system's setting for the maximum number of planes allowed per space (Table 55).

Table 55—Library function `get_max_planes`

```
import calc(get_max_planes):
    result(int);
```

7.14.4.6 `get_space_coordinate`

`get_space_coordinate` returns the integer that represents the current space index. Spaces are created with an integer index starting with zero for the first space created. As additional spaces are created, each one gets the next positive integer up to the maximum number of allowed spaces. The returned index represents the current space's position in the order of creation (Table 56).

Table 56—Library function `get_space_coordinate`

```
import calc(get_space_coordinate):
    result(int);
```

7.14.4.7 `get_plane_coordinate`

`get_plane_coordinate` returns the integer that represents the current plane index. Planes for each space are created with an integer index starting with zero for the first plane created on the space. As additional planes are created, each one gets the next positive integer up to the maximum number of allowed planes per space. The index represents the current plane's position in the order of creation (Table 57).

Table 57—Library function `get_plane_coordinate`

```
import calc(get_plane_coordinate):
    result(int);
```

7.14.4.8 `set_busy_wait`

`set_busy_wait` sets the number of tests the current queue shall perform when the work queue is empty before suspending operation. `set_busy_wait` returns the previous test count.

Each plane contains a queue of available work. When the current plane has finished a unit of work and the queue no longer contains any work items for processing, the planes suspend activities. When another plane places a unit of work to do for the current thread, the thread is reactivated and resumes processing the newly asserted unit of work. Busy wait controls the number of attempts the current thread shall make to determine if the queue has any newly asserted work before suspending operations. The initial value of busy wait is zero, indicating it shall do no additional attempts to check for newly asserted work before suspending. If a value greater than zero is asserted, then the current plane shall test the queue the asserted number of times before suspending. Values greater than zero create a busy wait whose duration is the time period required to do the asserted number of tests. This is useful when short periods of time may occur between units of work placed on the current plane's queue and the operating system's suspend and reinstatement periods are long enough compared to the duration of the actual work being performed to impact the overall throughput (Table 58).

Table 58—Library function set_busy_wait

```
import calc(set_busy_wait):
    passed(int: numberOfTests)
    result(int);
```

7.14.5 Debug controls

During the development of a library or program, the developer often needs additional information to determine where problems are. The following functions control the level of debug output and other available information.

7.14.5.1 change_debug_level

change_debug_level sets the debug level and returns the previous setting. change_debug_level allows the developer to change the debug level under library control. Its behavior is identical to that of dpcmChangeDebugLevel but called from within the library (Table 59).

Table 59—Library function change_debug_level

```
import calc(change_debug_level):
    passed(int: newDebugLevel)
    result(int);
```

7.14.5.2 get_caller_stack

When in debug mode, get_caller_stack returns as an array of strings the call stack of dcl functions only. Any C or C++ functions such as those used by the application or the run-time library are not returned (Table 60).

Table 60—Library function get_caller_stack

```
import calc(get_caller_stack):
    result(string[*]: callerStack);
```

7.14.6 Utility functions

Utility functions perform an eclectic set of functions that are not otherwise classified.

7.14.6.1 GET_LOAD_HISTORY

GET_LOAD_HISTORY returns the load histories one for each tech_family (Table 61).

Table 61—Library function GET_LOAD_HISTORY

```
import calc(GET_LOAD_HISTORY):
    result(LOAD_HISTORY_TYPE[*]: loadHistory; VOID : reserved);
```

7.14.6.1.1 loadHistory

loadHistory contains an array of LOAD_HISTORY_TYPES for each tech_family present in the requested space.

7.14.6.1.2 reserved

reserved is undefined and shall not be modified.

7.14.6.2 GET_CELL_LIST

GET_CELL_LIST returns the list of cells currently modeled for a tech_family (Table 62).

Table 62—Library function GET_CELL_LIST

```
import calc(GET_CELL_LIST):
    result(CELL_LIST_TYPE[*]: cellList; );
```

7.14.7 Table functions

Table functions allow low-level manipulation and querying of DCL tables.

7.14.7.1 GET_ROW_COUNT

GET_ROW_COUNT returns the number of rows not including the optional DEFAULT or the PROTOTYPE rows contained in a DCL table pointed to by its table descriptor (Table 63).

Table 63—Library function GET_ROW_COUNT

```
import calc(GET_ROW_COUNT):
    passed(TABLE_DESCRIPTOR: td)
    result(int);
```

7.14.7.2 STEP_TABLE

STEP_TABLE steps through a table by moving a cursor associated with each table. After each successful move, the data and qualifiers associated with the cursor's new position are returned.

When a table is searched, the cursor is set to the last row in the table successfully located (Table 64).

Table 64—Library function STEP_TABLE

```
import calc(STEP_TABLE):
    passed(TABLE_DESCRIPTOR: td;
           int: direction;
           string[*]: qualifiers;
           var abstract: dataRecord)
    result(int);
```

STEP_TABLE has the following four arguments: The first is the table descriptor, which identifies the table to be searched; the second is the direction to step the table; the third is an array of qualifiers that is filled in by step_table; and the fourth is the row's data record that is also filled in by step_table. Step_table first moves the cursor in the requested direction, fills in the passed arguments, and then returns an integer indicating whether the cursor has moved passed the end of the table. If either the qualifier array or the data record arguments have a value of nil, step_table shall allocate new space before filling them in.

The control of the cursor's movement is controlled by an argument that controls the direction. The direction values are controlled by the following predefined constants:

- STEP_TABLE_BACKWARDS: steps the cursor to the previous row in the table

- STEP_TABLE_CURRENT: steps the table to the last position successfully searched
- STEP_TABLE_END: steps the table to the last row in the table
- STEP_TABLE_FORWARDS: steps the table to the next row in the table
- STEP_TABLE_START: steps the table to the first row in the table excluding the default record
- STEP_TABLE_TO_DEFAULT_RECORD: steps the table cursor to the default record

7.14.8 Subrule controls

The following functions discussed in this subclause control or query the state of a subrule.

7.14.8.1 GET_LOAD_PATH

GET_LOAD_PATH returns to the path the directory where current subrule was loaded from (Table 65).

Table 65—Library function GET_LOAD_PATH

```
import calc(GET_LOAD_PATH):  
    result(string);
```

7.14.8.2 GET_RULE_NAME

GET_RULE_NAME returns the module name of the current subrule (Table 66).

Table 66—Library function GET_RULE_NAME

```
import calc(GET_RULE_NAME):  
    result(string);
```

7.14.8.3 ADD_RULE

ADD_RULE loads another subrule module. The modules loaded are peers to the existing subrules. A subrule may load another subrule associated with the same or different tech_families. ADD_RULE returns an integer where a value of zero indicates a successful load of the subrule and its associated components (Table 67).

Table 67—Library function ADD_RULE

```
import calc(ADD_RULE):  
    passed(string: ruleName, rulePath, tablePath, controlParm)  
    result(int);
```

7.14.8.3.1 ruleName

ruleName is the name of the module that is the subrule to be loaded.

7.14.8.3.2 rulePath

rulePath is the name of the environment variable containing the search path for the subrule to be loaded.

7.14.8.3.3 tablePath

tablePath is the name of the environment variable containing the search path for any tables the subrule to be loaded might need.

7.14.8.3.4 controlParm

controlParm is the control parameter the subrule to be loaded shall be given.

7.15 Library control statements

This subclause provides an overview, as well as the purpose, syntax, description, examples, and restrictions for use of the DCL library control statement classes and their components.

Library control statements control the logical organization and loading of subrules and identify where these subrules shall be found. A DPCM can be made up of several subrules. The application loads the first (root) subrule. The DPCM shall automatically load any additional subrules necessary to make up the complete system. The subrule statement controls the loading of one additional subrule per statement. The subrule statement identifies a file that contains a list of subrules to be loaded.

A library developer may organize a collection of subrules into a technology family with the TECH_FAMILY statement. The TECH_FAMILY statement allows the same PI to be implemented for multiple technologies.

Library control statements shall not be referenced by other statements. Any statement in a subrule with the name LATENT_EXPRESSION or TERMINATE_EXPRESSION is specially recognized and evaluated by the DPCM.

7.15.1 Meta-variables

Library control statements control the setting of library meta-variables. Meta-variables are variables that are set by the run-time linker when the DPCM is loaded and remain in effect while the DPCM is loaded.

7.15.2 TECH_FAMILY

The TECH_FAMILY meta-variable is set by the TECH_FAMILY statement that begins each subrule. This variable can be used within a PATH environment variable (UNIX) or user variable (Windows NT) to alter the search location.

7.15.3 RULENAME

The RULENAME meta-variable is set to the name of each subrule loaded. This variable may be used within a PATH environment variable (UNIX) or user variable (Windows NT) to alter the search location.

7.15.4 CONTROL_PARM

The CONTROL_PARM meta-variable may be set by a SUBRULE statement or the control file associated with the SUBRULES statement. There shall be one CONTROL_PARM meta-variable for each subrule loaded. If the statement does not specify a CONTROL_PARM, its value shall be the empty string. The CONTROL_PARM meta-variable may be used within the DCL source code by referencing the predefined identifier CONTROL_PARM.

7.15.5 SUBRULE statement

The SUBRULE statement shall accept zero or one of each of the following clauses:

- RULE_PATH
- TABLE_PATH
- CONTROL_PARM

The syntax for the SUBRULE statement is given in Syntax 7.76.

```
subrule_statement ::= SUBRULE ( name ) [OPTIONAL] : {rule_or_table_path} ;
rule_or_table_path ::= RULE_PATH ( string_literal )
| TABLE_PATH ( string_literal )
| CONTROL_PARM ( string_literal )
```

Syntax 7.76—subrule_statement

SUBRULE statements allow one subrule to load another subrule. Subrules are not referenced; rather they are loaded in the order they appear within the DCL source. Nested subrules are loaded in depth first order. Subrule statements shall complete the loading of the specified subrule before the next subrule or subrules statement in the current file is executed.

The SUBRULE statement name shall be the name of the file to be loaded.

The optional RULE_PATH and the TABLE_PATH clauses can be used to control where the technology library loading subsystem searches for the subrules.

7.15.5.1 OPTIONAL modifier

By default, if a subrule is not found, a fatal load error shall occur. However, if the OPTIONAL post-fix modifier is specified and the subrule cannot be found, normal execution shall continue. The OPTIONAL modifier shall not suppress errors if the subrule itself encounters errors while being loaded.

7.15.5.2 RULE_PATH clause

The optional RULE_PATH clause shall take one argument, a string constant, which designates an operating system environment variable (UNIX) or user variable (Windows NT) containing the path list for the subrule.

This variable shall contain a colon delimited list of file system directory names. The technology library loading subsystem shall search each subdirectory in the path list, in order, for a file name matching the SUBRULE name. It shall attempt to load as a subrule the first file (see 7.15.6) it encounters with that name.

By default the current working directory shall be searched if this clause is not present.

7.15.5.3 TABLE_PATH clause

The optional TABLE_PATH clause shall take one argument, a string constant, which designates an operating system environment variable (UNIX) or user variable (Windows NT) containing the path list for locating compiled tables associated with the subrule.

This variable shall contain a colon delimited list of file system directory names. The technology library loading subsystem shall search each subdirectory in the path list, in order, for compiled tables used by the subrule (see 7.15.6).

By default the current working directory shall be searched if this clause is not present.

7.15.6 Path list expansion rules

If the following strings are encountered in the TABLE_PATH and RULE_PATH environment or user variables, they are replaced as follows:

- %RULENAME
Replaced with the *subrule_name* being loaded. On operating systems that do not allow the % to exist in a path (i.e., Windows NT), the expansion variable is ?RULENAME.
- %TECH_FAMILY
Replaced with the *tech_family_name* of the subrule performing the subrule load operation. On operating systems that do not allow the % to exist in a path (i.e., Windows NT), the expansion variable is ?TECH_FAMILY.
- %CONTROL_PARM
Replaced with the CONTROL_PARM value obtained from either the SUBRULE statement's CONTROL_PARM clause or the control file's CONTROL_PARM field associated with the subrule load operation. On operating the systems that do not allow the % to exist in a path (i.e., Windows NT), the expansions variable is ?CONTROL_PARM.

7.15.7 SUBRULES statement

The syntax for the SUBRULES statement is given in Syntax 7.77.

```
subrules_statement ::= SUBRULES ( name ) : {file_or_path};
file_or_path ::= [FILE ( string_literal ) ]
               | [FILE_PATH ( string_literal ) ]
```

Syntax 7.77—subrules_statement

The SUBRULES statement shall instruct the rule loading subsystem that a separate ASCII file contains additional instructions for loading subrules.

7.15.7.1 FILE_PATH clause

The FILE_PATH clause shall identify the environment variable (UNIX) or user variable (Windows NT) that contains the colon delimited list of paths to search (in order) for the file named in the FILE clause. If that search fails, the current working directory shall be searched last. If the FILE clause is omitted, the FILE_PATH environment or user variable shall contain the file name as well as the path.

7.15.7.2 FILE clause

The FILE clause shall contain the name of the file that contains the instructions to control subrule loading (see 7.15.8). If the FILE_PATH clause is omitted, the FILE clause shall contain the file name, as well as the path.

7.15.8 Control file

The control file is an ASCII file consisting of a list of directives that instruct the rule loading subsystem which subrules are required. Subrules shall be loaded in the order encountered in the file.

7.15.8.1 Directives

Each directive in the control file shall be contained on a single line (record). Each record in the file may be a comment record, a default record, or a load record, as follows:

- Comment records
Shall begin with a # symbol or a // symbol starting in the first character position of the line. The remainder of the line may be used as comment text.

- Default records
Shall begin with the word “default” starting in the first character position of the line. The default record does not load any subrules but sets the default value for any field in a load record (except the rule name) that is omitted. There may be as many default records as required. Subsequent default records shall override field values if already set by previous ones.
- Load records
Shall load subrules, according to the rules defined in 7.15.8.3 and 7.15.8.4.

7.15.8.2 Default record fields

If no default value for a given field has been defined in a default record, the following are the predefined default values used:

- Rulename
This is the name of the environment variable (UNIX) or user variable (Windows NT) that shall contain the path for the subrule. DCMRULEPATH shall be used to locate the rule to be loaded.
- Tablename
This is name of the environment or user variable that shall contain the path for the compiled tables associated with the subrule. DCMTABLEPATH shall be used to locate the tables associated with this subrule.
- Optional
If this field has any of the characters, y, Y, or 1, then loading the subrule shall be optional. If any other non-blank character (except *) is used, loading shall be mandatory. By default, subrule loading is mandatory, and if the load of the subrule or any of its compiled tables fails, an error shall be generated. If the subrule is optional and is not found, the system shall continue and no messages shall be issued. However, if the subrule is optional, and is found, but generates a loading error (either on the subrule itself or as a result of associated tables), the load shall be terminated and the error shall be reported to the application.
- Control parameter
If this field contains a string, then the meta-variable CONTROL_PARM shall be set to the value of this string. The control parameter string shall not contain embedded white space.

It shall be an error to attempt to load the same subrule more than once. Subrules within the same system shall come from a unique combination of source file and TECH_FAMILY names.

7.15.8.3 Load and default record fields

Both the default record and the load record shall consist of the following five fields on one line, each separated by at least one white space: The default record fields are preceded by the keyword default:

- File name
The first field shall contain the file name of the rule to be loaded.
- Rulename
This is the name of the environment variable (UNIX) or user variable (Windows NT) that shall contain the path for the subrule.
- Table name
This is name of the environment or user variable that shall contain the path for the compiled tables associated with the subrule.
- Optional
If this field has any of the characters, y, Y, or 1, then loading the subrule shall be optional. If any other non-blank character (except *) is used, loading shall be mandatory. By default, subrule

loading is mandatory and if the load of the subrule or any of its compiled tables fails, an error shall be generated. If the subrule is optional and is not found, the system shall continue and no messages shall be issued. However, if the subrule is optional, and is found, but generates a loading error (either on the subrule itself or as a result of associated tables), the load shall be terminated and the error shall be reported to the application.

— Control parameter

If this field contains a string, then the meta-variable CONTROL_PARM shall be set to the value of this string.

7.15.8.4 Using a default value in the load or default record

A record may indicate selection of a default value (from a default record) for a particular field in the following two ways:

- a) Use the * default operator.
- b) Leave the field blank. However, since the fields are free-format, with blanks used as the separators, this can only be done for trailing fields after the last non-blank field.

7.15.9 TECH_FAMILY statement

The syntax for the TECH_FAMILY statement is given in Syntax 7.78.

```
tech_family_statement ::= TECH_FAMILY ( APP | name ) [MAIN] ;
```

Syntax 7.78—tech_family_statement

DCL allows the library developer to organize a collection of subrules into a technology family. Through the use of the TECH_FAMILY statement, each subrule associated with a particular technology has its activities coordinated based on issues related to that technology.

Subrules separated into families of technologies provide access to the application through an identical set of statements, cell names, and so on. The application can work with a consistent interface for all technologies regardless of the number of TECH_FAMILY(s) loaded, while still being able to distinguish which parts of their design are associated with a specific technology.

Multiple technology definitions can be loaded simultaneously. With this capability, technology definitions can be designed independent of each other and each technology can cooperate with the other as required.

Technology families can represent entire chips or major portions of a chip; generally, these are units of manufacturing by a single manufacturer and not individual library elements. The TECH_FAMILY statement allows subrules from different vendors to work together, even though no information was exchanged between the organizations.

NOTE—A run-time cost is associated with changing between TECH_FAMILYs. It is therefore recommended that the use of TECH_FAMILY subrule grouping be limited to entire technologies.

7.15.9.1 TECH_FAMILY name

Subrules are identified as belonging to the same family by including the TECH_FAMILY statement, whose argument shall be the family name.

Any subrules that do not contain the TECH_FAMILY statement shall be members of the GENERIC technology. The family name can be any legal identifier. It shall be unique among the other

TECH_FAMILY names. The TECH_FAMILY name does not have to be unique to other types of statement names in the other technology families.

7.15.9.2 MAIN option

Modules compiled with a tech_family containing a MAIN option shall be program module that is executable. Modules with the MAIN option shall perform the normal loading and linking options such as load subrules associated with any SUBRULE or SUBRULES statements contained within the module. After the modules and tables are loaded, the LATENT_EXPRESSIONS are executed and then the expose named MAIN shall be executed if one exists.

7.15.10 SUBRULE and SUBRULES statements

A SUBRULE (see 7.15.5) or SUBRULES (see 7.15.7) statement may only load other subrules that are of the same TECH_FAMILY name, with the following exceptions:

- A technology subrule with the name GENERIC can load any other subrule, generic or technology-specific. In this case, the loaded subrule shall retain its technology-specific characteristics.
- A technology-specific subrule may load a GENERIC subrule. In this case, the GENERIC subrule shall inherit the name of the technology-specific subrule that loaded it.

7.16 Modeling

DCL defines a flexible approach to modeling cells based on a graph topology that represents the desired properties of a cell. A MODELPROC contains a series of statements used to describe the topology to the application. The application is informed of the topology through a series of callbacks.

Locating the proper modelproc for a cell is done by mapping the cell to a specific modelproc. The MODEL statement defines those cells that are described by a MODELPROC.

For improved flexibility, commonly used groups of modeling statements can be gathered into a SUBMODEL that can be called from either a MODELPROC or a SUBMODEL.

7.16.1 Types of modeling

MODELPROC (“model procedure”) statements describe the actions of a circuit with respect to timing, power, vector power, vector timing, or function.

7.16.1.1 Timing

Modeling for static timing uses the modeling capabilities to generate a graph that represents the cell’s static timing behavior. This graph consists of internal points called “nodes” and arcs called “paths,” “buses,” or “tests.” Each arc and node carries with it additional information to assist the application and the library to perform the task of static timing.

For example, the additional information could include the direction of the edges, formulas to use for the analysis of delay or slew, as well as additional data needed for these calculations.

7.16.1.2 Function modeling

The example in Figure 2 illustrates a typical transformation of a four-input AND-OR cell to an equivalent function graph form.

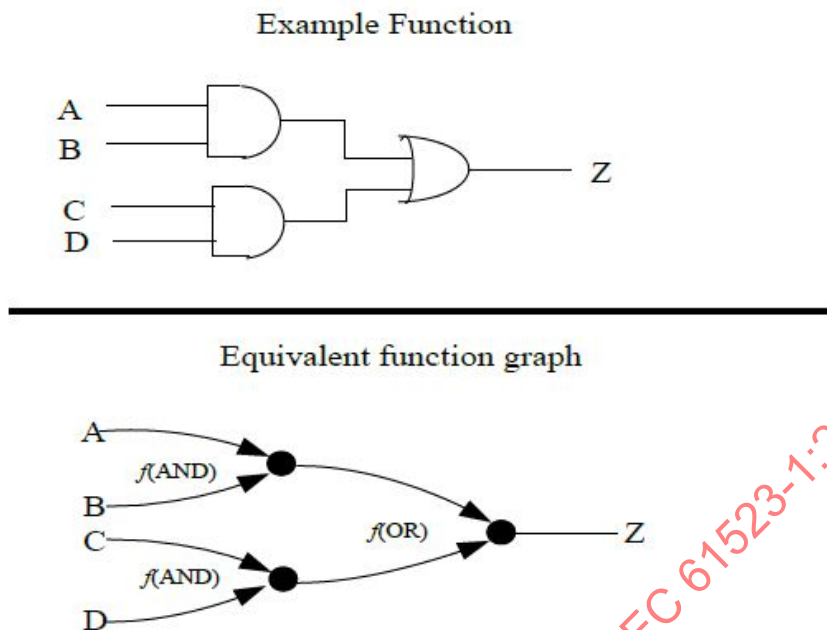


Figure 2—Function graph form

As shown in Figure 2, each node shall have a function operation (AND, OR , etc.). In addition, the arcs shall be defined as a particular data type. During the modeling or elaboration phase, the library shall inform the application, via implicit callback APIs, of the structure of the functional model under investigation. Library cells that model the function of a cell or cells shall use a domain named behavior. DCL model domains are classes of library descriptions that have common properties such as timing or function.

7.16.1.3 Vector power and vector timing modeling

Vector power and vector time use a graphical form similar to static timing. The primary difference is in the additional information carried for the arcs and notes.

Vector-timing, vector-power, and timing-state graphs use graph topologies similar to that of the function graph, except the nodes in these graph use different operators that apply to the vector timing, vector power, and timing states.

For example, a model representing the function of a cell uses a graph where each node or output pin represents a Boolean or data flow operation. Each arc in the graph represents the interconnection of the operators, the data types, and their widths and pin orientations. Vector timing and power graphs are similar to those representing function except each node or output pin in the graph represents a state transition.

7.16.2 Model organization

This subclause details the MODEL statement and model name matching.

7.16.2.1 MODEL statement

The syntax for the MODEL statement is given in Syntax 7.79.

```

model_statement ::= MODEL ( model_name ) :
    DEFINES ( cell_list ) ;
cell_list ::= cell_descriptor { , cell_descriptor }
cell_descriptor ::= cell_name
    [ . cell_qualifier [ . model_domain ] ]
cell_qualifier ::= name | *
model_domain ::= * | timing | power

```

Syntax 7.79—tech_family_statement

MODEL statements define which cells are described by a MODELPROC. Each MODEL statement shall have a corresponding MODELPROC statement of the same name. The MODEL statement shall lexically precede the corresponding MODELPROC statement.

A *cell_descriptor* shall be an ordered list of one to three fields, corresponding to the CELL, CELL_QUAL, and MODEL_DOMAIN components of a fully qualified model. The first field shall be CELL, the second CELL_QUAL, and the third MODEL_DOMAIN. Omitted (trailing) fields shall be treated as * (asterisk).

The MODEL_DOMAIN values have the following semantics:

- * Only one model exists for power and/or timing.
- Timing is the timing model only.
- Power is the power model only.

7.16.2.1.1 Model name matching

The search for a model proceeds through the three components (CELL, CELL_QUAL, and MODEL_DOMAIN) in order. For each component, let component *app* be the field value supplied by the application, and let component *DPCM* be the field as present in the DPCM.

It shall be an error for any component *app* to use the string prefix operator or be a literal *.

If the application does not have a specific value for a component *app*, it shall use the empty string ("") for that component *app*.

For each component, the following precedence rules apply to name matching:

- If an exactly-matching component DPCM exists, match it.
- If a match exists with a component DPCM that uses the string prefix operator, match it.
- If a component DPCM exists that consists of *, match it.

7.16.3 MODELPROC statement

The syntax for the MODELPROC statement is given in Syntax 7.80.

```

model_procedure ::= MODELPROC ( name ) modelproc_postfix_modifier :
    modelproc_statement_list END ;
modelproc_postfix_modifier ::= COMPLEX | MONOLITHIC
modelproc_statement_list ::=
    {setvar_statement} properties_statement | {submodel_statement}
submodel_statement ::= do_statement
    | bus_statement
    | path_statement
    | input_statement
    | output_statement
    | test_statement
    | test_bus_statement
    | setvar_statement
    | path_separator_stmt

```

Syntax 7.80—model_procedure

A **MODELPROC** statement (“model procedure”) describes the actions of a circuit with respect to timing and/or power.

7.16.3.1 MODELPROC flow of control

MODELPROC statements, statement references, and embedded “C” code contained within a MODELPROC shall be executed in lexical order, except as follows:

- Statement references identified as arguments to the delay, slew, or check statements within the delay, slew, or check clauses shall be evaluated when the application calls for delay, slew, or check, respectively. SETVAR variables shall not be referenced as arguments to these statements.
- Statement references identified as arguments to action statements registered to a method within a METHODS clause shall be evaluated at the time of the method call. SETVAR variables shall not be referenced as arguments to these statements.
- Statement sequences in WHEN / OTHERWISE clauses shall be skipped if the controlling logical expression evaluates to false.
- The application shall provide a list of PIN(s) present in the design.
- PATH, BUS, and TEST statements shall skip any PIN pairs not found.
- INPUT and OUTPUT statements shall skip all PIN(s) not found.

7.16.3.2 MONOLITHIC modifier

The **MONOLITHIC** modifier directs the DCL compiler to process the MODELPROC as a single compilation unit.

NOTE—A DCL compiler can choose to handle a MODELPROC as several separately compiled units. However, if the MODELPROC contained embedded C code with labels and GOTO statements, it may be necessary to force the MODELPROC to be processed as a single compilation unit.

7.16.4 SUBMODEL statement

Submodel statements represent a portion of a cell model. A submodel is a collection of modeling statements that can be accessed from other functions or models within the library. The submodel statement may accept arguments and return results. The submodel statement may be exported for use by more than one library module. Similar to other calculation statements, the submodel statement may be used to define a type (Syntax 7.81).

```

submodel_procedure ::= [EXPORT] SUBMODEL ( name ) :
    [passed_clause] [result_prototype] [key_clause]
    [consistent_clause] [using_clause] [submodel_statement_list]
    END [ ( result_name = expression
    { ; result_name = expression } ) ] ;
key_clause ::= KEY
    ( passed_argument_name { , passed_argument_name } )
consistent_clause ::= CONSISTENT
    ( passed_argument_name { , passed_argument_name } )
using_clause ::= USING ( using_expression_list )
using_expression_list ::=
    using_predefined_variable = parameter_name
using_predefined_variable ::=
    input_pins | output_pins | nodes

```

Syntax 7.81—submodel_procedure

A SUBMODEL shall not contain a PROPERTIES statement.

Submodels can be called from calculation statements, MODELPROCs, or SUBMODELs. When a SUBMODEL is called, it attempts to inherit the current MODELPROC's environment if one is still in scope. When a SUBMODEL is called and no MODELPROC environment is in scope, it creates an environment as if it were a MODELPROC. The MODELPROC environment includes the input pins list, output pins list, anyin list, anyout list, and nodes passed in the Standard Structure and created during the current call to model.

7.16.4.1 Model consistency information

To reduce the work required to model a cell, both modelprocs and submodels keep consistency information available to determine if a store operation has been previously performed and if that store information can be reused. A store operation is the process of caching information of calculations associated with a cell, a pin, or an arc. Consistent information is information that does not change with different instances of the same cell. If the information is consistent, the cost of performing the information lookup or the calculation can be eliminated by reusing the previous store operation.

To make the reuse of consistent information possible, the models as they execute keep track of what cell, cell_qual, domain, submodel, statement, path, from point, to point, pin, and logical condition a consistent store was performed. Repeated consistent stores are eliminated by examining if the store under these same conditions has already occurred, and if so, they reuse the previous results.

7.16.4.2 The PASSED clause

The SUBMODEL's PASSED clause follows the syntax and semantics of PASSED clauses in other DCL statements.

7.16.4.3 The RESULT clause

The SUBMODEL's RESULT clause follows the syntax and semantics of the DCL EXTERNAL statement's RESULT clause.

7.16.4.4 KEY clause

The KEY clause identifies the passed arguments that are to be added to the consistency information tracked. The passed arguments identified by the KEY clause shall be either string or integer types.

7.16.4.5 USING clause

By default, modelprocs and submodels assume the list of inputs, outputs, and nodes are contained in the Standard Structure. The USING clause allows submodels to be passed arguments that contain the list of inputs, outputs, and nodes to be used, and they are treated as though they were contained in the Standard Structure. This keeps in tack modeling constructs such as ANYIN or ANYOUT, even though these lists were derived from passed arguments instead of being created from the Standard Structure. The arguments identified in the using clause shall be arrays of pins. The array of pins assigned to INPUT_PINS represents the list of inputs to the submodel, the array of pins assigned to OUTPUT_PINS represents the list of output pins, and the array of pins assigned to the NODES represent the list of internal points. If any of these variables are not identified in the USING clause, the list defaults to those identified in the Standard Structure.

7.16.4.6 CONSISTENT clause

Passed arguments to submodels are assumed to be inconsistent. The consistent clause identifies those arguments that shall be treated as consistent.

7.16.4.7 The END clause

The SUBMODEL's END clause contains an semicolon-delimited (;) list of assignment expressions that follows the syntax and semantics of a DCL EXTERNAL statement's DEFAULT clause. The expression list is only present when the RESULT clause is present.

7.16.5 Modeling statements

This subclause lists the modeling statements in DCL.

7.16.5.1 PATH_SEPARATOR statement

The syntax for the PATH_SEPARATOR statement is given in Syntax 7.82.

```
path_separator_stmt ::= PATH_SEPARATOR ( string_literal ) ;
```

Syntax 7.82—path_separator_stmt

The *PATH_SEPARATOR* statement defines a string that can disambiguate segment names generated by the default operator * used in the PATH clause of the PATH statement. The PATH_SEPARATOR string is inserted between the concatenation of the FROM pin name and the TO pin name. This constructed string names the timing segment and can be accessed by the PATH predefined variable.

The definition of a PATH_SEPARATOR string shall extend from the PATH_SEPARATOR statement until the next PATH_SEPARATOR statement or until the end of the enclosing subrule (whichever occurs lexically first). Initially within a subrule, the PATH_SEPARATOR string shall be the empty string. The PATH_SEPARATOR is a compiler directive and has no run-time effect.

7.16.5.2 PATH statement

The syntax for the PATH statement is given in Syntax 7.83.

```

path_statement ::= PATH ( path_list ) : from_to_sequence
conditional_propagation_sequence ;
from_to_sequence ::= FROM ( pins ) TO ( pins )
pins ::= pin_range_list
| VAR ( pin_assign_variable_reference )
| VAR ( pinlist_assign_variable_reference )
| VAR ( pin_setvar_variable_reference )
| VAR ( pinlist_setvar_variable_reference )
| pin_statement_reference
| pin_list_statement_reference
conditional_propagation_sequence ::= propagation_sequence
| when_propagation
| data_type_sequence
propagation_sequence ::= PROPAGATE
( edge_mode_expression )
pre_code [delay_slew_methods_store_list] post_code
| data_type_sequence
delay_slew_methods_store_list ::= delay_slew_methods_store {
delay_slew_methods_store }
delay_slew_methods_store ::=
DELAY ( name_of_delay_stmt ( parameter_list ) )
| SLEW ( name_of_slew_stmt ( parameter_list ) )
| store_clause
| methods_clause
| clkflg_clause
| ckttype_clause
| objtype_clause
when_propagation ::= when_propagation_list [ , OTHERWISE
propagation_sequence ]
when_propagation_list ::=
WHEN ( logical_expression ) propagation_sequence
{ , WHEN ( logical_expression ) propagation_sequence }
edge_mode_expression ::= edge_mode_operation
{ ; edge_mode_operation }
edge_mode_operation ::= edge mode edge
edge ::= RISE | FALL | BOTH | TERM | ONE_TO_Z | ZERO_TO_Z
| Z_TO_ZERO | Z_TO_ONE
mode ::= -> | <- | <-> | <-X-> | ->X<-

```

Syntax 7.83—path_statement

There shall be either zero or one of each of delay, slew, or methods clauses present in a *delay_slew_methods_store_list*. There shall be zero or more store clauses present in the *delay_slew_methods_store_list*.

The **PATH** statement establishes an association (a segment) between two connection points, each of which may be an input pin, output pin, or internal timing point (node). The **PATH** statement associates the following with each segment:

- Statements to use for computing delay and slew values
- Properties, such as the signal edges that propagate across the segment
- Propagation mode
- Information cached with the **STORE** clause
- Method names and their associated action statements

For each explicitly named path in the PATH clause, the PATH statement establishes a segment between every pin specified in the FROM clause to every pin specified in the TO clause.

7.16.5.2.1 VAR clause

The *VAR* clause indicates that the pin(s) described by either the FROM or TO clauses are specified by the value of a SETVAR or ASSIGN statement result variable having data type PIN or PINLIST.

7.16.5.2.2 Path list

The syntax for the path_list statement is given in Syntax 7.84.

```
path_list ::= [default_path_list] | path_name_list
default_path_list ::= *
path_name_list ::= name { , name }
```

Syntax 7.84—path_list

For each name in the PATH clause (but at least once, even if no name is specified), a segment shall be constructed during model elaboration between each pair of specified endpoints. The PATH variable in the Standard Structure is set to the name of the current segment. The name of the segment shall be determined based on the following rules:

- If no path name is specified, for example, PATH(), the segment shall have the empty string ("") as its name.
- If a single path name is specified, for example, PATH(A), the segment shall be given that name.
- If the default operator * is specified, for example, PATH(*), then the segment's name shall be constructed by concatenating the name of the FROM pin, the lexically most recent PATH_SEPARATOR string, and the name of the TO pin.
- If multiple path names are specified, for example, PATH(A, B), then a separate segment shall be created for each name.

7.16.5.2.3 FROM clause

The *FROM* clause identifies the timing points where the PROPAGATE segments begin. These points can be an input pin, output pin, or a node. These pins may be specified directly in the DCL source, or they may be returned by a statement at run time. The order of search when a pin listed in the FROM clause is identified as the beginning of a segment shall be the list of application-supplied input pins, then nodes, and then output pins. Only those pins listed in the FROM clause and found in the application's supplied pin lists or the list of nodes shall be considered valid segment starting points. All pins listed in the FROM clause but not found shall be ignored. The application's lists of supplied pins are searched input pins first, nodes second, and output pins last.

7.16.5.2.4 TO clause

The *TO* clause identifies the timing points where the PROPAGATE segments end. These points can be an input pin, output pin, or a node. These pins may be specified directly in the DCL source, or they may be returned by a statement at run time. The order of search when a pin listed in the TO clause is identified as the end of a segment shall be the list of application-supplied output pins, then nodes, and then input pins. Only those pins listed in the TO clause and found in the application's supplied pin lists or the list of nodes shall be considered valid segment ending points. All pins listed in the TO clause but not found shall be ignored. The application's lists of supplied pins are searched output pins first, nodes second, and input pins last.

7.16.5.2.5 PROPAGATION sequence

The *PROPAGATION sequence* describes which signal edges at the source of the segment shall be propagated to the load (sink) of the segment. It can also include the following clauses:

— Delay clause

The delay clause associates a delay statement and the arguments that shall be passed to it with a segment. The arguments identified as parameters to the delay statement in the delay clause shall be evaluated when the application calls for delay calculation.

— Slew clause

The slew clause associates a slew statement and the arguments that shall be passed to it with a segment. The arguments identified as parameters to the slew statement in the slew clause shall be evaluated when the application calls for slew calculation.

— METHODS clause

The syntax for the METHODS clause is given in 7.16.7.2.

— STORE clause

The syntax for the STORE clause is given in 7.16.7.3.

— CLKFLG clause

The syntax for the CLKFLG clause is given in Syntax 7.85.

```
clkflg_clause ::= CLKFLG ( string_literal )
```

Syntax 7.85—clkflg_clause

The CLKFLG clause identifies segments where the clock performs memory operations.

The following strings have meaning in the context of a CLKFLG argument:

- 1) X shall be used on the clock segment where a clock combines with data to form a latching operation. Typically this X flag is specified where the clock is to be converted to data.
- 2) R shall be used where the rising edge of the clock causes the dynamic circuit to evaluate.
- 3) F shall be used where the falling edge of the clock causes the dynamic circuit to evaluate.

The string literal value shall be accessible by the application when the PATH_DATA field is not NIL. The default setting in the Standard Structure if this clause is omitted shall be the string consisting of a single blank ("").

— CKTTYPE clause

The syntax for the CKTTYPE clause is given in Syntax 7.86.

```
ckttype_clause ::= CKTTYPE ( string_literal )
```

Syntax 7.86—ckttype_clause

The ckttype clause enables identification of the class of circuit a path belongs to. The string_literal value shall be accessible by the application when the PATH_DATA field in the Standard Structure is not NIL. This standard does not define any special values for string_literal. The default setting if this clause is omitted shall be a blank (" ").

OBJTYPE for retain modeling is given in Syntax 7.87.

```

object_type_clause ::= OBJECT_TYPE ( string_literal )
string_literal ::= 'retain'

```

Syntax 7.87—object_type_clause

7.16.5.2.6 Data type sequence

The arcs that span function nodes carry the data the terminating node needs. The arcs also define the data's properties such as type, bit width, and positional orientation. The PATH and BUS statements have two new clauses that describe these properties. These clauses allow the identification of the data type, which is traversing from one logical operation to another, and the bit stranding of this data. The syntax is presented in Syntax 7.83 and Syntax 7.88.

```

data_type_sequence ::= DATA_TYPE ( data_type_enumeration_exp )
[ SOURCE_STRANDS ( bit_value -- bit_value )
[ SINK_STRANDS ( bit_value -- bit_value ) ] ]
| DATA_TYPE ( data_type_enumeration_exp )
[ ROUTE ( bit_value -- bit_value ) ]
data_type_enumeration_exp ::= integer_expression
| BIT | INT | UNSIGNED INT | CHARACTER
| UNSIGNED CHARACTER | LONG | UNSIGNED LONG | FLOAT
| DOUBLE | EXCESS64 | EXCESS128
bit_value ::= scalar | function_call
| VAR ( variable_reference )

```

Syntax 7.88—data_type_sequence

7.16.5.2.7 DATA_TYPE clause

Arcs represent the connections between any combination of input pins, output pins, or internal pins. Arcs can represent one or more logical connections between each node. Each arc in a function or vector representation shall have associated with it properties denoting what type of data and, if it is a bus, the strand range it represents. The data type clause (Table 68) defines the default bit configuration settings for the standard data types supported.

Table 68—Data type clause

Data type	Default bit layout
BIT	Unsigned array of bits, where the width is controlled by the strand fields.
CHARACTER	Signed array of 8 bits (0:7), where the most significant bit (0) is the sign. The bit ordering can be overridden by the strand clauses.
INTEGER	Signed array of 32 bits (0:31), where the most significant bit (0) is the sign. The bit ordering can be overridden by the strand clauses.
LONG	Signed array of 64 bits (0:63), where the most significant bit (0) is the sign. The bit ordering can be overridden by the strand clauses.
UNSIGNED CHARACTER	Unsigned array of 8 bits (0:7), where the most significant bit is bit (0). The bit ordering can be overridden by the strand clauses.
UNSIGNED INTEGER	Unsigned array of 32 bits (0:31), where the most significant bit is bit (0). The bit ordering can be overridden by the strand clauses.
UNSIGNED LONG	Unsigned array of 64 bits (0:63), where the most significant bit is bit (0). The bit ordering can be overridden by the strand clauses.
FLOAT	32 bit IEEE floating point notation bits (0:31), where bits (9:31) represent the fraction, bits (1:8) represent the exponent, and bit (0) represents the sign.
DOUBLE	64 bit IEEE floating point notation bits (0:63), where bits (12:63) represent the fraction, bits (1:11) represent the exponent, and bit (0) represents the sign.

The arc data type (Table 69) defines the enumeration values associated with each type being represented by the arc.

Table 69—Arc data types

Data type	Enumeration value
When the terminating node is a bit or word operator node	
BIT	0x0000
When the terminating node is a bit operator or a node	
Propagate previous type	0xFFFF
CHARACTER	0x0001
INTEGER (Array of 32 Bits)	0x0002
LONG	0x0004
UNSIGNED CHARACTER	0x1001
UNSIGNED INTEGER (Array of 32 Bits)	0x1002
UNSIGNED LONG	0x1004
When the terminating node is a floating point operation	
FLOAT (IEEE floating point)	0x0008
DOUBLE (IEEE double-precision floating point)	0x0010
EXCESS64 (Excess 64 single precision floating point)	0x0020
EXCESS128 (Excess 128 double-precision floating point)	0x0040
Precedence control radiating from a precedence node	
PRECEDENCE scope start	0x8000
PRECEDENCE else list	0xC000
PRECEDENCE action list	0xE000
When the terminating node is a user-defined function	
User-defined	0x0000 - 0xFFFF

7.16.5.2.8 SOURCE_STRANDS clause

The SOURCE_STRANDS clause contains two integer expressions indicating a range of strands representing the least significant bit at the source end and the most significant bit at the source end (inclusive). These two integers are separated by a range delimiter (--). The value on the left represents the least significant bit, and the value on the right represents the most significant bit.

Example

SOURCE_STRANDS(63--0) indicates bit 63 is the least significant bit and bit 0 is the most significant bit.

7.16.5.2.9 SINK_STRANDS clause

The SINK_STRANDS clause contains two integer expressions indicating a range of strands representing the least significant bit at the sink end and the most significant bit at the sink end (inclusive). The arguments to the SINK_STRANDS clause have the same semantic meaning as SOURCE_STRANDS.

7.16.5.2.10 ROUTE clause

The ROUTE clause contains the same information as the SOURCE_STRANDS clause and is a short-hand notation to indicate the source and sink have the identical strand range values.

7.16.5.3 BUS statement

The syntax for the BUS statement is given in Syntax 7.89.

```
bus_statement ::= BUS ( path_list ) :  
    FROM ( pin_range_list ) TO ( pin_range_list  
    ) conditional_propagation_sequence ;
```

Syntax 7.89—bus_statement

The **BUS** statement has the same syntax and semantics as the **PATH** statement, with the following differences:

- a) The **PATH** statement shall construct a fully connected graph between the **FROM** *pin_range* and the **TO** *pin_range*; that is, the **PATH** statement shall establish a segment between every pin specified and present in the **FROM** clause to every pin specified and present in the **TO** clause.
- b) The **BUS** statement shall construct a parallel graph between the **FROM** *pin_range* and the **TO** *pin_range*; that is, the **BUS** statement shall establish a segment between the lexically first pin identified and present in the **FROM** clause and the lexically first pin identified and present in the **TO** clause. If more pins are present, the **BUS** statement shall establish a segment between the lexically next pin identified and present in the **FROM** clause and the lexically next pin identified and present in the **TO** clause.
- c) **ANYIN** and **ANYOUT** shall not be allowed in the Pin Range list.
- d) The pin count in the **FROM** clause shall match the pin count in the **TO** clause.
- e) The number of pins in the design that match the **FROM** pin list shall match and be paired with the number of pins in the **TO** pin list.

7.16.5.4 TEST statement

The syntax for the **TEST** statement is given in Syntax 7.90.

```
test_statement ::= TEST ( path_list ) :  
    conditional_compare ;  
conditional_compare ::= pre_compare_post | when_compare_list [ ,  
    OTHERWISE pre_compare_post ]  
pre_compare_post ::= pre_code compare_list post_code  
when_compare_list ::= WHEN ( logical_expression ) pre_compare_post { ,  
    WHEN ( logical_expression ) pre_compare_post }
```

Syntax 7.90—test_statement

The **TEST** statement inserts test points into a design. The statement describes the types of tests to be performed, and the application does the tests. The passed sub_list serve the same purpose as path names in 7.16.5.2.

7.16.5.4.1 Compare_list

The syntax for the *compare_list* clause is given in Syntax 7.91.

```

compare_list ::= [compare_clause] [edges_clause] [compare_sequence_list]
compare_sequence ::= [test_type_clause]
                  | [checks_clause]
                  | [clkflg_clause]
                  | [ampersand_store_clause]
                  | [ampersand_methods_clause]
                  | [ckdtype_clause]
                  | [cycleadj_clause]

```

Syntax 7.91—compare_list

There shall be one *test_type* and *checks* clause for each *compare_list*. There may be zero or one of *clkflg*, *ckdtype*, and *cycleadj* clauses for each *compare_list*.

The number of *compare_lists* in the *compare_clause* shall be the same as the number of *compare_edges_lists* in the *edges_clause*. Corresponding elements of these two lists in the ordinal position shall have the same mode.

7.16.5.4.2 COMPARE clause

The syntax for the COMPARE clause is given in Syntax 7.92.

```

compare_clause ::= COMPARE ( multi_compare_pin_list )
multi_compare_pin_list ::= compare_pin_list
                        { ; compare_pin_list }
compare_pin_list ::= reference_signal_pin test_mode reference_signal_pin
reference_signal_pin ::= pin_range_list
                    | REFERENCE ( pin_range_list )
                    | SIGNAL ( pin_range_list )
test_mode ::= -> | <- | <->

```

Syntax 7.92—compare_clause

The *COMPARE clause* defines which pins are to be tested, which are the references, which are the signals, and in what modes are they to be tested. This clause allows the specification of both early and late mode tests in the same clause.

Only those pins listed in the COMPARE clause and found in the application's supplied pin lists or the list of nodes shall be considered. All pins listed in the COMPARE clause but not found shall be ignored.

If no signal or reference is used, the pins on the left of the mode shall represent the reference pins and those on the right shall represent the signal pins.

7.16.5.4.3 EDGES clause

The syntax for the EDGES clause is given in Syntax 7.93.

```
edges_clause ::= EDGES ( multi_compare_edge_list )
multi_compare_edge_list ::= compare_edges_list
    { ; compare_edges_list }
compare_edges_list ::= reference_signal_edge mode reference_signal_edge
reference_signal_edge ::= edge
    | REFERENCE ( edge )
    | SIGNAL ( edge )
```

Syntax 7.93—edges_clause

The *EDGES* clause identifies the edges to be tested for both the signal and reference pins.

If no signal or reference is used, the edges on the left of the mode shall represent the reference edges and those on the right shall represent the signal edges.

7.16.5.4.4 TEST_TYPE clause

The syntax for the TEST_TYPE clause is given in Syntax 7.94.

```
test_type_clause ::= TEST_TYPE ( test_type_sequence_list )
test_type_sequence_list ::= test_type_sequence
    { ; test_type_sequence }
test_type_sequence ::= test_type | test_type dual_mode test_type
test_type ::= SETUP | HOLD | CPW | CST | DHT | DPW | DST | CGPW | CGHT |
    CGST | NOCHANGE | RECOVERY | REMOVAL | SKEW
dual_mode ::= <->
```

Syntax 7.94—test_type_clause

The *TEST_TYPE* clause indicates to the application the type of test to be performed using the following techniques:

- The TEST_TYPE argument expression designates one or two test types depending on the mode operator of the corresponding *compare_pin_list*. The *test_types* to the left of the *dual_modes* operator shall indicate the *test_type* to be used for late mode tests, and the *test_types* to the right of the *dual_modes* operator shall indicate the *test_type* to be used for early mode tests. In cases where the corresponding *compare_pin_list* uses no multimode operators, *test_type* shall be the same as the corresponding type of the *compare_pin_list*.

- CYCLEADJ clause.

The syntax for the CYCLEADJ clause is given in Syntax 7.95.

```
cycleadj_clause ::= CYCLEADJ ( integer_expression )
```

Syntax 7.95—cycleadj_clause

The *cycleadj* clause enables identification of special multicycle paths. The *integer_expression* value shall be accessible by the application when the PATH_DATA field in the *Standard Structure* is not NIL. This standard does not define any special values for *integer_expression*. The default setting if this clause is omitted shall be zero.

7.16.5.4.5 CHECKS clause

The syntax for the CHECKS clause is given in Syntax 7.96.

```
checks_clause ::= CHECKS ( checks_sequence_list )
checks_sequence_list ::= checks_sequence
    { ; checks_sequence }
checks_sequence ::= check_statement_name
    ( expression_list )
    | check_statement_name ( expression_list )
    dual_mode check_statement_name ( expression_list )
```

Syntax 7.96—checks_clause

The **CHECKS** clause identifies the CHECK statement(s) to use when determining the allowable offsets (bias) between the edge of a signal and the edge of a reference.

The CHECKS argument expression designates one or two CHECKS statements depending on the mode operator of the corresponding *compare_pin_list*. If the corresponding *compare_pin_list* uses only one mode, early or late, but not any of the combined modes, the *checks_sequence* shall not contain a *dual_modes* operator. If the corresponding checks clause contains a multiple mode operator, then the checks clause shall contain a *dual_modes* operator. The check statement left of the dual mode operator shall be the check statement identified for late mode calculations, and the check statement on the right of the dual mode operator shall be the check statement used for early mode calculations. In clauses where there is no *dual_modes* operator, the check statement referenced shall be for the mode of the corresponding compare clause. Each CHECKS statement reference shall include all required PASSED parameters.

7.16.5.4.6 METHODS clause

The syntax for the **METHODS** clause is given in Syntax 7.97.

```
methods_list ::= METHODS ( methods_action_lists )
methods_action_lists ::= methods_action_stmt_list
    { ; methods_action_stmt_list }
```

Syntax 7.97—methods_list

The **METHODS** clause registers action statements to the specified test.

There shall be a one to one correspondence between the number of *method_action_stmt_list* (s) and the number of *compare_pin_list* (s).

7.16.5.4.7 STORE clause

The syntax for the **STORE** clause is given in Syntax 7.98.

```
store_clause ::= STORE ( store_list )
store_list ::= store_cache_list { ; store_cache_list }
```

Syntax 7.98—store_clause

The **STORE** clause caches statement results associated with the specified test(s).

There shall be a one-to-one correspondence between the number of *store_cache_list* (s) and the number of *compare_pin_list* (s). Each *store_cache_list* shall contain the store caches for its equivalent lexically positioned *compare_pin_list*.

7.16.6 TEST_BUS statement

The **TEST_BUS** syntax is given in Syntax 7.99.


```
test_bus_statement ::= TEST_BUS ( path_list ) : conditional_compare ;
```

Syntax 7.99—test_bus_statement

The **TEST_BUS** statement has the same syntax and semantics as the **TEST** statement, with the following differences:

- The **TEST** statement shall construct a fully connected test arc sequence between the **COMPARE** reference *pin_range* and the **COMPARE** signal *pin_range*; that is, the **TEST** statement shall establish a segment between every pin specified and present in the **COMPARE** clause reference sequence to every pin specified and present in the **COMPARE** clause signal sequence.
- The **TEST_BUS** statement shall construct a parallel graph between the *reference pin_range* and the signal *pin_range*; that is, the **TEST_BUS** statement shall establish a segment between the lexically first pin identified and present in the reference sequence and the lexically first pin identified and present in the signal sequence. If more pins are present, the **TEST_BUS** statement shall establish a segment between the lexically next pin identified and present in the reference sequence and the lexically next pin identified and present in the signal sequence.
- **ANYIN** and **ANYOUT** shall not be allowed in the Pin Range list.
- The pin count in the reference sequence shall match the pin count in the signal sequence.
- The number of pins in the design that match the reference pin list shall match and be paired with the number of pins in the signal pin list.

7.16.7 INPUT statement

The syntax for the **INPUT** statement is given in Syntax 7.100.

```
input_statement ::= INPUT ( pin_range_list )
    [ : opt_conditional_propagation_sequence ] ;
opt_conditional_propagation_sequence ::= propagation_sequence |
    when_opt_propagation
when_opt_propagation ::= when_opt_propagation_list
    [ , OTHERWISE opt_propagation_sequence ]
when_opt_propagation_list ::=
    WHEN ( logical_expression ) opt_propagation_sequence
    { , WHEN ( logical_expression ) opt_propagation_sequence }
opt_propagation_sequence ::=
    pre_code methods_store_cache_list post_code
    | propagation_sequence
methods_store_cache_list ::= store_clause methods_clause
    | methods_clause store_clause
```

Syntax 7.100—input_statement

The **INPUT** statement models nets, caches information and associates action statements with methods that relate to input pins. If the **INPUT** statement includes a *propagation_sequence*, the application shall connect timing segments between all sources and the specified load pin.

The *pin_range* list designates the list of pins to which subsequent propagation clauses apply. If the *pin_range* is not **ANYIN**, then the listed pins shall be excluded from any subsequent expansion of **ANYIN** in the same **MODELPROC** (by any statement or any submodel called). Only those pins listed in the *input_clause* and found in application's lists of supplied pins or list of node shall be considered. All pins listed in the *input_clause* and not found shall be ignored. The application's lists of supplied pins are searched input pins first, nodes second and output pins last.

NOTE—There can be double creation of net segments if cell descriptions in a technology library contain both INPUT and OUTPUT statements.

7.16.7.1 Propagation clause

DCL does not allow control over an individual net segment, but rather it applies the same actions (specified in the relevant clauses) to all net segments attached to a pin.

Example

If the delay equation *netdly* is specified for a pin, all net segments connected to that pin shall use the delay equation *netdly* to calculate the wire delay.

7.16.7.2 METHODS clause

The syntax for the *METHODS* clause is given in Syntax 7.101.

```
methods_clause ::= METHODS ( methods_action_stmt_list )
methods_action_stmt_list ::= method_action_pair
    { , method_action_pair }
method_action_pair ::= method_name : statement_name (
    comma_expression_list ] )
```

Syntax 7.101—methods_clause

The *METHODS* clause establishes an association of three parts: a segment, a method name, and an action statement. It shall be an error to associate more than one action statement with the same method name and segment.

The *action statements* associated with a method may require passed parameters. If so, any such parameters shall be supplied as part of the *METHODS* clause in the parameter list of the statement. The values of such parameters shall be determined at the time of the method reference.

7.16.7.3 STORE clause

The syntax for the *STORE* clause is given in Syntax 7.102.

```
store_clause ::= STORE ( store_cache_list )
    [ : KEY ( [key_expression_list] ) ]
store_cache_list ::= store_statement_access
    | slot_definition
    | store_cache_list , store_statement_access
    | store_cache_list , slot_definition
store_statement_access ::= statement_name ( expression_list )
slot_definition ::= statement_name [scalar_list] : ( slot_list )
scalar_list ::= scalar { , scalar }
slot_list ::= [scalar_list] statement_name ( expression_list )
    { , [scalar_list] statement_name ( expression_list ) }
key_expression_list ::= key_expression { , key_expression }
key_expression ::= key_target = string_expression
key_target ::= PATH | TO_POINT | FROM_POINT | PIN
```

Syntax 7.102—store_clause

The *STORE* clause describes information to be calculated and cached at model elaboration time (presumably because that information is independent of which instance references the model).

One may wish to cache the results of accessing the same statement more than once with varying arguments. To effect this capability, the STORE cache may be declared as an arbitrary dimension array, each element of which (a *slot*) contains the result of accessing the statement.

During model elaboration, each STORE shall explicitly specify the array indices of the slot to contain the result. The syntax identifies the name of the statement being accessed, the number of slots to allocate, and the number of dimensions for the slot array, and for each statement access, the slot into which the result shall be stored. Slot array indices shall start with 0 in each dimension. All slots need not be filled. During expression evaluation, each STORE reference shall explicitly specify the array indices of the slot whose result is to be used.

The RESULT clause of the statement in the *slot_list* definition shall match the RESULT clause of the statement in the *slot_definition*.

Example

```
tabledef(coeffcient):
  passed(string:sourceEdge, sinkEdge)
  qualifiers(cell, from_point, to_point, sourceEdge, sinkEdge)
  data(number: k);

model (AND2P) : defines (AND2P);

modelproc (AND2P) :
  path(*) :
    from (A, B) to (Z)
    propagate (rise <-> rise & fall <-> fall)
    delay (stdDlyEq())
    slew (stdSlwEq())
    store(coeffTbl[2]: (
      [0]coeffTbl(rise,rise),
      [1]coeffTbl(fall,fall)
    ));
end;
calc (example) : result(number: [0]coeffTbl.k + [1]coeffTbl.k);
```

The CALC statement references a complete copy of the actual computed results. References to a TABLEDEF statement only cache the pointer to the table row containing the data. Referencing non-slotted stored data is syntactically identical to referencing an ASSIGN statement variable. Statements that reference STORE variables shall have the STORE definition statement in scope. For slotted stored data a reference is pre-appended with an array_index.

A STORE clause can reference CALC, EXPOSE, EXTERNAL, INTERNAL, and TABLEDEF statements. A STORE clause shall not reference an ASSIGN statement.

A STORE clause can use predefined identifiers. See 7.2.3.4 to identify the information to be saved. The validity of predefined identifiers for the STORE table enumerates which predefined identifiers are valid, based on the type of statement containing the **STORE** clause (Table 70).

Table 70—Validity of predefined identifiers for STORE clause

Predefined identifier	Model statement type
BLOCK	all
CELL	all
CELL_DATA	all but PROPERTIES
CELL_QUAL	all
CLKFLG	INPUT, OUTPUT, PATH, BUS, TEST
COMPILATION_TIME_STAMP	all
FROM_POINT	OUTPUT, PATH, BUS
INPUT_PIN_COUNT	all
MODEL_DOMAIN	all
MODEL_NAME	all
NODE_COUNT	all
OUTPUT_PIN_COUNT	all
PATH	PATH, BUS, TEST
PATH_DATA	INPUT, OUTPUT, PATH, BUS, TEST
REFERENCE_POINT	TEST
SIGNAL_POINT	TEST
TO_POINT	INPUT, PATH, BUS

A variable referenced in STORE clauses shall have a defined value at model elaboration time.

Example

The predefined identifiers `EARLY_MODE`, `LATE_MODE`, `SOURCE_EDGE`, and `SINK_EDGE` shall not be referenced in the statements contained in a STORE clause because at model elaboration time, these variables are undefined.

7.16.7.4 KEY store modifier

The KEY modifier overrides the store clause's default value for the consistency keys with the value expression. The keys that can be overridden are `PATH`, `FROM_POINT`, `TO_POINT`, and `PIN`. `PATH`, `FROM_POINT`, and `TO_POINT` are valid on `TEST`, `TEST_BUS`, `PATH`, and `BUS` statements. `PIN` is only valid on `DO` statements `NODE`, `CONDITIONAL_NODE`, and `PIN` sequences.

7.16.8 OUTPUT statement

The syntax for the *OUTPUT* statement is given in Syntax 7.103.

```
output_statement ::= OUTPUT ( pin_range_list )
                  [ : opt_conditional_propagation_sequence ] ;
```

Syntax 7.103—output_statement

The *OUTPUT* statement controls actions that involve output pins. If the *OUTPUT* statement contains a *propagation_sequence*, the application to connect timing segments between the specified load pin and all sinks. Only those pins listed in the *output_clause* and found in the application's lists of supplied pin list or the list of nodes shall be considered. All pins listed in the *output_clause* but not found shall be ignored. The application's lists of supplied pins are searched output pins first, nodes second and input pins last.

The *OUTPUT* statement shall not set the value of the predefined identifier `PATH`.

The *pin_range_list* expression (see 7.8.13.1) designates the list of pins to which subsequent propagation clauses apply. If the *pin_range_list* is not ANYOUT, then the listed pins shall be excluded from any subsequent expansion of ANYOUT in the same MODELPROC (by any statement or any submodel called).

NOTE—There can be double creation of net segments if the technology library contains descriptions of both the INPUT and OUTPUT statements.

7.16.8.1 METHODS clause

The syntax for the METHODS clause is given in 7.16.5.4.6.

7.16.8.2 STORE clause

The syntax for the STORE clause is given in 7.16.5.4.7.

7.16.9 DO statement

The *DO* statement has two classes of operations that it may perform. The *DO* statement controls the scope within a *MODELPROC* or *SUBMODEL*, and it can create internal points within a model called “nodes.”

7.16.9.1 DO statement scope

DO statements create and nest scope. Each scope can contain decision logic, reference variables, or statements, and it can contain sequences of other statements. Each nested scope that is created is contained within its parent scope and uses standard scoping rules. That is to say, the nested scope can access the variables created by the parent, but the parent cannot access the variables created by the nested scope.

7.16.9.2 DO statement nodes

The application analyzing a design has visibility to all the cells, nets, and pins contained within the design. Before calling the library to model a cell, the application has no visibility to any arcs or nodes that are used to model the cell. The nodes that are created in the *DO* statement identify these internal points to the application so it can reference them as it does detailed analysis of the design.

7.16.9.3 Looping constructs

This specification extends the DCL looping constructs within *MODELPROC*s and *SUBMODEL*s to include REPEAT, FOR, WHILE, BREAK, and CONTINUE.

7.16.9.4 FOR loops

FOR loops consist of the following components:

- The initialization, which allows the loop to set the starting values of variables used for loop control.
- The logical expression, which determines whether the loop continues or terminates. When the logical expression evaluates true, the loop continues to the next iteration.
- The loop variable processing, which adjusts the loop variable each iteration.
- The *do_clause_list*, which consists of clauses following the FOR clause. The *do_clause_list* represents the section of program that is executed each iteration.

When a FOR loop is entered, the loop initialization clause is executed and then the logical expression is evaluated. If the logical expression evaluates true, the *do_clause_list* is executed followed by the loop

variable processing clause. The process repeats, with the exclusion of the loop initialization clause, until the logical expression evaluates `false`, at which point the loop terminates.

7.16.9.5 WHILE loops

When a WHILE loop is encountered, it evaluates the logical expression. If that logical expression evaluates `true`, it executes the associated `do_clause_list`. When the `do_clause_list` processing completes, the control returns to the WHILE's logical expression. If the logical expression evaluates to `true`, the process repeats; otherwise, the loop exits.

7.16.9.6 REPEAT loops

When a REPEAT loop is encountered, the evaluation of the loop body begins. When the loop body execution has completed, the UNTIL logical expression is evaluated. If the UNTIL logical expression evaluates to `false`, the loop body is again executed; otherwise, the loop terminates.

7.16.9.7 BREAK processing

BREAK terminates the execution of a loop at points other than the loop's logical expression evaluation. When a BREAK is encountered, the innermost DO statement containing the loop is immediately exited.

7.16.9.8 CONTINUE processing

CONTINUE skips the remaining portion of a loop body and begins execution at the next iteration. When a CONTINUE is encountered, the flow of control immediately resumes at the logical expression evaluation of the innermost DO statement containing the loop (Syntax 7.104).

```
do_statement ::= { DO : do_clause_list } ;
| DO : do_when_sequence
[ , OTHERWISE {do_clause_list} [BREAK | CONTINUE] ;
| DO : do_while_sequence ;
| DO : do_repeat_sequence ;
| DO : do_for_sequence ;

do_clause_list ::= call_clause | pre_code | statements_clause
| node_clause | function_sequence | vector_sequence

do_when_sequence ::= WHEN ( logical_expression ) {do_clause_list} [BREAK
| CONTINUE]
{ , WHEN ( logical_expression ) {do_clause_list} [BREAK | CONTINUE]
}

do_while_sequence ::= WHILE ( logical_expression ) {do_clause_list}
do_repeat_sequence ::= REPEAT {do_clause_list}
UNTIL ( logical_expression )
do_for_sequence ::= FOR ( [ ( loop_initialization ) ] ,
[ ( logical_expression ) ] ,
[ ( loop_variable_adjustment ) ] ) {do_clause_list}
```

Syntax 7.104—do_statement – BREAK and CONTINUE

The syntax for the DO statement is given in Syntax 7.105.

```

do_statement ::= { DO : do_clause_list } ;
| DO : do_when_sequence
[ , OTHERWISE {do_clause_list} ] [BREAK | CONTINUE] ;
| DO : do_while_sequence ;
| DO : do_repeat_sequence ;
| DO : do_for_sequence ;
do_clause_list ::= call_clause
| pre_code
| statements_clause
| node_clause
| function_sequence
| vector_sequence
do_when_sequence ::= WHEN ( logical_expression ) {do_clause_list} [BREAK
| CONTINUE]
{ , WHEN ( logical_expression ) {do_clause_list}
[BREAK | CONTINUE] }
do_while_sequence ::= WHILE ( logical_expression ) {do_clause_list}
do_repeat_sequence ::= REPEAT {do_clause_list}
UNTIL ( logical_expression )
do_for_sequence ::= FOR ( ( [loop_initialization] ) , (
logical_expression ) ,
( [loop_variable_adjustment] ) ) {do_clause_list}

```

Syntax 7.105—do_statement

The DO statement enables the use of conditional clauses. Within these conditional clauses, the following can be specified:

- Calls to submodel procedures
- Nested modelproc statements
- New timing points

A DO statement may contain zero or more occurrences of a NODE clause (see 7.16.9.11.1), atomic statement reference, embedded C code, statement references, and/or do statement scope.

7.16.9.9 Statement reference

The *CALL* clause allows the DO statements to invoke a SUBMODEL procedure statement (see Syntax 7.105). The syntax for the CALL clause is given in Syntax 7.106.

```

statement_reference ::= statement_name ( parameter_list )

```

Syntax 7.106—statement_reference

7.16.9.10 DO statement brace scope

The *DO* statement scope allows the use of modelproc statements with the exception of the *PROPERTIES* statement, which in turn allows nesting of WHEN clauses. The *statement_reference* syntax is presented in Syntax 7.107.

```

do_statement_scope ::= {modelproc_statement_list}

```

Syntax 7.107—statement_reference

7.16.9.11 Node_sequence grammar

To simplify the description of complex cells, the use of internal points or nodes is often useful. A *node* is a point in the graph used for static timing or it contains the behavioral operation to be performed, as shown in Syntax 7.108. Nodes can be created using a **NODE**, a **COND_NODE**, or the **PIN** clause within a **DO** statement.

```

node_sequence ::= NODE ( name { , name } ) [import_export_sequence]
               [primitive_sequence]
               | COND_NODE ( (name | VAR ( pin_assign_variable_reference ))
               [ REPLACE pin_variable_reference ] ) [primitive_sequence]
               | PIN (( name | VAR ( string_or_pin_reference ))
               REPLACE pin_variable_reference ) [primitive_sequence]
primitive_sequence ::= PRIMITIVE ( primitive_operator )
                   [ MODIFIERS ( modifier_enumeration_exp ) ]
                   [ CKTTYPE ( string_literal ) ]
primitive_operator ::= integer_expression | function_operator_set
function_operator_set ::= unary_operators | diadiac_operators
                      | triadic_operators | other_node_types
unary_operators ::= ! | -|+ | +|- | ++ | -- | ~ | -|?
                 | +|? | ?|- | ?|+ | ?|? | ?|~ | ?|! | ?|*
dyadic_operators ::= > | <= | < | >= | << | >> | + - | *
                 | / | % | || | !|| | && | !&& | ^^ | !^^ | ~ | ~~ | ~&
                 | ~^ | ~^~ | ~^& | ~^> | ~^< | ~^<>
triadic_operator ::= ?: | @ | @:: | @-> | :
new_predefined_variables ::= PRIMITIVE | MODIFIERS
                          | DCM_NEG_DYNAMIC_LATCH | DCM_NEG_PRECHARGE_NODE
                          | DCM_POS_DYNAMIC_LATCH | DCM_POS_PRECHARGE_NODE
modifier_enumeration_exp ::= integer_expression |
                           new_predefined_variables

```

Syntax 7.108—node_sequence

7.16.9.11.1 NODE clause

The nodes created by a **NODE** clause are visible to the **MODELPROC** and **SUBMODEL** statements encountered in the elaboration of a cell's description. The **NODE** clause unconditionally creates a node. The node created by the **NODE** clause is placed on a list of nodes that can be accessed by its name (which is the argument to the **NODE** clause). When using the **NODE** clause, it shall be an error to create more than one node with the same name within the same model sequence.

7.16.9.11.2 COND_NODE clause

The **COND_NODE** clause creates a node if and only if that node does not already exist. This allows for the reuse of constant nodes. The node created, if any, is placed on the list of nodes that can be accessed by name.

7.16.9.11.3 PIN clause

Nodes are named to allow them to be referenced by name in other modeling statements within a **MODELPROC** or **SUBMODEL**. **PIN**(s) that are referenced by a variable can have duplicate names. Use the **PIN** clause to associate a node with a variable (instead of its name).

The **PIN** clause requests the application construct a named node, but the **PIN** clause keeps the pin held privately so name conflicts do not occur. The use of a **PIN** clause in the **DO** statement requires setting a variable to the value of the node created by the application; the **REPLACE** sequence should be used, where the target is a **VAR** variable of type **PIN** that accepts the application's handle.

7.16.9.11.4 REPLACE operator

In some situations, reusing an existing node is useful. Having direct access to these nodes can eliminate redundancies, reduce the access time, and simplify the program.

The REPLACE operator overwrites the value of a variable with the pin or node found in a FROM, TO, NODE, PIN, COND_NODE, OUTPUT, or INPUT clause. The operand target (on the left) of the REPLACE operator shall be of the VAR variable of type PIN.

7.16.9.11.5 PRIMITIVE clause

The PRIMITIVE clause takes a primitive operator or an integer valued expression, which represents the desired function at this node, as its argument. See Table 71 through Table 77. Table 81 defines the function operators to primitive enumeration values in DCL.

Table 71—Logic operators
(valid for behavior, vectorTiming, and vectorPower model domains)

DCL operator	Enumeration value	Description
><	0x0000	No logic functions performed, but merging of strands and buses occurs with this type of operation. If used in conjunction with address operators and @, this node becomes the bulk storage node.
!	0x0001	NOT, (!A)
	0x0002	OR, (A B)
!	0x0003	NOR, !(A B)
&&	0x0004	AND, (A && B)
! &&	0x0005	NAND, !(A && B)
^^	0x0006	XOR, ((!A && B) (A && !B))
! ^^	0x0007	XNOR, !((!A && B) (A && !B))

Table 72—Logical equivalence operators (valid for behavior model domain)

DCL operator	Enumeration value	Description
>	0x0010	Greater than, (A > B)
<=	0x0011	Less than or equal, (A <= B)
<	0x0012	Less than, (A < B)
>=	0x0013	Greater than or equal, (A >= B)
==	0x0014	Equal, (A == B)
!=	0x0015	Not equal, (A != B)

Table 73—Unary bitwise operators
(valid for behavior, vectorTiming, and vectorPower model domains)

DCL operator	Enumeration value	Description
~	0x0009	Bitwise inversion of 0110 produces 1001.

**Table 74—Binary bitwise operators
(valid for behavior, vectorTiming, and vectorPower model domains)**

DCL operator	Enumeration value	Description
`	0x000A	0101 & 0011 (bitwise OR) produces 0111.
`~	0x000B	0101 & 0011 (bitwise NOR) produces 1000.
`&	0x000C	0101 & 0011 (bitwise AND) produces 0001.
`~&	0x000D	0101 & 0011 (bitwise NAND) produces 1110.
`^	0x000E	0101 & 0011 (bitwise XOR) produces 0110.
`~^	0x000F	0101 & 0011 (bitwise XNOR) produces 1001.

Table 75—Binary operators (valid for behavior model domain)

DCL operator	Enumeration value	Description
`<	0x0020	Shift left, (A << B)
`>	0x0021	Shift right, (A >> B)
+	0x0022	Addition, (A + B)
-	0x0023	Subtraction, (A - B)
*	0x0024	Multiplication, (A * B)
/	0x0025	Division, (A / B)
%	0x0026	Remainder, (A % B)

Table 76—Node primitives for control operators (valid for behavior model domain)

DCL operator	Enumeration value	Description
? : (Selector)	0x0028	Implements the selection function, where one set of inputs (control) determines which one of another set of inputs (data) is propagated to the output. There are always two control data inputs.
@? : (Priority mux)	0x0029	Implements a selection function where a single data input is selected to propagate based on the condition of a set of control inputs. The selection is based on priority order. There are (control + 1) data inputs. If none of the control inputs are true, the last data input is propagated.
@ : : (Priority storage node)	0x002A	Remembers the last successful state change. A state change is controlled by a selection process. The selection process is similar to the priority mux. There are always the same number of control inputs as data inputs. If none of the control inputs are, true no state change occurs.

Table 77—Node primitives for edge operators
(valid for behavior, vectorTiming, and vectorPower model domains)

DCL operator	Enumeration value	Description
− +	0x0030	Transitions from zero to one.
+ −	0x0031	Transitions from one to zero.
+ +	0x0032	Steady one.
− −	0x0033	Steady zero.
− ?	0x0034	Transitions from zero or remains constant.
+ ?	0x0035	Transitions from one or remains constant.
? −	0x0036	Transitions from any arbitrary value to zero, including the remaining constant.
? +	0x0037	Transitions from any arbitrary value to one, including the remaining constant.
? ?	0x0038	Transitions from any arbitrary value to another, including the possibility of remaining constant.
? *	0x0039	No transition allowed.
? !	0x003A	Arbitrary transition, excluding the possibility of a constant.
? ~	0x003B	Arbitrary transition, where all bits shall toggle.
−>	0x003C	Left occurs before the right.
<−>	0x003D	Listed elements can occur in any order, excluding the possibility of simultaneously occurring events.
&>	0x003E	Left occurs either before, or at the same time as, the right.
<&>	0x003F	Listed elements can occur in any order, including the possibility of simultaneously switching.
* +	0x0043	Arbitrary transition to one.
* −	0x0044	Arbitrary transition to zero.
+ *	0x0045	Arbitrary transition from one.
− *	0x0046	Arbitrary transition from zero.
+ ^	0x0047	Transition from logical one to high-impedance state.
− ^	0x0048	Transition from logical zero to high-impedance state.
^ +	0x0049	Transition from high impedance state to logical one.
^ −	0x004A	Transition from high impedance state to logical zero.
~>	0x004B	Left occurs before right, with the possibility of other edges in between.
* ?	0x004C	Several arbitrary signal transitions, including possibility of constant value, with arbitrary final value.
? =	0x004D	Arbitrary steady state, followed by arbitrary transitions.

For the operators listed in Table 78, @ designates a lower priority storage, whereas @ designates a higher priority storage. See Table 79 through Table 81.

Table 78—Node primitives for precedence control operators
(valid for behavior model domain)

DCL operator	Enumeration value	Description
@	0x0040	Precedence start
:	0x0041	Precedence else
=	0x0050	Blocking assignment

**Table 79—Node primitives for constant operators
(valid for behavior, vectorTiming, vectorPower model domains)**

DCL operator	Enumeration value	Description
=	0x0051	Constant value
?	0x0052	Unknown value
^	0x0053	High impedance value

Table 80—Node primitives for user-defined operators

Primitive operator (Enumeration Value)	Description
USER_DEFINED_MACRO (0xFFFE)	The user has defined a macro and its details are at the next lower level of hierarchy. The name of the macro can be found by examining the <code>cktType</code> field of the Standard Structure during the call to <code>newTimingPin()</code> .
USER_DEFINED_LOGIC_FUNCTION (0xFFFF)	User-defined operator. The user is obligated to supply a match service in the library, which can be called by the application.

Table 81—Node primitives for miscellaneous operators

Primitive operator (Enumeration Value)	Description
[@] (0x0054)	Address selection reserve for the primitive clause. The standard index operators [Boolean expression] are used in the function clause (see Table 87).
DCM_PRIMITIVE_VECTOR_DELAY_TARGET (0x0090)	Target of a vector Boolean expression.
DCM_PRIMITIVE_VECTOR_CHECK_TARGET (0x0091)	Target of a vector Boolean expression.
DCM_PRIMITIVE_VECTOR_POWER_TARGET (0x0092)	Target of a vector Boolean expression.
DCM_POSITIVE_DYNAMIC_LATCH (0x0060)	A latch that remembers its state through the storage of charge. The latch can change state only when the clock is high.
DCM_NEGATIVE_DYNAMIC_LATCH (0x0061)	A latch that remembers its state through the storage of charge. The latch can change state only when the clock is low.
DCM_POSITIVE_PRECHARGE_NODE (0x0062)	Logic devices that precharge during a portion of the clock cycle where no logical decisions are allowed. During the other half of the cycle, if the logical function evaluates true, the charge is discharged; otherwise, it is left in a charged state. Each positive pre-Scharge node precharges its output on the positive portion of the clock and evaluates on the negative portion of the clock.
DCM_NEGATIVE_PRECHARGE_NODE (0x0063)	Logic devices that precharge during a portion of the clock cycle where no logical decisions are allowed. During the other half of the cycle, if the logical function evaluates true, the charge is discharged; otherwise it is left in a charged state. A negative precharge node precharges its output on a negative portion of the clock and evaluates on the positive portion of the clock.
DCM_HIGH_Z_NODE (0x0064)	High impedance drive point.

Table 82 defines the inputs and outputs for the binary reduction operators. Table 90 defines the inputs and outputs for a given node.

Table 82—Binary reduction operators

Node function	Input definition	Output definition	Description
><	Any number of inputs of any type.	One output of the same values as entered the node.	Used to collect or disperse strand groups.
NOT	Any number of inputs of type: INTEGER, UNSIGNED INTEGER, LONG, UNSIGNED LONG, CHARACTER, UNSIGNED CHARACTER, or BIT.	One output of type BIT.	Logical inversion. The result value is zero if any of the input bits has a value of one.
OR			Evaluates to true if any of the bits in the left arc or the right arc has the value of one.
NOR			Evaluates to true if no bits from the left or right arcs have the value of one.
AND			Evaluates to true if all the bits have a value of one.
NAND			Evaluates to true if any of the bits has a value of zero.
XOR			Evaluates to true if at least one bit is set and an odd number of bits have a value of one.
XNOR	Any number of inputs of type: INTEGER, UNSIGNED INTEGER, LONG, UNSIGNED LONG, CHARACTER, UNSIGNED CHARACTER, or BIT.	One output of type BIT.	Evaluates to true if no bits are set or an even number of bits have a value of one.

Table 83 defines the inputs and outputs for the bitwise reduction operators.

Table 83—Bitwise reduction operators

Node function	Input definition	Output definition	Description
Bitwise NOT	One input, any type and width.	Same type and width as the input.	Bitwise complement.
Bitwise OR	Two inputs of any type. Both inputs have to be the same width or one can be a single strand of BIT.	One output, with the same type and number of strands as the stranded input.	Bitwise logical OR. All the array of bits are ORed on a strand-by-strand basis. Single strands of type BIT are applied uniformly across all strands of the other input.
Bitwise NOR			Bitwise logical NOR. All the array of bits are NORed on a strand-by-strand basis. Single binary logic lines are applied uniformly across all strands of the other input.
Bitwise AND			Bitwise logical AND. All the array of bits are ANDed on a strand-by-strand basis. Single binary logic lines are applied uniformly across all strands of the other input.
Bitwise NAND			Bitwise logical NAND. All the array of bits are NANDed on a strand-by-strand basis. Single binary logic lines are applied uniformly across all strands of the other input.
Bitwise XOR			Bitwise logical XOR. All the array of bits are XORed on a strand-by-strand basis. Single binary logic lines are applied uniformly across all strands of the other input.
Bitwise XNOR			Bitwise logical XNOR. All the array of bits are XNORed on a strand-by-strand basis. Single binary logic lines are applied uniformly across all strands of the other input.

Table 84 defines the inputs and outputs for the logical reduction operators.

Table 84 —Logical reduction operators

Node function	Input definition	Output definition	Description
Greater than	Two inputs, of type: INTEGER, UNSIGNED INTEGER, LONG, UNSIGNED LONG, CHARACTER, UNSIGNED CHARACTER, FLOAT, DOUBLE, or BIT. and both the inputs shall have the same type and width.	One output of type BIT.	Mathematical test of magnitude. This evaluates to true if arc zero's bit pattern has a greater value than arc one's bit pattern.
Less than or equal			Mathematical test of magnitude. This evaluates to true if arc zero's bit pattern has a lesser or equal value than arc one's bit pattern.
Less than			Mathematical test of magnitude. This evaluates to true if arc zero's bit pattern has a lesser value than arc one's bit pattern.
Greater than or equal	Two inputs, of type: INTEGER, UNSIGNED INTEGER, LONG, UNSIGNED LONG, CHARACTER, UNSIGNED CHARACTER, FLOAT, DOUBLE, or BIT. and both the inputs shall have the same type and width.	One output, single binary logic.	Mathematical test of magnitude. This evaluates to true if arc zero's bit pattern has a greater or equal value of arc one's bit pattern.

Table 85 defines the inputs and outputs for the array of bits operators.

Table 85—Array of bits operators

Node function	Input definition	Output definition	Description
Shift left	Two inputs with left (DATA) of type: INTEGER, UNSIGNED INTEGER, LONG, UNSIGNED LONG, CHARACTER, UNSIGNED CHARACTER, FLOAT, DOUBLE, or BIT (stranded greater than one). Right input (DATA) of type: INTEGER, UNSIGNED INTEGER, LONG, UNSIGNED LONG, CHARACTER, UNSIGNED CHARACTER, or BIT.	One output, with the same number of strands and type as the left (DATA) arc.	Shifts the left operand to the left by the value contained in the right arc.
Shift right			Shifts the left operand to the right by the value contained in the right arc.
Addition	Two inputs, a left and a right with the same number of strands and the same type, but not a user-defined type.		The right arc is added to the left arc.
Subtraction			The right arc is subtracted from the left arc using two's complement arithmetic.
Multiplication	Two inputs, a left and a right with the same number of strands and the same type, but not a user-defined type.	One output. The number of strands depends on the input type. For arrays of bits, the number of strands is twice the number of strands in the input arcs. For floating point arcs, the number of strands is the same.	The left arc is multiplied by the right arc. For array of bits type, the value can take up to twice as many bits to represent.
Division	Two inputs, a left and a right with the same number of strands and the same type, but not a user-defined type.	One output. The number of strands depends on the input type. For types other than DOUBLE or FLOAT, the number of strands is twice the number of strands in the input arcs. For floating point arcs, the number of strands is the same.	Division of the left arc by the right arc. In the case of type BIT, LONG CHARACTER, INTEGER, and the unsigned variants, the high-order strands represent the whole value of the division and the low-order bits represent the remainder. In the case of FLOAT or DOUBLE, there is no whole or remainder values, just the resultant number.
Remainder	Two inputs of type: INTEGER, UNSIGNED INTEGER, LONG, UNSIGNED LONG, CHARACTER, UNSIGNED CHARACTER, or BIT (strand range > 1).	One output, with the same number of strands as the input arcs.	Division is performed, but only the remainder is kept.

Table 86 defines the inputs and outputs for the edge operators.

Table 86—Edge operators

Node function	Input definition	Output definition	Description
- + (01)	One input of single strand of BIT.	One output of single strand of BIT.	The output is true when the input line transitions from zero to one.
+ - (10)			The output is true when the input transitions from one to zero.
+ + (11)			The output is true when the input remains at a steady one.
- - (00)			The output is true when the input remain at a steady zero.
- ? (0?)			The output is true when the input was zero and it transitions to any other state or remains zero.
+ ? (1?)			The output is true when the input was one and it transitions to any other state or remains one.
? - (?0)			The output is true when the input was at any state and it transitions to zero.
? + (?1)			The output is true when the input was at any state and it transitions to one.
? ? (??)	One input of any type and any width.	One output of single strand of BIT.	The input was at an arbitrary value and changes to another arbitrary value including remaining constant.
? ! (?!)			Barterer transition with at least one bit toggling.
? ~ (?~)			Barterer transition with all bits toggling.
->	Two inputs of single strand of type BIT.	One output strand of type BIT.	Left-hand argument transitions before the right hand argument.
<->	List of inputs of single strand of type BIT.		The output is true when the list of input transitions occur in any order, excluding the possibility of simultaneous transitions.
&>	Two inputs of single strand of type BIT.		Left-hand argument transitions before or simultaneously with the right-hand argument.
<&>	List of inputs of single strand of type BIT.		The output is true when the list of input transitions occur in any order, including the possibility of simultaneous transitions.
* + Arbitrary transition to one	One input of single strand of BIT.	One output of single strand of BIT.	The output is true when the input strand transitions from an arbitrary value to a value of one, including the possibility of a constant value.
* - Arbitrary transition to zero			The output is true when the input strand transitions from an arbitrary value to a value of zero, including the possibility of a constant value.
+ * Arbitrary transition from one			The output is true when the input strand transitions from an initial value of one to an arbitrary value, including the possibility of a constant value.
			The output is true when the input strand transitions from an initial value of zero to an arbitrary value, including the possibility of a constant value.

Node function	Input definition	Output definition	Description
- * Arbitrary transition from zero			The output is true when the input has any number of arbitrary signal transitions, including the possibility of a constant value, with arbitrary final value.
* ?			The output is true when the input transitions from an arbitrary steady state and can be followed by arbitrary transitions.
? *			The output is true when the input transitions from a one to a high-impedance state.
(+ ^) Transition from a logical one to a high impedance state			The output is true when the input transitions from a zero to a high-impedance state.
(- ^) Transition from a impedance state			The output is true when the input transitions from a high-impedance state to a one.
(^ +) Transition from a high impedance state to a logical one			The output is true when the input transitions from a high-impedance state to a zero.
(^ -) Transition from a high impedance state to a logical zero			
(~>) Left occurs before right, with the possibility of other edges in between	Two inputs of single strand of type BIT.		The output is true when the left argument occurs before the right, with the possibility of other edges occurring in between.

Table 87 defines the inputs and outputs for the higher function nodes.

Table 87—Higher function nodes

Node function	Input definition	Output definition	Description
Selector (?:)	Two sets of inputs: one for data and the other for control. Control can be a set of inputs where the type shall be a single strand of BIT or a single input where the types can be: INTEGER, UNSIGNED INTEGER, LONG, UNSIGNED LONG, CHARACTER, UNSIGNED CHARACTER, or BIT (strand range > 1). When the number of data inputs is greater than one, there shall be 2 control inputs in the case of multiple control inputs or 2 control strands in the case of a single control input.	One output only. Or In cases where there is a single data input, there shall be an output type of BIT with a single strand. Or In cases where there are multiple inputs, the output type shall be the same type and width as the input.	The output data value is the value of the corresponding data input. Correspondence is determined by selecting the correct input data bus or bit depending on the value of the control input(s). When control evaluates to zero, the data bit or bus zero is the output value. When the control evaluates to 0x01, the output is the value of data bit one or data bus one and so on. The control is evaluated as a binary number from the concatenation of the individual input bits or the single control input word. The values of the control have the least significant bit corresponding to the bit or input with the lowest strand value or control count.

Node function	Input definition	Output definition	Description
	When there is a single data input, there shall be a single control input and the number of strands shall be 2 control strands. When there are multiple data inputs, all these inputs shall have the same data type and width.		
Priority mux (@?:)	Two sets of inputs: one for data and the other for control. Control can be a set of inputs where the type shall be a single strand of BIT or a single input where the types can be: INTEGER, UNSIGNED INTEGER, LONG, UNSIGNED LONG, CHARACTER, UNSIGNED CHARACTER, or BIT (strand range > 1). When the number of data inputs is greater than one, there shall be (control + 1) inputs in the case of multiple control inputs or (control + 1) strands in the case of a single control input. When there is a single data input, there shall be a single control input and the number of strands shall be (control + 1) strands. When there are multiple data inputs, all these inputs shall have the same data type and width.	One output only. Or In cases where there is a single data input, there shall be an output type of BIT with a single strand. Or In cases with multiple inputs, the output type shall be the same type and width as the input.	The output data value is the value of the corresponding data input. Correspondence is determined by selecting the correct input data bus or bit depending on the value of the control input(s). Each control input or bit selects a different data input. When the control bit zero or input zero (modifier bit 0x4000) is on, regardless of the other higher order control inputs or bits, the output is either the value of data bit zero or data bus zero. If control bit or input one is on and control bit of input zero is zero, the output is either data bit one or data bus one and so on. When there are no control inputs or bits on, the output is the value of the last data input or bus.
Vector storage element (@::)	N control logic lines and N data sources of any type. All data input types shall be the same type and strand width for a node.	One output of the same type and strand width as the data inputs.	The output is the value of the data whose corresponding control line evaluates to true. The control lines are evaluated based on each line's modifier value in the order from lowest to highest. The First control line to evaluate to true sets the value of the node to the corresponding data input. If no control line evaluates to true, the node retains its previous value.
Precedence (@)	A single control line whose type shall be single binary logic.	Three optional outputs: a) Action list b) Precedence else c) New scope	If the control line evaluates to true, the action list associated with the new scope is evaluated; otherwise, the precedence else action list is evaluated. The action list members are evaluated one node at a time in the order they are encountered.

Node function	Input definition	Output definition	Description
			When an action list member is encountered, it is completely evaluated before moving to the next member of the list. However, a precedence operator can have an action list if the operator is a member of the action list at a higher nesting level.
Precedence else (:)	One single binary logic input for the control function. Or One input for an else_list.	One output (Action_list). Or One output (else_list) for threading else nodes.	If the control line evaluates to true, the action list is evaluated; otherwise, the precedence else list is evaluated. The action list members are evaluated, one node at a time, in the order they are encountered. When an action list member is encountered, it is completely evaluated before moving to the next member of the list.
Index ([@])	One input of any type used as data. Or One input of type: INTEGER, UNSIGNED INTEGER, LONG, UNSIGNED LONG, CHARACTER, UNSIGNED CHARACTER, or BIT used for addressing.	One output of the same type and width as the data input.	This node shall immediately precede or follow a named node. The data are put into the named node according to the address at this node. This node is typically used as part of precedence operator sequence to capture the clocking conditions.

Table 88 defines the inputs and outputs for the constant value nodes.

Table 88—Constant value nodes

Node function	Input definition	Output definition	Description
Constant, (=)	No inputs.	One BIT of up to 16 strands.	The constant value is the value held by the modifiers field.
Unknown, [?]			The constant unknown value.
High Z, [^]			The constant value of high impedance.

Table 89 defines the inputs and outputs for the miscellaneous operators.

Table 89—Miscellaneous operators

Node function	Input definition	Output definition	Description
Vector delay target	One input, where the output is from a vector Boolean expression.	No output.	Contains the delay and slew actions to take when the application calls delay or slew after evaluating the preceding graph as true.
Vector check target	One input, where the output is from a vector Boolean expression.	No output.	Contains the check actions to take when the application calls check after evaluating the proceeding graph as true.
Positive dynamic latch	Two inputs, both the left and right. are of BIT type with one strand.	One output, of type BIT with a single strand.	Positive dynamic latch tracks the input during the positive portion of the clock and remains latched during the negative portion of the clock. This latch holds its value as a stored charge on the net. Any discharge of this net during the latched portion of the cycle shall not be restored.
Negative dynamic latch			Negative dynamic latch tracks the input during the negative portion of the clock and remains latched during the positive portion of the clock. This latch holds its value as a stored charge on the net. Any discharge of this net during the latched portion of the cycle shall not be restored.
Positive pre-charge node	Two inputs, both the left and right. are of BIT type with one strand.	One output, of type BIT with a single strand.	Positive dynamic logic has the value of the right arc when the left arc is a one and a logic level of one when the left arc has a value of zero. This is a dynamic discharge node; that is, while the left arc is at a value of one and the net evaluates to zero, it shall not regain a one until the left arc goes to the value of zero.
Negative pre-charge node			Negative dynamic logic has the value of the right arc when the left arc is a zero and a logic level of one when the left arc has a value of one. This is a dynamic discharge node. That is, while the left arc is at a value of zero and the net evaluates to one, it shall not regain the value of zero until the left arc goes to the value of one.

Table 90 defines the inputs and outputs for the user-defined operators.

Table 90—User-defined operators

Node function	Input definition	Output definition	Description
Special logical and math functions	Any number of inputs and type with any number of strands.	Any number of outputs and type with any number of strands.	Evaluation is performed by the library. The library is responsible for generating the bit patterns. The application determines the total bit space by summing all the unique strands emitting from the node.

An event occurs at the time the signal's transition crosses the threshold voltage. One signal is considered to follow another when the following signal's event occurred at a later point in time.

A signal in the steady state shall be defined as a signal that has completed a transition and has not yet begun another transition. A transition is considered completed when the signal voltage has reached the upper or lower transition threshold voltage.

7.16.9.11.6 MODIFIERS clause

Some primitive operators take on different semantic meaning when operating on different types of data. The MODIFIERS clause identifies in which of the different possible semantic meanings the operator is being used. Modifiers are broken into different categories depending on the primitive. Basic primitives, such as AND and OR, have some generic drive strength bits. Other operators, such as + and –, have operations based on the data organization. Table 91 defines the various groups of operators and the meaning associated with the modifiers that can be present.

Table 91—Valid modifier enumerations for given node primitive operators

Node primitive operator	Modifier enumeration	Description
Logic reduction operators		
NONE	0x0001	Storage node
! ! && !&& ^^ !^^	0x0001	Weak one, no pull-up device (MODIFIER_WEAK_ONE)
		Weak zero, no pull-down device (MODIFIER_WEAK_ZERO)
Binary reduction operators		
> >= < <= ==	0x0001	Weak one, no pull-up device (MODIFIER_WEAK_ONE)
		Weak zero, no pull-down device (MODIFIER_WEAK_ZERO)
Bitwise operators		
~ ~ ~! ~! ~& ~!& ~^ ~!^	0x0001	Weak one, no pull-up device (MODIFIER_WEAK_ONE)
		Weak zero, no pull-down device (MODIFIER_WEAK_ZERO)
Array of bits operators these bits can be ORED together		
^> ^> ^>	0x0001	
	0x0000	

Node primitive operator	Modifier enumeration	Description
	0x0004	
	0x0008	
	0x0010	
		Inject a one (MODIFIER_INJECT_ONE)
Mathematical operators these bits can be ORed together		
+		
		Weak zero, no pull-down device (MODIFIER_WEAK_ZERO)
		Unsigned
-		
		Weak zero, no pull-down device (MODIFIER_WEAK_ZERO)
		Unsigned
Miscellaneous operators		
Positive dynamic latch, Negative dynamic latch, Positive pre-charge node, Negative pre-charge node, High impedance	0x0000 - 0xFFFF	Relative drive strength, where the greater the value, the greater the drive strength.
User-defined operators		
Special logical and math functions	0x0000 - 0xFFFF	User-defined(DATA_TYPE_USER_DEFINED).

7.16.9.11.7 OBJTYPE clause

In some situations, it is advantageous for both the library and the application to first represent a cell at a higher level of abstraction. Each node in this abstraction can hold a complex function that is known by name. If the application knows the function at this node by name, then the modeling of the lower level of detail shall be omitted. The OBJTYPE clause holds a string that represents the “well-known” name of the function. If the named function is not understood, the application can query the library for a more detailed expansion of the function (for the next level of hierarchy). The application shall call modelSearch with the Standard Structure fields configured as follows, when additional expansion is desired:

- Cell set to a string containing the same character sequence that was contained in the OBJTYPE clause.

- cellQual set to a string containing the same character sequence previously contained in cellQual (during the original call to modelSearch, which returned the OBJTYPE clause).
- modelDomain set to a string containing the same character sequence that was contained in modelDomain (during the original call to modelSearch, which returned the OBJTYPE clause).
- inputPins array containing the list of source nodes or pins that had arcs connected to the node containing the OBJTYPE clause (in the original model that is being expanded).
- inputPinCount set to the count of the elements contained in inputPins.
- outputPins array containing the list of sink nodes or pins that had arcs connected to the node containing the OBJTYPE clause (in the original model that is being expanded).
- outputPinCount set to the count of the elements contained in outputPins.
- Nodes and nodeCount set to a value of zero (0).
- Block set to the block name of the containing cell, which defined this internal node.

7.16.9.12 Function_sequence grammar

Functional descriptions are presented to the application as a graph. Sometimes, it is convenient for the library developer to construct the function directly using node and path notations. Other times, it is easier to represent the function of a cell as a Boolean expression. The function sequence represents a syntax that allows the library developer to represent the cell's function as a combination of boolean expressions and Boolean assignments. Syntax 7.109 defines the syntax for expressing function in terms of Boolean expressions.

```

function_assignment_expression ::= assignment_sequence =
    boolean_expression | precedence_expression
function_assignment_expression_list ::= function_assignment_expression
    { , function_assignment_expression }
precedence_expression ::= @ boolean_expression
    {function_assignment_expression_list}
    { : boolean_expression {function_assignment_expression_list} }
boolean_expression ::= pin_or_node
    | monadic_operator boolean_expression
    | boolean_expression dyadic_operator boolean_expression
    | <> {boolean_expression { , boolean_expression }}
    | {boolean_expression { , boolean_expression }}
    | <-> {boolean_expression { , boolean_expression }}
    | <&> {boolean_expression { , boolean_expression }}
    | @:: {boolean_expression { , boolean_expression }}
    | {boolean_expression { , boolean_expression }}
    | @?: {boolean_expression { , boolean_expression }}
    | {boolean_expression { , boolean_expression }}
    | ?: {boolean_expression { , boolean_expression }}
    | ( boolean_expression )
assignment_sequence ::= pin_or_node
    | pin_or_node [boolean_expression]
pin_or_node ::= pin_or_node_name |
    VAR ( pin_variable_expression )
function_sequence ::= FUNCTION ( function_assignment_list )
monadic_operator ::= |= | |? | |^ | ++ | -- | ! | -|+
    | +|- | +|^ | -|^ | ^|+ | ^|- | -|? | +|? | ?|+ | ?|-
    | ?|? | ?|~ | ?|! | ?|* | *|? | *|+ | *|- | ~ | +|* | -|*
dyadic_operator ::= / | ** | - | + | < | > | % | !! | !&& | ^^ | !^^ |
    ~| | ~& | ~^ | ~& | ~| | ^ | > | < | == | >= | <= | || | && | -
    > | &> | ~>

```

Syntax 7.109—function_assignment_expression

7.16.9.12.1 FUNCTION clause

The FUNCTION clause transforms a Boolean assignment and its associated Boolean expression (or a precedence expression) into a graph notation that the application can follow. The graph is transferred to the application via a sequence of implicit callbacks.

7.16.9.13 Vector_sequence grammar

Some applications track the state of a cell as it is performing timing analysis. These applications leverage the knowledge of state when computing delay, slew, timing checks, and power. The VECTOR clause expresses state-dependent delay, slew, timing checks, and power as a set of associated Boolean expressions and propagation sequences, as defined in Syntax 7.110.

```
vector_sequence ::= VECTOR ( boolean_expression )
                  vector_propagation_sequence
vector_propagation_sequence ::= vector_from_to
                              PROPAGATE ( edge_mode_expression ) vector_action_sequence
                              | [store_or_methods_clause]
vector_from_to ::=
    FROM ( pin_or_node ) TO ( pin_or_node )
vector_action_sequence ::=
    [DELAY ( delay_stmt_name ( parameter_list ) ) ]
    [store_or_methods_clauses]
    | [SLEW ( slew_stmt_name ( parameter_list ) ) ]
    [store_or_methods_clauses]
    | [CHECKS ( check_stmt_name ( parameter_list ) ) ]
    TEST_TYPE ( test_types ) [store_or_methods_clauses]
store_or_methods_clauses ::=
    { [store_clause] | [methods_clause] }
```

Syntax 7.110—vector_sequence

7.16.9.14 VECTOR clause

The VECTOR clause transforms a Boolean expression into a graph of nodes and arcs similar to those in the FUNCTION clause. The graph generated by a VECTOR clause terminates into a special node, which indicates to the application a vector timing operation needs to be performed. In addition to the node primitive value being unique to the vector terminating node, the path data and Standard Structure contain additional information enabling methods, store operations, delays, slews, or checks. If the VECTOR clause is used for timing, it shall be in the context of a model domain named vectorTiming. If the VECTOR clause is used for power, it shall be in the context of a model domain named vectorPower.

When a VECTOR clause is used to represent a timing-state expression for a timing segment, it shall be in the context of a model domain named “timing.” Furthermore, it shall be a degenerative form of the VECTOR clause used in other model domains. The vector expression in the clause shall contain only constant values and logical operators. The clause shall not contain a propagation sequence. The timing state represented by the clause shall be for the from-pin, to-pin and transition specified for the associated timing segment.

Example

```
submodel(triStateTimingStateGraph):
do: vector(E==1);
end;
```

7.16.9.15 IMPORT and EXPORT sequences

The syntax for the *NODE* clause is given in Syntax 7.111.

```
import_export_sequence ::= import_sequence | export_sequence  
import_sequence ::= IMPORT ( name_or_string ) propagation_sequence  
name_or_string ::= name | string_literal  
export_sequence ::= EXPORT ( name_or_string ) propagation_sequence
```

Syntax 7.111—import_export_sequence

A *NODE* clause creates a new timing point referred to as a “*NODE*.” A group of related clauses can potentially describe its interconnection to external circuits plus any propagation properties, including *DELAY* and *SLEW*.

If the timing point created is connected to the input or output pins of the circuit it is modeling, then the *PATH* statement shall be used to connect the new timing point. However, if the new timing point is connected to another circuit’s input or output pins, then the *NODE* clause’s *IMPORT* or *EXPORT* clause shall be used. The *IMPORT* and *EXPORT* clauses do not set the *PATH* predefined identifier.

The *NODE* name, *IMPORT* clause, and *EXPORT* clause are used as follows:

- *NODE* name

The *NODE* clause argument is the name of a single node to be created in parentheses. This name shall not collide with an existing pin name for the cell.

- *IMPORT* clause

The *IMPORT* clause is used to connect the newly created timing point (node) to the output pins of another circuit. The *IMPORT* clause instructs the application to create arcs from all pins that drive the net associated with the argument, except the argument itself, and connect them to the newly created timing point.

- *EXPORT* clause

The *EXPORT* clause is used to connect the newly created timing point (node) to the input pins of another circuit. The *EXPORT* clause instructs the application to create arcs from the newly created timing point to all pins that are sinks on the net associated with the argument, except the argument itself.

7.16.10 PROPERTIES statement

The syntax for the *PROPERTIES* statement is given in Syntax 7.112.

```

properties_statement ::= PROPERTIES : [conditional_store_seq] ;
conditional_store_seq ::= [method_store_sequence]
    | when_properties_clause [ , otherwise_properties_clause ]
method_store_sequence ::= [pre_code] [methods_and_store] [post_code]
pre_code ::= reference_list
reference_list ::= reference_item { reference_item }
reference_item ::= embedded_C_code
    | statement_reference
methods_and_store ::= methods_clause store_clause
    | store_clause methods_clause
methods_clause ::= METHODS ( methods_action_stmt_list )
store_clause ::= STORE ( store_cache_list )
post_code ::= reference_list
when_properties_clause ::=
    WHEN ( logical_expression ) [method_store_sequence]
    { , WHEN ( logical_expression ) [method_store_sequence] }
otherwise_properties_clause ::= OTHERWISE [method_store_sequence]

```

Syntax 7.112—properties_statement

The **PROPERTIES** statement stores function results (via the **STORE** clause) and associates **METHOD** action statements with a cell (via the **METHODS** clause). A **MODELPROC** shall have at most one **PROPERTIES** statement, which shall appear before the first **INPUT**, **OUTPUT**, **PATH**, **BUS**, or **TEST** statement and before any **DO** statement that contains a **NODE** clause or **CALL** clause.

7.16.11 SETVAR statement

The syntax for the **SETVAR** statement is given in Syntax 7.113.

```

setvar_statement ::= SETVAR ( name ) : conditional_result ;

```

Syntax 7.113—setvar_statement

The **SETVAR** statement creates and sets the values of variables local to a **MODELPROC** procedure.

The **SETVAR** statement has a similar meaning and syntax as the **ASSIGN** statement except for the following:

- **SETVAR** shall not be referenced as a statement or be passed any variables.
- **SETVAR** variables shall become undefined between calls to the containing **MODELPROC** and therefore shall not be used to save information between calls to the same model.
- **SETVAR** shall be executed, and its variable(s) created, when it is encountered.

A reference to a **SETVAR** variable is identical to that of an **ASSIGN** variable (see Syntax 7.23). **SETVAR** references may be used anywhere a variable reference is allowed, except it may not be used in **DELAY**, **SLEW**, or **CHECK** clauses.

SETVAR statements may be used inside successively nested **STATEMENTS** clauses. Each nested **STATEMENTS** clause shall introduce a new scope, such that each nested **SETVAR** temporarily “hides” the value of any **SETVAR** with the same name but is contained within an outer **STATEMENTS** clause.

7.17 Embedded C code

The syntax for the a C code statement is given in Syntax 7.114.

```
embedded_C_code ::= %{ C_language statement }%
```

Syntax 7.114—embedded_C_code

In-line C declarations and function definitions may be inserted anywhere in a subrule where a DCL statement may appear. Any include files that are required by embedded C code shall also be explicitly coded in the embedded C code.

In-line C-type definitions, type declarations, and C statements, other than function definitions or function prototypes, may be inserted within modeling statements as `pre_code` or `post_code`.

Embedded C code shall be executed when the DCL statement that references it is executed and the embedded C code reference is encountered.

7.18 Definition of a subrule

The syntax for a subrule is given in Syntax 7.115.

```
subrule ::= [tech_family_statement] {statement}
statement ::= prototype_statement
| statement_definition
| model_statement
| table_statement
| environment_control_statement
prototype_statement ::= common_prototype_statement
| unload_table_prototype
| load_table_prototype
| add_row_prototype
| delete_row_prototype
| tabledef_prototype
| delay_prototype
| check_prototype
statement_definition ::= assign_statement
| calc_statement
| expose_statement
| external_statement
| internal_statement
| embedded_C_code
table_statement ::= unload_table_statement
| add_row_statement
| delete_row_statement
| tabledef_statement
| table_statement
| load_table_statement
model_statement ::= model_statement
| model_procedure
| submodel_procedure
environment_control_statement ::= subrule_statement
| subrules_statement
```

Syntax 7.115—subrule

7.19 Pragma

A PRAGMA is a directive to the compiler that causes the compiler to change its behavior but does not change the behavior of the language.

7.19.1 IMPORT_EXPORT_TAG

For the purposes of linking, only IMPORT_EXPORT_TAG concatenates the tag name to the imported or exported statement's name. This tag may be used to create sets of imported and exported statements that link together based on the combination of the statement's name and the tag name. The syntax for a pragma_declare is given in Syntax 7.116.

```
pragma_declare ::= PRAGMA IMPORT_EXPORT_TAG ( name ) ;
```

Syntax 7.116—pragma_declare

8. Power modeling and calculation

This clause describes the power modeling and calculation used in this standard. The formal syntax is described using the BNF, the conventions of which are described in Clause 7.

8.1 Power overview

There are three techniques for power calculation. Each technique has its own requirements and responsibilities for both the application and the DPCM. The techniques can vary in computational accuracy and execution speed due to the type and amount of information needed. The three power calculation techniques are as follows:

- The AET or “All Events Trace” technique using *dpcmGetAETCellPowerWithSensitivity*
- The “Group” technique using *dpcmCellPowerWithState*
- The “Pin Power” technique using *dpcmPinPower*

The application and DPCM can choose to model any combination of the power computation techniques on an instance-by-instance basis. Therefore, “handshaking” between the application and the DPCM is required to agree on the technique to use for each instance. If the application and the DPCM do not support a common technique for each cell, power calculation may be severely limited.

The techniques of power calculation supported by the DPCM are returned by the call to *dpcmGetCellPowerInfo*. This call returns, per cell, the power techniques supported. The following information may also be returned: group pin lists, group condition lists, sensitivity lists, and initial state choices (depending on the DPCM supported power calculation techniques). A power state is an electrical condition in which the cell can persist.

For cells that have at least one initial state, the DPCM creates a state cache (during the call to *dpcmSetInitialState*) and returns a handle to this cache to the application for each instance (see 8.2). This state cache is used by the DPCM to track the state of this instance. The power model itself needs to define the choice and representation of the state. The DPCM can use a state cache for any of the power modeling techniques.

All load and slew information required for power calculation is supplied by the application. The load and slew information is cached by the DPCM and a handle to this cache is returned to the application. This caching technique and the associated load and slew cache handle are described in 8.2.

The net energy for completed logic transitions is calculated by a call to *dpcmGetNetEnergy*.

It is the responsibility of the application to accumulate the power over time. The power returned from the DPCM is given in terms of “static power” and/or “dynamic energy,” depending on the technique of power calculation. The dynamic and static components of power are defined and used as follows:

- Dynamic energy is as follows:

dynamic_energy_captured_during_the_transition +
(*static_power_for_the_state_transitioning_into* * *time_of_transition*)

The dynamic energy returned by the DPCM shall not include a static leakage component as follows:

- Static power is as follows:

the_power_for_the_state_just_transitioned_into

The application shall multiply the returned static power value by the time from this change to the next monitored change on this instance.

8.2 Caching state information

The state cache is private to the DPCM. The contents of the cache shall be defined by the individual power model. The DPCM is responsible for allocating the associated memory (by request from the application), returning the cache handle to the application, and updating the data stored in this cache.

The application is responsible to request the cache be created, to associate the returned cache handle with the instance for which it was requested, and to free the state cache when it is no longer needed.

For each instance with at least one initial modeled state, the application shall obtain a cache handle by calling *dpcmSetInitialState*. During a power calculation request, the DPCM shall call *appGetStateCache* to retrieve the state cache handle for the instance specified in the Standard Structure. The DPCM shall only call back to the application to retrieve this state cache handle for cell types that have at least one initial state modeled.

8.2.1 Initializing the state cache

The application shall initialize the state cache by calling *dpcmSetInitialState* and passing the desired initial state index. If a zero cache handle is passed in, the DPCM shall create and return a new cache handle initialized to the specified state. If a previously created cache handle is passed into *dpcmSetInitialState*, the DPCM may reuse this cache (the same cache handle is returned to the application), or may free this cache and allocate a new one (a different cache handle is returned to the application). In either case, the state of the cache shall be identical.

8.2.2 State cache lifetime

A cache handle is valid from the time it is created until it is freed by the application via *dpcmFreeStateCache* or freed by the DPCM when passed into *dpcmSetInitialState*.

8.3 Caching load and slew information

The load and slew cache is private to the DPCM. The DPCM is responsible for allocating the associated memory (by request from the application), returning the cache handle to the application, and updating the data stored in this cache.

The application is responsible to request the cache be created, to associate the returned cache handle with the cell type or instance for which it was requested, to request that the cache be updated, and to free the cache when it is no longer needed.

The application initiates a power calculation request using one of the following:

- *dpcmGetAETCellPowerWithSensitivity*
- *dpcmGetCellPowerWithState*
- *dpcmGetPinPower*

Then the DPCM shall call back to the application requesting the load and slew information necessary to perform the calculation (except in the case of *dpcmGetPinPower* if the application has specifically requested that it not be called back for this information).

This callback, *appRegisterCellInfo*, has three input parameters that indicate the specific data being requested: loading capacitance, loading resistance, and transition (slew). These input parameters also indicate the pin types (inputs, outputs, bidirectionals, or all types) for which the requested information is needed. If the application does not know a value being requested, it shall supply a value of zero (0) for that field.

8.3.1 Loading the load and slew cache

The application shall take one of the following actions after being called by the DPCM (via *appRegisterCellInfo*):

- Call *dpcmFillPinCache* to fill a cache with the requested data for each requested pin type on the instance for which power is being calculated. These calls fill the load and slew information into a cache to be used by the DPCM for the current power calculation request.

On the first call to *dpcmFillPinCache* within this *appRegisterCellInfo* callback, the application shall either pass in a0 handle (zero); in which case, the DPCM shall create a new cache or a cache handle created during a previous power calculation request provided the cell type remains the same. If a previous cache handle is used, the data previously filled into that cache remain valid, and *dpcmFillPinCache* only needs to be called for those pins where the data being requested are different than that already in the cache.

On all subsequent calls to *dpcmFillPinCache* within this *appRegisterCellInfo* callback, the application shall pass in the cache handle returned from the previous call to *dpcmFillPinCache*. The cache handle returned from the final *dpcmFillPinCache* call is then passed back to the DPCM as a return parameter on the *appRegisterCellInfo* call. This cache is then used for the current power calculation. The application may choose to save the cache handle returned to the DPCM for subsequent power calculation requests.

- Return the handle of a cache that was filled during a previous power calculation request of the same cell instance or type.

8.3.2 Load and slew cache lifetime

A cache handle remains valid, along with the contents of the cache, until either the application frees it (via *dpcmFreePinCache*) or the cache handle is invalidated during a call to *dpcmFillPinCache*. If the application passes a nonzero cache handle to *dpcmFillPinCache* and the DPCM returns a different cache handle, then the cache handle passed in by the application is invalidated and shall not be used for any subsequent power calculation request. If a cache is invalidated in this way, the DPCM is responsible to

copy all the data from the previous cache to the new cache, update the new cache with the data being passed in on the current call, and free the previous cache.

8.4 Simulation switching events

Two or more pin change events are considered simultaneous when these events occur within a defined time interval called the “simultaneous switching window.” Simultaneous switching windows are defined between pins on a cell using *dpcmAETGetSimultaneousSwitchTime* for the AET power calculation technique and *dpcmGroupGetSimultaneousSwitchTime* for group power calculation technique. There is no simultaneous switching window for the pin power calculation technique.

For AET and group power calculation techniques, events that are considered simultaneous shall be considered together and processed in the same power calculation request. For pin power calculation, power calculation requests shall be made separately for all events, regardless of how closely together they occur.

8.5 Partial swing events

A “Settling Time Window” is the time interval specified for a change on a pin to make a complete transition. A “Partial Swing” occurs when the pin change duration is less than the settling time for that pin, the electrical level of that pin changes and then changes back (becomes unstable) during the settling time window. The “settling time window” is defined as the time interval required for a change on a pin to make a complete transition.

Settling time windows are defined between pins on a cell using *dpcmAETGetSettlingTime* for the AET power calculation technique and *dpcmGroupGetSettlingTime* for group power calculation technique. There is no settling time window for the pin power calculation technique.

For the AET power technique, power is calculated for partial swing events by using a call to *dpcmCalcPartialSwingEnergy* rather than *dpcmGetAETCellPowerWithSensitivity*.

For the group power technique, power is calculated for partial swing events using a call to *dpcmCalcPartialSwingEnergy* instead of calls to *dpcmGetCellPowerWithState*. Here, the application evaluates the group condition expressions as if the pin change had made the full transition. For each condition expression that evaluates to true, a call to *dpcmCalcPartialSwingEnergy* is made.

There is no provision to calculate the power of a partial swing when an instance is being modeled with the pin power technique.

8.6 Power calculation

The following list details the sequence of events for power calculation:

- a) Model for power. Before calling any of the power functions, including *dpcmGetCellPowerInfo*, the application shall call *modelSearch* on the cells of interest. It can use *dpcmGetCellList* to determine whether power is modeled separately from timing. If so, then a separate call to *modelSearch* is required, with the MODEL_DOMAIN set to power.
- b) Determine the DPCM supported power calculation techniques (per instance). The application calls *dpcmGetCellPowerInfo* for each cell to determine the DPCM supported techniques of power calculation for instances of that cell. The application is free to call any of the DPCM-supported techniques per instance. Power calculation results are undefined if the power calculation technique for an instance is switched after the power computations have begun.
- c) Determine the application-supported power calculation techniques (per instance). If the application knows the chronological changes in logic levels of the requested pins of a instance, the AET power

calculation technique can be used for this instance. Guided by the sensitivity list, the application passes pin changes to the DPCM via *dpcmGetAETCellPowerWithSensitivity*. The DPCM is then responsible for tracking the state of the instance.

If the application knows the logic levels and change events of the requested pins of a instance and the application can process the group condition language, then the group power calculation technique can be used for this instance. Guided by the group pin list and the group condition list, the application is responsible to determine which condition expressions are true and request the power associated with each of these condition expressions via *dpcmGetCellPowerWithState*. The application is responsible for tracking the state of the instance.

If the application knows when pins of an instance transition but does not know the present or previous logic levels of these pins, then the following pin power calculation technique can be used for this instance. The application passes the pin that changes to the DPCM via *dpcmGetPinPower*:

- d) Establish initial states. Initial state choices are specified on a per instance basis. Setting the initial state may be done for any of the supported power calculation techniques. For each instance that has initial state choices (as returned by *dpcmGetCellPowerInfo*), the application shall initialize the instance to one of its initial states prior to any power computation via *dpcmSetInitialState*. The application shall associate the state cache handle returned from *dpcmSetInitialState* with the instance specified in the Standard Structure.
- e) While the application observes pin changes, the following should occur:
 - 1) For AET and group power calculation techniques only:
 - i) Determine whether the current pin changes are to be considered simultaneous; see 8.4. If these events are considered simultaneous, then the application accumulates these pin changes and makes a single call for power as if all the pins changed at the same time.
 - ii) Determine whether the current pin changes are to be considered a partial swing; see 8.5. If these events are to be treated as a partial swing, then the application shall make a separate call to calculate the power consumed by this partial swing in place of the AET or group power call.
 - 2) The application initiates a power calculation request:
 - i) *dpcmGetAETCellPowerWithSensitivity* technique (AET). If this technique is used, the application is responsible for monitoring the pins returned in the sensitivity list for changes. These changes are passed into this call in the form of a mask. The mask defines the type of change that has occurred, such as transitions 0->1, 0->0, 1->0, 0->X and 1->HIZ. See 8.11.2.6 for the details of the data being passed into this call.

Chronological ordering of events is important in this technique because the DPCM may keep the state history, within the state cache.

- ii) *dpcmCellPowerWithState* technique (group). If this technique is used, the application shall monitor the union of pins specified in the group pin list array. When a monitored pin transitions, the application shall identify which pin groups contain the pin (“affected pin groups”). For each affected pin group, the application shall evaluate all associated group condition expressions. For each group condition expression that evaluates true, the application shall call for power (either as these events occur or after accumulating these events).

- iii) Chronological ordering of events is not required for this technique. The DPCM cannot use the chronological ordering of power calculation requests to keep a representation of previous states.
 - iv) *dpcmPinPower* technique (pin). If this technique is used, the application is responsible to call *dpcmPinPower* for power on each pin change event. This technique requires no knowledge on the part of the application or the DPCM about the present or previous pin logic levels. It can be thought of as a power estimate for a single transition on a pin. This call can be made as each event occurs or after tracking the pin transitions over time. The DPCM cannot use the chronological ordering of power calculation requests to keep a representation of previous states.
- 3) The DPCM calls the application for state information. For each of these power calculation techniques, if the instance for which a power calculation is being requested has at least one initial state choice, then the DPCM shall call back to the application via *appGetStateCache* for the state cache handle associated with this instance.
 - 4) The DPCM calls the application for load and slew information. For each of these power calculation techniques (except when the application calls for pin power and explicitly indicates it should not be called back for load and slew data), the DPCM shall call back to the application via *appRegisterCellInfo*. This function passes in three flags indicating the type of information being requested (capacitance, resistance, and/or slew) and the types of pins for which the requested information is needed (i.e., inputs, outputs, bidirectionals, or all). This callback (*appRegisterCellInfo*) enables the application to update (if necessary) the load and slew cache for this instance prior to the power calculation via *dpcmFillPinCache*.
 - 5) The DPCM calculates power for this instance. The DPCM shall calculate and return the static power and dynamic energy for the specific event (AET power), condition (group power), or pin transition (pin power) for each power calculation request.

8.7 Accumulation of power consumption by the design

The application is responsible for accumulating the power calculations for the individual events to determine the total power consumption for a design.

8.8 Group Pin List syntax and semantics

The Group Pin List is an array of strings that is returned by *dpcmGetCellPowerInfo* when the *dpcmCellPowerWithState* power calculation technique is supported. A zero (0) length array is returned when this information is not available.

Each element of this array is called a *GroupPinString*. The index into this array is called the *GroupIndex*. This array is indexed from 0 to $n - 1$, where n is the number of array elements. The *GroupIndex* is one of the parameters passed into *dpcmCellPowerWithState*.

The application shall parse each string in the group pin list array to determine which pins are associated with each *GroupIndex*.

8.8.1 Syntax

The syntax for a *GroupPinString* is given in Table 92.

Table 92—Syntax for a GroupPinString

```
GroupPinString ::= 'group_pin_list'
group_pin_list ::= group_pin_name { , group_pin_name }
group_pin_name ::= PinName | ALLIN | ALLOUT | ANYIN | ANYOUT
```

PinName is a sequence of any ASCII, non-whitespace characters except the following special characters, which shall be escaped with a preceding backslash (\) if used (\), ('), (.).

8.8.2 Semantics

The semantics for a *GroupPinString* are as follows:

- A *PinName* shall not be duplicated within one *GroupPinString*.
- A *PinName* may be duplicated in other *GroupPinStrings*.
- A *PinName* shall be an actual pin name, ANYIN, ANYOUT, ALLIN, or ALLOUT.
- A Pin Range (see 7.8.13.1) is not allowed in a *PinName*.

8.8.2.1 Interpreting ANYIN or ANYOUT in a GroupPinString

The *PinName* ANYIN is equivalent to listing all the inputs and bidirectional pins of the cell in question. This means if any one of those pins changes value, the associated condition expressions shall be evaluated.

The *PinName* ANYOUT is equivalent to listing all the outputs and bidirectional pins of the cell in question. This means if any one of those pins changes value, the associated condition expressions shall be evaluated.

8.8.2.2 Interpreting ALLIN or ALLOUT in a GroupPinString

The *PinName* ALLIN means all inputs and bidirectional pins shall transition together before the associated condition expressions are evaluated.

The *PinName* ALLOUT means that all outputs and bidirectional pins shall transition together before the associated condition expressions are evaluated.

8.8.3 Example

The following is a sample Group Pin List.

```
group_pin_list[0] = 'A, B, C, D, E, F'
group_pin_list[1] = 'A, B, C'
group_pin_list[2] = 'X, Y, Z'
group_pin_list[3] = 'A[0],A[1],A[2],A[3],B[0],B[1],B[2],B[3],Q'
```

8.9 Group Condition List syntax and semantics

The Group Condition List is an array of strings that is returned by *dpcmGetCellPowerInfo* when the *dpcmCellPowerWithState* power calculation technique is supported. A zero (0) length array is returned when this information is not available.

The Group Condition List array and the Group Pin List array are parallel arrays. This means the data in each array at the corresponding index are related (e.g., the Group Pin List array data at index 2 corresponds to the Group Condition List array data at index 2).

Each element of the Group Condition List array is called a *GroupConditionString*. The application uses the index of the Group Pin List array to index into the Group Condition List array and find the associated *GroupConditionString*. The *GroupConditionString* is composed of one or more elements (separated by commas). These elements are called “condition expressions.” The position of each element in the *GroupConditionString* is called the *ConditionIndex*. These positions are indexed from 0 to $n - 1$, left to right, where n is the number of condition expressions in the *GroupConditionString*.

The *GroupIndex* and the *ConditionIndex* uniquely identify a condition expression. These two indices are passed to *dpcmCellPowerWithState* to compute the power for this condition expression.

The application shall parse each *GroupConditionString* to determine both the condition expressions associated with this index (row) and the position (column) of these condition expressions within each row. The interpretation of these condition expressions is described in 8.11.2.

8.9.1 Syntax

The syntax for a *GroupConditionString* is given in Table 93.

Table 93—Syntax for a *GroupConditionString*

```
GroupConditionString ::= ' condition_list '
condition_list ::= condition_expression
                 { , condition_expression }
```

Condition_expression is defined in Table 95.

8.9.2 Semantics

The semantics for a *GroupConditionString* are as follows:

- Each comma delimited *condition_expression* constitutes a *ConditionIndex*.
- The application shall evaluate all *condition_expressions* within the selected *GroupConditionString*. For each *condition_expression* that evaluates to true, the application shall call *dpcmCellPowerWithState* to compute the power.
- The list of condition expressions within a group condition string does not have to be exhaustive. The * (the universal complement operator) condition string, when specified, is the default condition used when none of the other condition expressions apply.

8.9.3 Example

The following is a sample Group Condition List.

Example

```
GroupStateArray[0] = 'A&&B&&C&&Q=Q-1, A==A-1&&!B&&!C&&!Q'
GroupStateArray[1] = '*'
GroupStateArray[2] = '!X||!Y||Z!=Z-1,*'
```

8.10 Sensitivity list syntax and semantics

The Sensitivity List is an array of strings that is returned by *dpcmGetCellPowerInfo* when the *dpcmGetAETCellPowerWithSensitivity* power calculation technique is supported. A zero (0) length array is returned when this information is not available.

The Sensitivity List array is indexed 0 to $n - 1$, where n is the number of array elements. The application shall parse each element of the Sensitivity List array to associate the *PinNames* found with their array element index. Pin changes are communicated to the DPCM through a parallel array in which the pin change values are passed to the DPCM in the same array position as that in which the *PinName* was found in the Sensitivity List. See 8.11.4 for more information on how pin changes are communicated to the DPCM.

8.10.1 Syntax

The syntax for a SensitivityPinString is given in Table 94.

Table 94—Syntax for a SensitivityPinString

```
SensitivityPinString ::= ' sensitivity_pin_list '  
sensitivity_pin_list ::= sensitivity_pin_name  
    { , sensitivity_pin_name }  
sensitivity_pin_name ::= PinName | ANYIN | ANYOUT
```

PinName is a sequence of any ASCII, non-whitespace characters except the following special characters, which shall be escaped with a preceding backslash (\) if used: '\,

8.10.2 Semantics

The semantics for a SensitivityPinString are as follows:

A *PinName* shall not be duplicated within one *SensitivityPinString*.

A *PinName* may be duplicated in other *SensitivityPinStrings*.

A *PinName* shall be an actual pin name, ANYIN, or ANYOUT.

A *PinName* shall not be ALLIN or ALLOUT.

A Pin Range (see 8.11.2) is not allowed in a *PinName*.

The *PinName* ANYIN in a *SensitivityPinString* is equivalent to listing all the inputs and bidirectional pins of the cell in question.

The *PinName* ANYOUT in a *SensitivityPinString* is equivalent to listing all the outputs and bidirectional pins of the cell in question.

8.10.3 Example

The following is a sample Sensitivity List:

Example

```
sensitivity_list[0] = 'A0'  
sensitivity_list[1] = 'A1'  
sensitivity_list[2] = 'A[0],A[1],A[2],A[3]'  
sensitivity_list[3] = 'A0, WCLK, READ'
```

8.11 Group condition language

For the *dpcmCellPowerWithState* power calculation technique, the DPCM defines each *condition_expression*, which represents a logical function of the pins on a cell (internal nodes of a cell shall not be included in these expressions). When a *condition_expression* evaluates to true, the application calls *dpcmCellPowerWithState* to request the power consumption associated with that *condition_expression*.

8.11.1 Syntax

The syntax for a *condition_expression* is given in Table 95.

Table 95—Syntax for a *condition_expression*

```

condition_expression ::= * | L
L ::= PinName_State
    | quoted_label_string
    | ! L
    | ( L )
    | L == L
    | L != L
    | L && L
    | L || L
    | L ^ L
PinName_State ::= PinName_Level | ~ PinName_Level | @ PinName_Level
PinName_Level ::= PinName_Identifier | PinName_Identifier == level
PinName_Identifier ::= PinName | PinName - 1
PinName ::= PinNameId | ANYIN | ANYOUT | ALLIN | ALLOUT
level ::= 1 | 0 | X | Z

```

* is defined as the universal complement of all explicitly named states in this *group_condition_string* involving all the pins listed in the associated *GroupPinString*, and *PinNameId* is a sequence of any ASCII, non-whitespace characters except the following special characters, which shall be escaped with a preceding backslash (\) if used: - ~ " = ! () . , " \ @ & | ^.

8.11.2 Semantics

The semantics for a *condition_expression* are detailed in the following subclauses.

8.11.2.1 Semantic rules for *PinName* and *PinNameId*

The semantics for *PinName* and *PinNameId* are as follows:

- A Pin Range (see 7.8.13.1) is not allowed in a *PinName*.
- A *PinNameId* shall be a valid *PinName* for the cell being modeled.

8.11.2.1.1 Interpreting ANYIN and ANYOUT in a *condition_expression*

The *PinName* ANYIN has the implied ORing of all the inputs and bidirectional pins of the cell specified in the associated *GroupPinString* element.

The *PinName* ANYOUT has the implied ORing of all the outputs and bidirectional pins of the cell specified in the associated *GroupPinString* element.

Where an operator is applied to ANYIN/ANYOUT, the meaning is defined as ORing the effect of the operator on each pin, as shown in the following example.

Example

GroupPinString: A,B : ~ANYIN implies ~A || ~B
 GroupPinString: A,B : (ANYIN) == (ANYIN-1) implies
 (A || B) == (A-1 || B-1)

8.11.2.1.2 Interpreting ALLIN and ALLOUT in a condition_expression

The *PinName* ALLIN has the implied ANDing of all the inputs and bidirectional pins of the cell specified in the associated *GroupPinString* element.

The *PinName* ALLOUT has the implied ANDing of all the outputs and bidirectional pins of the cell specified in the associated *GroupPinString* element.

Where an operator is applied to ALLIN/ALLOUT, the meaning is defined as ANDing the effect of the operator on each pin, as shown in the following example.

Example

GroupPinString: A,B :~ALLIN implies ~A && ~B
GroupPinString: A,B :(ALLIN) == (ALLIN-1) implies
(A && B) == (A-1 && B-1)

8.11.2.2 Semantic rules for PinName_Identifier (named P_{id})

The semantics for a *PinName_Identifier* are shown in Table 96.

Table 96—PinName_Identifier semantics

Operator	Example	Description
	P	Means the present state of P , where P is a pin name.
-1	$P-1$	Means at the last sample of P , where P is a pin name.

8.11.2.3 Semantic rules for PinName_Level (named P_{level})

The semantics for a *PinName_Level* are shown in Table 97.

Table 97—PinName_Level semantics

Operator	Example	Description
==	$P_{id} == V$	Is TRUE when the logic level of P_{id} is V , where P_{id} is either the present (P) or previous ($P-1$) state of the pin named P and V is one of the logic levels: 1, 0, X, Z.
	P_{id}	Is shorthand for $P_{id} == 1$ in a logical condition expression.

8.11.2.4 Semantic rules for PinName_State (shorthand operators)

The semantics for a *PinName_State* are shown in Table 98.

Table 98—PinName_State semantics

Operator	Example	Description
~	$\sim P_{level}$	Is an abbreviation for $(P \neq P-1 \ \&\& \ P_{level})$ and is TRUE when this expression is TRUE, where P is a pin name and P_{level} is a pin name level expression. For example, P_{level} could be one of the following pin name level expressions: $P == X$, $P-1 == 0$, $P == 1$, $P-1$, etc.
@	$@P_{level}$	is an abbreviation for $(P == P-1 \ \&\& \ P_{level})$ and is TRUE when this expression is TRUE, where P is a pin name and P_{level} is a pin name level expression. For example, P_{level} could be one of the following pin name level expressions: $P == X$, $P-1 == 0$, $P == 1$, $P-1$, etc.

8.11.2.5 Condition expression labels

A *condition_expression* consisting of a double-quoted string is considered a label or name that represents a state of the cell in question. This *condition_expression* is **TRUE** when that string or label represents the current state of the cell.

8.11.2.6 Condition expression operators

The *condition_expression* operators are detailed in Table 99.

Table 99—Condition expression operators

Operator	Example	Description
==	<i>L1</i> == <i>L2</i>	Is TRUE when the logical expression <i>L1</i> is equal to <i>L2</i> .
!=	<i>L1</i> != <i>L2</i>	Is TRUE when the logical expression <i>L1</i> is not equal to <i>L2</i> .
&&	<i>L1</i> && <i>L2</i>	Is TRUE when both the logical expressions <i>L1</i> and <i>L2</i> are TRUE .
	<i>L1</i> <i>L2</i>	Is TRUE when one or both the logical expressions <i>L1</i> and <i>L2</i> are TRUE .
^	<i>L1</i> ^ <i>L2</i>	Is TRUE when one but not both the logical expressions <i>L1</i> and <i>L2</i> are TRUE .
!	! <i>L1</i>	Is TRUE when the logical expression <i>L1</i> is FALSE .

8.11.2.6.1 Semantics for Z (high Z) state

If a pin has logic level **Z**, any condition including that pin is **FALSE** unless the condition explicitly enumerates **Z** (e.g., *pin* == **Z** or *ANYIN* == **Z**).

8.11.2.6.2 Semantics for X (unknown) state

If a pin has logic level **X**, any condition including that pin is **FALSE** unless the condition explicitly enumerates **X** (e.g., *pin* == **X** or *ANYIN* == **X**).

8.11.3 Condition expression operator precedence

Condition expressions are evaluated left to right. The precedence (from highest to lowest) is as follows:

-1
 ==
 ~
 @
 ()
 !
 &&
 ^
 ||
 !=
 ==

The group operator() can always be used to force groupings or expressions to override the default precedence order. See 8.11.2.1.1 and 8.11.2.1.2 for examples.

8.11.4 Condition expressions referencing pin states and transitions

A *condition_expression* can express references to states and to transitions. A *condition_expression* that contains no references to transitions is **TRUE** if and only if a transition occurred on at least one of the pins referenced in the *condition_expression* and the state evaluates to **TRUE**. A *condition_expression* that

contains references to transitions is `TRUE` if and only if those explicitly described transitions occurred and for the remaining pins not containing transition references transitions did not occur and the state evaluates to `TRUE`.

8.11.5 Semantics of nonexistent pins

A DCL model can be written for a varying number of interface pins. When an application associates an instance with a model, it supplies the actual interface pins for that instance. Interface pins that are declared in a model and not supplied by the application are called “nonexistent pins.”

An expression with a nonexistent pin shall behave as if the pin and the tightest binding binary logical operator adjacent to it were dropped from the expression.

Example

`00 && A01 && A02 && A03`

shall behave like

`A00 && A01 && A02`

if pin `A03` does not exist.

If both pins bound to a binary operator do not exist, the next highest level in the interpretation in the parse tree shall treat that subtree as a nonexistent pin. If all of the pins disappear, then the meaning of the expression shall be 1 if the top level operator is `&&` and 0 otherwise.

Example

<code>L && L'</code>	means L and L'
<code>nonexistent_pin && L'</code>	means L'
<code>L && nonexistent_pin</code>	means L
<code>nonexistent_pin && nonexistent_pin</code>	means nonexistent pin (except at the top, when it means 1)
<code>L L'</code>	means L or L'
<code>nonexistent_pin L'</code>	means L'
<code>L nonexistent_pin</code>	means L
<code>nonexistent_pin nonexistent_pin</code>	means nonexistent pin (except at the top, when it means 0)
<code>!(L)</code>	means not L
<code>!(nonexistent_pin)</code>	means nonexistent pin (except at the top, where it means 0)

9. Application and library interaction

The library communicates to the application a description of a cell's characteristics. Each cell's characteristics are categorized into domains. The use of domains narrows the scope of a cell's model.

9.1 behavior model domain

The behavior domain uses a graph to represent the Boolean functions of a cell. Each graph transferred during model elaboration of a cell represents an independent function performed by that cell.

9.2 vectorTiming and vectorPower model domains

The model elaboration process shall describe a cell's set of vector expressions in a graph format. Each vector that evaluates to `true` represents a cell state change. Each cell state change has associated with it a delay, slew, and check calculation. Once elaboration is complete, the application can evaluate each graph that evaluates to `true` and call the library for delay, slew, and check. The application requesting (for the library to calculate) delay, slew, or check shall provide the *pathData* pointer associated with the *DCM_PRIMITIVE_VECTOR_DELAY_TARGET* or *DCM_PRIMITIVE_VECTOR_CHECK_TARGET* to that library. Once the application requests vector delay and vector slew evaluations, the library shall return the early and late delay and slew values.

Vectors permit description of multiple transitions. For example, the following describes a sequence of transitions for a given from and to point:

Example

```
/*
 * 2 input AND gate vector power description
 */
model (AND2_vectorTiming) : defines (AND2.*.vectorTiming);

modelproc (AND2_vectorTiming) :
  do: vector(-|+A->-|+B->-|+Z)
    from(B) to(Z) propagate(rise->rise);

  do: vector(-|+B->-|+A->-|+Z)
    from(A) to(Z) propagate(rise->rise);
end;
```

The 1999 version of this standard only permitted the delay and output slew calculation to be functions of input slew at pin A. However, as the previous example indicates, delay and output slew calculations can be functions of more than two pins. Therefore, a library that contains *vectorTiming* or *vectorPower* models can call the application for input slews, output capacitance loads, and resistance on multiple pins via the *EXTERNAL* API *appRegisterCellInfo()*. In response to this call, the application shall provide slews and loads for the specified pins by calling the *EXPOSE* API *dpcmFillPinCache()* for those pins listed in the vector expression. The library then uses those slews and loads for computing the delay, slew, check, or power.

9.2.1 Power unit conversion

The library can represent its power modeling in either energy (Joules), power (Watts), or any other proprietary unit. The library needs to help ensure the appropriate energy units are passed for all the functions in the *power* and *vectorPower* domains. These units shall match the exponents returned by *dpcmGetRuleUnitToJoules* and *dpcmGetRuleUnitToWatts*.

9.2.2 Vector power calculation

The model elaboration process shall describe a cell's set of vector expressions in a graph format. Each vector represents either a cell state change or a steady state of a cell. Each vector has a power calculation associated with it. Once the model elaboration process for a cell is complete, the application has a set of vectors and their associated *pathData* pointers. The application can evaluate the graph for each vector to determine which occur during simulation. For those that evaluate to `true`, the application shall call *dpcmGetCellVectorPower* (see 10.23.13.15) for the power associated with each vector. The application shall provide the *pathData* pointer associated with the *DCM_PRIMITIVE_VECTOR_POWER_TARGET* when calling the library to calculate the power.

Once the application has set up the models of power vectors and determined the conditions described by the vector function graph are met, it can call the DPCM to calculate power for a given vector. The application shall call *dpcmGetCellVectorPower*. The application needs to pass the *cellData* information for the cell and the *pathData* for the vector.

To obtain slewrate and capacitance data for the instance, the DPCM shall call back to the application via the *EXTERNAL* API *appRegisterCellInfo()*. This function passes in three flags indicating the type of information being requested (capacitance, resistance and/or slew) and the types of pins for which the requested information is needed [i.e., input(s), output(s), bidirectional(s), or all]. This callback (*appRegisterCellInfo()*) enables the application to update (if necessary) the load and slew cache for this instance prior to the power calculation via the *EXPOSE* API *dpcmFillPinCache()*.

Use the following calls to request voltage and temperature parameters:

- *appGetCurrentRailVoltage*
- *appGetCurrentTemperature*

The process point is set by the application by calling *dpcmSetCurrentProcessPoint* (see 10.23.11.1.1).

The switching bits information is requested by calling *appGetSwitchingBits* (see 10.23.13.5).

10. Procedural interface (PI)

A standard PI is used for communication between an application and a compiled DPCM.

10.1 Overview

The functions that make up the PI are defined in three different logical components: the DPCM, the application and the *libdcmr*. Each of these components may consist of more than one compiled object, which are dynamically linked at run time, as needed.

10.1.1 DPCM

The following three categories of functions result from compilation of the DCL subrules of a technology library:

- The main calculation entry points that perform cell modeling and calculate delay, slew, and check are presented to the application automatically in the *DCMTransmittedInfo* structure as a result of the pointer exchange resulting from the call to *dcmRT_BindRule*, the DPCM primary entry point. These entry points map directly to the *MODELPROC*, *DELAY*, *SLEW*, and *CHECK* statements coded in the DCL source.
- Explicitly named *EXPOSE* functions (see 7.9.7.1) defined in the DPCM are made available to the application in a name/function-pointer table after the call to *dcmRT_BindRule*. These functions are explicitly defined in the DCL by the library developer. These functions can be called by the application to request information (such as calculated or default values) relating to the cell library.
- Run-time library functions (see 10.16.2) are implicitly available as a result of code generated automatically by the DCL compiler. No DCL statements directly map to these functions. Each of these functions are available for *dynamic linking* to the application.

10.1.2 Application

The following two categories of functions shall be defined in the EDA application code; the third category is optional:

- *EXTERNAL* function entry points are presented to the DPCM via name/function-pointer pairs in the *DCM_FunctionTable* argument in the call to *dcmRT_BindRule*, the DPCM primary entry point (see 10.25.4.5). The DPCM accesses these functions, using calls to the *EXTERNAL* functions in its subrules, for design-specific information (such as interconnect configuration and parasitics) required for accurate delay calculation.
- Modeling callback functions (see 10.16.6) are used by the DPCM as a result of the application's invocation of the *modelSearch* function to report back to the application information about timing arcs and propagation characteristics.
- Initialization functions that initialize the library run-time system and loads and links library modules to an application.

10.1.3 libdcmir

The DPCM-independent set of initialization functions, *dcmRT_BindRule* and *dcmRT_initRuleSystem* (see 10.16.2) shall be available for dynamic linking to the application. These key functions load and initialize a DPCM.

10.2 Control and data flow

A design goal of the DPCS is to isolate applications from requiring a detailed knowledge of timing models. However, applications shall conform to the control and data flow mandates of the PI, as shown in Figure 3.

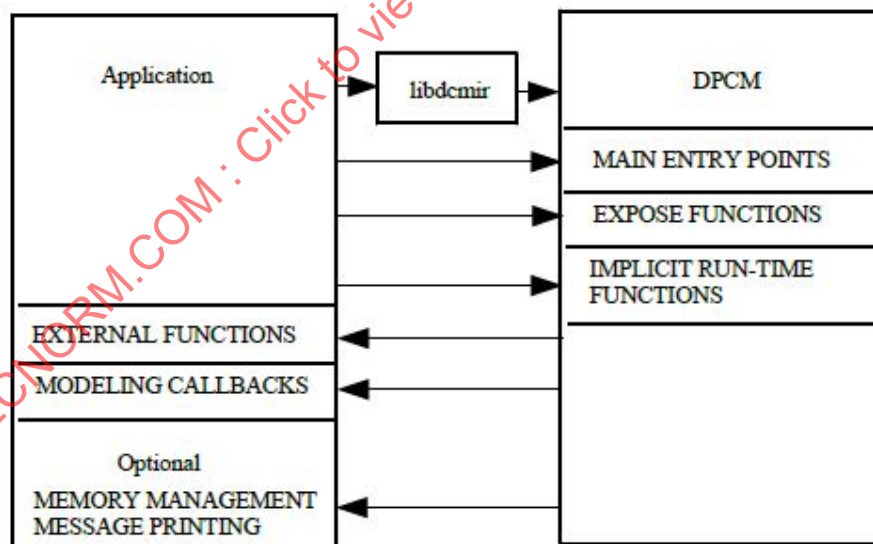


Figure 3—DPCM/application procedural interface

10.3 Architectural requirements

The interface requires integers to be 32 bits (or greater) to properly construct return codes (see 10.10.1).

10.4 Data ownership technique

Many of the DPCM PI functions take pointers to data as passed arguments (text strings, timing values, etc.). The DPCM assumes all data referenced across the interface is read only (by both the DPCM and the application).

The DPCS architecture separates the ownership of delay and power models from that of electronic designs. The application is assumed to "own" (store and fully understand) the design for which delay and power calculations are desired. The DPCM, on the other hand, "owns" the delay and power models and their evaluation process, but does not own -- and shall not cache -- any design-specific information, except where specifically identified.

10.4.1 Persistence of data passed across the PI

For calls across the PI, data only need to persist as follows:

- a) For data passed by an application to a DPCM-supplied function, the DPCM assumes the data persist only for the duration of the DPCM function's execution. The practical effect of this assumption is a DPCM does not store pointers to any data "owned" by the application.
- b) For data (other than structures and arrays that are not marked transient) returned by a DPCM function to an application, the DPCM assumes the data need to persist only until the next call to any DPCM PI function (including any recursive calls resulting from callbacks by the DPCM to the application). The practical effect of this assumption is when *persistence* is desired, an application shall, immediately after the function call, make its own copy of any data returned by a DPCM function.
- c) For arrays or structures returned by a DPCM function to an application that are not marked transient:
 - 1) The data contained in the structure or array data are read only unless otherwise specified.
 - 2) If the application claims the structure or array (see 10.25.1.4) the data persists until the application disclaims the array (see 10.25.1.5). When the structure or array is claimed, the application need not copy the array data and the DPCM shall not free the array data.
 - 3) If the application does not claim the array, the data only persists until the next call to any DPCM PI function or until the DPCM removes any outstanding claims it may have. In this case, the application shall copy the array data if the application requires this array data to persist.
 - 4) Arrays returned by the calls *dpcmGetCellPowerWithState*, *dpcmGetPinPower*, *dpcmSetInitialSet*, and *dpcmGetAETCellPowerWithSensitivity* shall not be claimed by the application. For these calls, the application shall copy the array data if the application requires this array data to persist.
- d) For structures and arrays marked transient:
 - 1) The application shall assume the data persist until the next call. If the application requires the data to be persistent, it shall make a copy of the structure and the data.

10.4.2 Data cache guidelines for the DPCM

The DPCM shall not cache any design-specific data, except where specifically identified. The pin cache is one instance where the standard directs the DPCM to cache information under the control of the

application. The pin cache is created and maintained by the DPCM but is updated by the application. The application shall call the DPCM to free the cache.

10.4.3 Application/DPCM interaction

This subclause describes a representative scenario for an application's use of a DPCM to perform timing delay calculations. By assumption, the application has a design and wants to evaluate design-specific timing. Not all of these steps need occur for all designs. A similar scenario is provided for an application's use of the DPCM for power calculation (see Clause 8.).

10.4.4 Application initializes message/memory handling

The DPCM allows the application to provide message handling functions (see 10.25.4.10). In addition, the application is required to register memory management functions (see 10.25.4.4), which shall be used by both the DPCM and the application.

10.4.5 Application loads and initializes the DPCM

The application first calls `dcmRT_InitRuleSystem` entry point (see 10.25.4.4) to initialize the DPCM. The application then calls `dcmRT_BindRule` to load the main entry point within the DPCM dynamically. Through this call, the application passes a table of function pointers (for the *EXTERNAL* functions the application has defined) to the DPCM and receives back a table of function pointers (for the *EXPOSE* functions the DPCM has defined). This handshaking establishes the PI between the application and the DPCM.

To enable an application to be independent of implementation-specific characteristics of a DPCM, the code that implements the calls `dcmRT_InitRuleSystem` and/or `dcmRT_BindRule` shall not be statically linked with the application.

10.4.6 Application requests timing models for cell instances

An application shall model a cell before calling any PI function that may use *pathData* or *cellData* in the Standard Structure. Timing models in a DPCM become accessible to an application after it calls the *modelSearch* function (see 10.27.1).

The application shall determine the association between instances (unique occurrences in the equivalent flattened design) and particular timing models. The association is often based on an instance's model name; an application shall use the *modelSearch* function to locate a timing model given its name.

10.5 Model domain issues

This subclause defines model issues across the PI.

10.5.1 Model domain selection

The DPCM may have one or more model domains that represent independent but related views of a technology, one for power, one for timing, and one for function or any combination of these. The DPCM may combine one or more of these together into a single domain. To prevent the duplication of calls within multiple model domains, it is the responsibility of the application to select the correct domain when responding to an interface call. Each *EXPOSE* function in this specification has a model domain associated with it. Each *EXPOSE* function may also require domain-specific *pathData* and *cellData* pointer values. It shall be the responsibility of the application to populate the Standard Structure *pathData* and *cellData* pointer fields from the proper domain before calling the *EXPOSE* functions.

10.5.2 Model domain determination

The model domain of the *pathData* and *cellData* pointers are determined at the time the cell was modeled. A cell modeled with the domain of timing shall have its *pathData* and *cellData* pointers valid for calls in the timing domain. A cell modeled in the power domain shall have its *pathData* and *cellData* pointers valid for calls in the power domain. If a cell in the library uses the asterisk (*) as its domain, the *cellData* and *pathData* pointers shall be valid for both the timing and power domains.

10.5.3 DPCM invokes application modeling callback functions

Applications shall be capable of dealing with DPCM timing models, which can be parameterized in a variety of potentially complex ways. For example, a timing model for a scan-sensitive latch may include certain timing arcs whose existence depends on the latch's functional "mode." There can be more than one timing model structure corresponding to a given timing model name, and timing models can be implicitly instance specific.

As part of its call to *modelSearch*, the application supplies the *cellName* (see 10.17.1.1) and the input and output pins. Bidirectional pins, which are those pins on a cell that act as both an input and output, shall appear as an entry on both the list of input pins and the list of output pins. After the call to *modelSearch* but before control returns to the application, the DPCM elaborates the timing model by evaluating the "setup" code in the model. Once the timing arcs and other structures of the timing model have been determined for this call to *modelSearch*, the DPCM conveys that information to the application by making a sequence of calls to the application-supplied modeling functions (see 10.27.6 and 10.27.12). The information conveyed is sufficient for an application to determine, for example, which input pins connect to which output pins and which input signal transitions cause which output signal transitions.

The application shall save whatever part of this structural information it needs for future use; the DPCM PI does not have any support for later interrogation of the timing model structure. At the very least, an application shall save the timing arc-specific or pin-specific values of the *pathData* to the *pathData* pointer field, the delay matrices, the test matrices, and the cell-specific value of the *cellData* pointer fields in the Standard Structure (see 10.12) because those values are necessary for any subsequent application request for a delay calculation from the DPCM. To prevent ambiguity, the DPCM shall not create a model for a cell that contains more than one *pathData* per pin or arc. Bidirectional pins are considered two pins: one acting as an input and one acting as an output. Pins whose delay matrix is not zero shall be taken to mean that the delay matrix controls the propagation properties for all segments radiating to or from that pin.

NOTE—An application can determine the nature of any reasonable timing model parameterization because the model's "setup" code would have to call application-supplied functions to determine parameter values, and the application could keep track of whether any such functions were called during *modelSearch*. Given this knowledge, an application can efficiently reuse previously elaborated models. Without this special care, however, an application shall call *modelSearch* for every instance in the design (even if any particular timing model was previously elaborated).

10.5.4 Application requests propagation delay

After an instance has been modeled, an application typically wants delay values from the DPCM for each combination of signal transitions and paths in each instance of the design. This requires iteration of the following simple request:

- Given a timing model, an instance of a cell, and a pair of pins and state transitions, get the pin-to-pin propagation delay.

To obtain a propagation delay value for either an arc of a cell or a net, the application calls the *delay* function see (10.26.1) and passes the following parameters using the DCL *Standard Structure*:

- A "handle" for the specific instance (BLOCK) (see 10.11)

- The name of the instance's timing model (CellName)
- The “from” pin (FROM_POINT)
This argument is a pointer to an application-created pin structure (see 10.27.6).
- The “to” pin (TO_POINT)
This argument is a pointer to an application-created pin structure (see 10.27.6).
- Input slew value (EARLY_SLEW and LATE_SLEW)
- Input (“from” pin) signal transition (SOURCE_EDGE and SOURCE_MODE)
- Output (“to” pin) signal transition (SINK_EDGE and SINK_MODE)
- Value for pathData pointer (PATH_DATA) (see 10.27.13).
This argument was originally passed by the DPCM to the application during model elaboration (see 10.5.3).
- Value for cellData pointer (CELL_DATA)
This argument was originally passed by the DPCM to the application during model elaboration (see 10.5.3).

The call to *delay* results in the DPCM invoking the *DELAY* function defined for that particular propagation in the *MODELPROC* for the cell. This may result in a simple numerical calculation, a reference to a *TABLE* of coefficient data, and/or calls to other functions within the DPCM or *EXTERNAL* functions that the application shall support. The application shall retain the *pathData* pointer given during modeling for use when requesting delays. arc-specific *pathData* pointers created for delay segments internal to the cell are associated with those arcs. Pin-specific *pathData* pointers created on a pin of a cell (due to an *INPUT* or *OUTPUT* statement containing a propagation clause) shall be used for all interconnect delay arcs that originate or terminate on that pin (see 10.27.13 for information on interconnect delay calculation).

10.5.5 DPCM calls application EXTERNAL functions

The expression for propagation delay depends on the input pin's slew value and the output pin's total load capacitance. The DPCM is passed a value for input slew via the Standard Structure. To get a value for loading capacitance, the DPCM calls an application-supplied function, passing to the application the same Standard Structure the DPCM received from the application; see 10.5.4.

10.6 Reentry requirements

In some cases, a call to an *EXTERNAL* or *EXPOSE* can result in nested calls to the same function. Both *EXTERNAL* and *EXPOSE* functions (as well as any functions they might reference) shall be coded for reentry in order to cope with such situations.

10.7 Application responsibilities when using a DPCM

This subclause details the responsibilities an application shall have when using a DPCM.

10.7.1 Standard Structure rules

The application is required to establish and maintain Standard Structures. The DCL Standard Structure is a collection of commonly used information and predefined variables. A pointer to this structure is passed as the first argument in most function calls (see 10.10.2). Most PI functions require values to be initialized in one or more of the DCL Standard Structure fields (see 10.12).

An application request for a DPCM function frequently leads to the DPCM making callbacks to the application for more information. As long as the subsequent function calls involve the same cell instance, the *DCL Standard Structure* remains unaltered and the pointer is merely passed in during the next function call. If, however, the application calls a function for a different cell instance during this callback process, then the application shall use a different *Standard Structure* for those function calls that reference different cell instances.

NOTE—The number of extra Standard Structures required for this approach is small (generally one or two).

10.7.2 User object registration

The DPCM has a method `dcmRT_registerUserObject` (see 10.25.4.20) for allowing an application to register an object with the *Standard Structure*. This is purely optional but can be useful if the application has to allocate memory within a function that loses control before the memory can be freed. Registered objects are under the control of the application and can be freed on demand by the application. The DPCM shall delete registered user objects when the *Standard Structure* they are registered with is deleted.

A registered user object shall conform to the following:

- It shall be a structure whose first element is a pointer to a function that deletes the object when called.
- The delete function shall accept only one parameter, the address of the user object.

10.7.3 Selection of early and late slew values

When the application calls the DPCM to compute delays and slews, the DPCM returns two numbers—the “early” and “late” values of the computed quantity. The use of two values allows the delay calculation process to account for the convergence of paths through the design along which different slew values are propagated. Slew convergence occurs at the outputs of cells that have delay arcs from multiple inputs and at the inputs of cells connected to interconnects with more than one driver.

Example

- An AND gate with inputs A and B and output Y shall have delay arcs from A to Y and from B to Y. Signals arriving at A and B typically have different slew values, depending on the capacitance loading characteristics of the drivers of those interconnects. If the cell is modeled such that the slew at the output is dependent on the slew at the input, then two values shall be computed for the slew at Y, one due to changes at input A and the other due to changes at input B. The slews from A and B can be said to have “converged” at Y.
- For wider gates, clearly more slew values may converge at the output. A value or values shall be chosen from the converging slews for propagation over the interconnect to inputs driven by this output to avoid the explosion of the number of slew values being considered in the design.

Many delay calculation systems chose one slew value using some algorithm, such as choosing the arithmetic mean. This results in the loss of considerable information and potentially in less accurate results. The DPCM uses “early” and “late” values to bracket the range of values converging and retain accuracy while limiting the amount of information that shall be propagated through the design.

The application is responsible for choosing the early and late slew values to propagate forward in the design from the converging values computed by the DPCM. For example, a static timing analysis application may choose to propagate as the early slew value the one associated with the earliest arriving signal and to propagate as the late slew value the one associated with the latest arriving signal. By contrast, a batch delay calculator or simulation application, which has no notion of arrival times, may choose to propagate the average of all converging values; in this case, early and late slew values shall be the same. If the application propagates as the early value the one that results in the smallest delay at the next level of

logic, and as the late value the one that results in the largest delay, then, it shall know whether delay values vary in the same sense as slew values or in the opposite sense to them.

From the perspective of the DPCM, the use of early and late values means all calculations that depend on slew shall be repeated for both values. The subrule may detect that early and late slew values (provided by the application) at the input to a path are sufficiently similar so the computation can be performed once and the result presented as both early and late result values. Otherwise, if the result of the calculation depends on slew, it shall be done twice (once using the early value and once using the late value), and the results presented as the early and late result values, respectively.

10.7.4 Semantics of slew values

The DPCS specification defines the precise, real-world semantics of the slew values passed between the application and the DPCM to represent the rising and falling logic signals so delay and timing check values may be computed as a function of these signal shapes. An application can call to determine whether the slew values have the dimensions of time, and therefore represent the rise and fall time, or have the dimensions of time/voltage; in which case, they represent slew rate.

NOTE—The DPCS does not specify how different applications select and propagate converging slew values. Therefore, users of the same DPCM with different applications may observe differences in the computed timing properties of their design. The goals of the DPCS include providing *consistent* timing information to various applications, but it may not be feasible to attain *identical* final results. The user can be confident residual differences are due to different application assumptions or capabilities and not to discrepancies in the data used for timing calculation.

10.7.5 Slew calculations

It is up to the technology modeler to define whether or not the slew at the input of a cell effects the slew value at the output of a cell. DCL allows the library developer to define slew equations for timing arcs within a cell, as well as for the interconnects between cells. DCL also allows the creation of default slew equations for those situations where the specific equation has not been specified.

An application shall assume there is a slew equation for all delay arcs and simply request a slew be calculated based on that assumption. The DPCM shall determine the correct slew equation to use, including the default if one has not yet been specified. The application, when calling for a slew calculation, shall supply the appropriate slew value in the *Standard Structure*. This can be used by the library developer in the slew equation, if needed.

If the application is coded in this manner, the slew can be propagated or not at the library developer's discretion.

10.8 Application use of the DPCM

This subclause details how an application interacts with the DPCM.

10.8.1 Initialization of the DPCM

A running application first calls the function *dcmRT_InitRuleSystem*, which initializes the library run-time library system and prepares it for dynamic linking.

The four parameters passed to *dcmRT_InitRuleSystem* are a function pointer to the application's preferred malloc, free, and realloc function pointers as well as a pointer to where *dcmRT_InitRuleSystem* can place an integer return value.

dcmRT_InitRuleSystem returns the system's initial *Standard Structure*. All other application interactions shall require a *Standard Structure*.

10.8.1.1 Standard Structure management

The DPCM maintains many contexts. A context is a combination of a space and a plane. The application shall assume that one context is independent from another. The application shall assume that one context can operate simultaneously with other contexts either at the request of the application or the library.

The context is identified in the Standard Structure. When a Standard Structure is created, it contains the identification of the context that created it. Any operations requested of the DPCM by the application where the Standard Structure passed in as the first argument shall cause the DPCM to perform the requested operations on the context identified in that Standard Structure.

The Standard Structure returned from *dcmRT_InitRuleSystem* shall contain the run time's initial context. The initial context is only valid for creating other contexts by calling *dcmRT_BindRule*.

The application shall not call for any service except *dcmRT_BindRule* with the Standard Structure returned by *dcmRT_InitRuleSystem*.

10.8.1.2 Tech_family

Each space has one or more tech_families. A tech_family is a set of DPCM modules containing the same TECH_FAMILY identifier. Each tech_family is independent of any other tech_family within the same space.

When an application requests an operation of a specific tech_family, the application sets the corresponding tech_type in the Standard Structure before making the call.

To facilitate cooperation between tech_families within the DPCM's system of modules the DPCM may temporarily change the tech_family setting in the Standard Structure. The privilege of changing the tech_family setting is independent of whether or not the Standard Structure is const or not.

10.8.2 Context creation

A context is a system of executable modules called a “space” and a set of concurrent operations each being performed in a separate area of memory called a “plane.” Each system is created by a call to *dcmRT_BindRule*, which creates the space and the initial plane. For each concurrent operation the application wants performed on a space, the application shall create a additional plane to separate the concurrent operations. The application creates a new plane on a space by calling *dcmRT_newPlaneInSpace*. The application shall not attempt concurrent operations on the same context.

dcmRT_BindRule dynamically loads a subrule system and returns a Standard Structure identified with the newly created context.

10.8.3 Dynamic linking

The process of dynamic linking the library and space creation are combined into a single operation. When *dcmRT_BindRule* is called, the required parameters are passed to the run-time library such that any external references from the library to the application can be resolved. In addition, the library also resolves any interdependencies between library modules and tables.

When the run-time library returns from the call to *dcmRT_BindRule*, the application has been given *DCMTransmittedInfo* structure pointer containing the EXPOSE entry points made available to the application from the library called the “xmit block.”

The *xmit* block contains the address of a location for the DPCM to store a *DCMTransmittedInfo* structure pointer. The *DCMTransmittedInfo* (see 10.13) is a structure containing pointers to the DPCM functions *modelSearch*, *delay*, *slew*, and *check*, followed by a table of *EXPOSE* pairs. Each *EXPOSE* pair consists of a string containing the name of the *EXPOSE* (as it is defined in the subrule) and a pointer to that function's entry point. The DPCM shall fill in this structure with its function addresses.

To enable the library to perform its dynamic linking a pointer to a *DCM_FunctionTable* structure containing pairs of PI *EXTERNAL* names and pointers to the functions implementing them, the application is passed as an argument to *dcmRT_BindRule*. It is the application's responsibility to create this structure.

10.8.3.1 Linking order

Subrules are dynamically loaded and linked in the order their references are encountered. The subrules are scanned from beginning to end, in a depth-first fashion, to locate other subrule references.

EXPOSE entry points with the same name (originating in separate subrules) shall be linked together in a chain (see 10.8.3.1). Expose chaining is a process that occurs when two or more *EXPOSE* functions are defined with the same name within the same *TECH_FAMILY*. A separate *EXPOSE* function definition can potentially exist in each subrule. These *EXPOSE* functions are linked together in a chain in the order the individual *EXPOSE* function definitions were loaded.

The *IMPORT* prototype of an *EXPOSE* function links to the first *EXPOSE* function within the chain. When the application calls the chained *EXPOSE* function, it references this first *EXPOSE* function in the chain. If this function returns a return code of zero (0), control returns to the caller (the application). If this function returns an error return code with severity less than 3 (severe) and the return code is not the value returned by *dcmHardErrorRC*, the next *EXPOSE* function in the chain is called with the same *PASSED* parameters.

This process continues until one of the following occur:

- The current *EXPOSE* function returns a zero (0) return code
- The current *EXPOSE* function returns a return code with severity 3 or greater
- The current *EXPOSE* function is the last function in the chain (in which case, it returns its return code back to the application).

10.8.4 Subrule initialization

Each subrule in a DPCM is initialized in the order in which it is loaded. This initialization involves the following actions:

- Resolution of references to *EXTERNAL* function defined within the application
- Static *TABLE* s, if any, defined within the subrules are loaded into memory
- Execution of the *LATENT_EXPRESSION* functions

Any number of these functions may be used per subrule. These functions are executed after all subrules have been loaded but before control has been returned to the application. The *LATENT_EXPRESSION* functions within a subrule are executed in the order seen in the subrule. This capability gives the library developer the opportunity to accomplish initialization tasks, such as initializing variables. Once all subrules have been loaded, these functions are executed in the order the subrules were loaded.

The following rules apply to the *LATENT_EXPRESSION* function:

- The function name shall be *LATENT_EXPRESSION*.
- Only one *LATENT_EXPRESSION* is allowed per subrule.

- The function shall not have *PASSED* parameters.
- The function shall be one of the following types: *ASSIGN*, *CALC*, *INTERNAL*, or *EXPOSE*.
- The *LATENT_EXPRESSION* function may reference any other legal DCL statement.

10.8.5 Use of the DPCM

When *dcmRT_BindRule* is called, the DPCM loads the remaining subrules specified for this context, cross links all the *EXPORT* and *IMPORT* statements, and uses the *DCM_FunctionTable* to link the application *EXTERNAL* functions to the corresponding *EXTERNAL* functions listed in the DPCM. It then fills in the field of the *DCMTransmittedInfo* structure pointed to by the address specified by the last argument to the *dcmRT_BindRule* call.

10.8.6 Application control

Control is returned to the application that then does the following:

- Initializes its function pointer variables for the *modelSearch*, *delay*, *slew*, and *check* functions to the addresses provided by the DPCM in the corresponding fields of *DCMTransmittedInfo*.
- Initializes its function pointer variables to the DPCM services it requires from the table of *EXPOSE* functions provided in *DCMTransmittedInfo*. The calls *dcmQuietFindFunction* and *dcmFindFunction* are convenience tools to find the location of the function pointer given an *EXPOSE* name.

10.8.7 Application execution

After subrule loading, run-time linking and subrule initialization has been completed by the DPCM, control is returned to the application. The application then initiates PI function calls to the DPCM. These function calls execute run-time library functions or portions of subrules (as required by the application request), and then return to the application.

The application drives execution in the DPCM. The application shall call the *modelSearch* function (see 10.27.1) prior to calling functions that require *pathData* or *cellData* Standard Structure fields (see Table 117). The *MODELPROC* function (see 7.16.3) describes to the application the correct delay, slew, and check formulas to use. If a cell has not been modeled, the “default” delay or slew is used (if one is defined). The application typically asks the DPCM to model each specific cell instance in a design (by calling the *modelSearch* function). The DPCM identifies the corresponding *MODELPROC* to use for this task from the *MODEL* function with the given cell name in its *DEFINES* clause.

The DPCM models a cell by describing specific static timing arcs of the cell to the application through a system of callback functions (see 10.27). This description might include interconnectivity, which delay equations to use, and which edges are propagated. *SUBMODEL* procedures within the DPCM generally process a portion of a cell’s description and may be executed as a result of *MODELPROC* execution. The functions contained within a *MODELPROC* or *SUBMODEL* are generally executed in the order encountered but within the control of decision logic intrinsic to the function.

10.8.8 Termination of DPCM

When an application is finished using a space and all its associated planes, it may call *dcmRT_UnbindRule* (see 10.25.4.4). *dcmRT_UnbindRule* invokes all *TERMINATE_EXPRESSION* functions, if defined, in each loaded subrule, in the opposite order in which the subrules were loaded. This capability gives the library developer the opportunity to accomplish termination tasks, such as freeing memory.

There may be any number of `TERMINATE_EXPRESSION` functions within a subrule. Each `TERMINATE_EXPRESSION` function is executed in the opposite order in which they were found within each subrule.

In the case where the application comes to a normal termination (by calling *exit*) without calling *dcmRT_UnbindRule*, all the `TERMINATE_EXPRESSION` functions shall be executed in the opposite order in which the subrules were loaded. After the execution of these functions, the normal termination process shall continue.

10.9 DPCM library organization

This subclause highlights the DPCM library organization.

10.9.1 Multiple technologies

A subrule may load other subrule that is not of the same *TECH_FAMILY*. When a subrule is being loaded, if its *tech_family* is already in the current context, this subrule becomes a member of that *tech_family* already loaded; otherwise, a new *tech_family* grouping is started that is a peer to the subrule doing the loading.

Subrules that do not contain a specific *tech_family* name are given the name `GENERIC`.

Interactions between an application and a potentially multiple-technology DPCM are described in Table 100.

Table 100—Interaction between multiple technologies and application

Number of technologies	Technology names visible to application	Default DPCM visibility to application
1	Loaded <i>TECH_FAMILY</i> name	Everything
>1	All loaded <i>TECH_FAMILY</i> names	First <i>tech_family</i> loaded

10.9.2 Model names

A fully qualified model name consists of the three strings *cell*, *cellQual*, and *modelDomain*.

10.9.3 DPCM error handling

When the DPCM detects errors of severity 0, 1, or 2 (see Table 101), it shall perform *DEFAULT* actions defined by the function. The DPCM itself, however, shall never generate an error with a severity less than 2 back to the application. When the DPCM detects a return error of severity 3 or 4, it shall terminate all functions in the current expose chain (see 10.8.3) and return this error to the application at the original calling function. The *DEFAULT* actions shall not be processed in this case.

Severity level 2 indicates a local function failure. If the caller has a *DEFAULT* clause, it shall fire and the caller shall return a 0 if the *DEFAULT* clause executed correctly; otherwise, its failure code shall be passed up the call chain. If there is no *DEFAULT* clause, then the original level 2 return code (severity and message number) shall be passed up the call chain to the top level call, or until a successful *DEFAULT* clause fires, setting the return code to zero.

Severity level 3 indicates this particular call chain from the DPCM has failed; however, subsequent calls to the DPCM are still possible. Severity level 4 indicates a catastrophic failure has occurred in the DPCM (such as a memory allocation error), and the application shall not attempt to reenter this DPCM.

The DPCM shall never terminate the process (call *exit(2)*).

10.10 C level language for EXPOSE and EXTERNAL functions

The following C language interface conventions shall be honored by applications interfacing with PI *EXTERNALs* or *EXPOSEs*.

10.10.1 Integer return code

Functions return an integer code, which indicates the following:

- Zero means the function completed successfully.
- Nonzero indicates one of several possible conditions.
- The most significant byte of the return code is set according the severity of the condition defined by Table 101.

Table 101—Return code most significant byte

Decimal severity	Meaning
0	Informational
1	Warning
2	Error
3	Severe
4	Terminate

- The least significant bytes set a message number defined by Table 102.

Table 102—Return code least significant bytes

Decimal range	Meaning
1 through 10 000	Reserved for DCL compiler use
10 001 and above	Available for application use

Application developers need to set the return codes according to the conventions described above or unpredictable results may occur. The tactic of returning a -1 or other negative value shall be avoided.

10.10.2 The Standard Structure pointer

The DCL Standard Structure (see 10.12) contains the frequently used information and predefined variables of a DPCM. The Standard Structure pointer shall always be passed as the first argument to the DPCM (in an *EXPOSE* call) and be expected as the first argument by the application (in its definition of an *EXTERNAL*), even if none of the structure variables are being used in that particular function.

10.10.3 Result structure pointer

All *EXTERNAL* and *EXPOSE* functions return value(s) through a result structure the address of which shall always be passed as the second argument to the function. It is the responsibility of the caller to allocate the memory required for the result structure.

NOTE—This result structure is different from the integer return code described in 10.10.1 .

10.10.4 Passed arguments

The PASSED arguments (as listed in the function declaration) follow the results pointer.

The types for the results structure and the passed parameters shall conform to data types as shown in Table 103. The definitions for these data types are available in the header files *dcmwords.h* and *std_stru.h*.

10.10.5 DCL array indexing

A DCL array with a n element dimension shall be indexed from 0 to $n - 1$. The first data element of an array shall have index 0.

10.10.6 Conversion to C data types

The C data types for each function's result and passed type names are shown in see Table 103. The *std_stru.h* header file provides these definitions.

Table 103—Data types defined in DCL and C

DCL Notation	std_stru.h #define	ISO C data type
INT or int	DCM_INTEGER	Int
CHAR or char	DCM_CHARACTER	Char
SHORT or short	DCM_SHORT	Short
LONG or long	DCM_LONG	Long
STRING or string	DCM_STRING	char *
NUMBER or number	DCM_FLOAT or DCM_DOUBLE	float or double
DOUBLE or double	DCM_DOUBLE	Double
FLOAT or float	DCM_FLOAT	Float
PIN or pin A pointer to an application-private data structure with a first field type char *.	DCM_PIN	char **
PINLIST or pinlist A 0-terminated array of PINs.	DCM_PINLIST	char ***
VOID or void A pointer to an indeterminate data structure.	DCM_VOID	void *
Structure A pointer to a structure of the defined type.	DCM_STRUCT	void * casted to the compatible application type. Applications shall not attempt to free these pointers. Applications shall only claim or disclaim these as appropriate.
Array A pointer to an array of the type contained in the array.	DCM_ARRAY	void * casted to the compatible application type. Applications shall not attempt to free these pointers. Applications shall only claim or disclaim these as appropriate.

10.10.7 include files

An application shall include the header files shown in see Table 104 (as needed to compile correctly) to obtain *typedef* and other necessary declarations.

Table 104—Header files

Header file	Definition
dcmdebug.h	This header file contains code necessary for uniform implementation of debugging facilities in the DPCM.
dcmintf.h	This header file contains the FindFunction declarations, environment variable names (UNIX), or user variable names (Windows NT), as well as the definition for the DCMTransmittedInfo structure (see 10.13).
dcmload.h	This header file defines how subrules are loaded.
std_macs.h	This header file contains macro definitions of the predefined variables that simplify access to fields in the DCL Standard Structure. It also contains edge and mode enumeration values, and macros for RESULT variables of INTERNAL functions.
std_stru.h	This header file contains the Standard Structure definition (see 10.12) and the typedefs used by many of the PI functions.
dcmgarray.h dcmpltfm.h dcmstate.h dcmstab.h	Miscellaneous header files containing defines used by the other header files.

To help ensure interoperability, header files shall not be modified.

10.11 PIN and BLOCK data structure requirements

The PI uses a common data structure to represent electrical nodes (ports and internal signals) in timing models and pins on instances of cells. The DPCM expects the application to create this data structure, including all the information the application may need to process the pin. The DPCM only requires the structure start with a character pointer (*char **) with the name of the node; the application is free to allocate additional space for its own specialized information. The diagram shown in Figure 4 illustrates the relationship between *PINLIST*(s), *PIN*(s), and *STRING*(s).

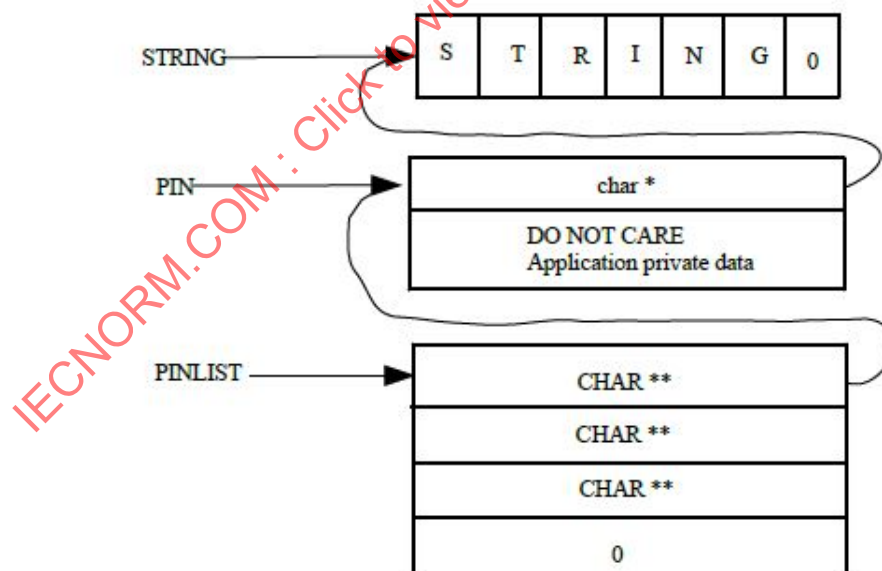


Figure 4—PIN and PINLIST

The application shall store enough private data in a PIN's data structure to be able to answer `appGetHierPinName` (see 10.18.6.3), `appGetHierBlockName` (see 10.18.6.4), and `appGetHierNetName` (see 10.18.6.5).

10.12 DCM_STD_STRUCT Standard Structure

Most PI functions exchange information in fields of a DCL Standard Structure (*DCM_STD_STRUCT*). The *DCM_STD_STRUCT* pointer is the first argument passed to most functions (see 10.17). The application developer shall not alter the definition of the *DCM_STD_STRUCT*.

Most of the important fields in the *DCM_STD_STRUCT* can be accessed using DCL reserved words or C define names. The predefined variable names can be used in DCL subrules and define names can be used in application source that includes the *std_stru.h* and *std_macs.h* header files (or from in-line C code in a DCL subrule) as shown in Table 105, providing that the application names its Standard Structure variable “STD_STRUCT.”

Table 105—Predefined macro names

#define (used within C source)	Description
DCM__BLOCK	Identifies the instance of a cell.
DCM__CALC_MODE	Gives access to the calcMode, which can be the best case, worst case, or nominal.
DCM__CELL	Cell is the name of the cell
DCM__CELL_QUAL	Gives access to the cellQual, which is a qualifier for the name of the cell.
DCM__CKTTYPE	Circuit type flag, set in a PROPAGATE or COMPARE clause.
DCM__CLKFLG	Set by the DPCM during modeling time and queried by the application. The Standard Structure field in the path data control cell holds the value.
DCM__EARLY_SLEW	Gives access to slew.early.
DCM__FROM_POINT	Gives access to fromPoint. Represents a pointer to the sourcePin of the starting point of an arc.
DCM__FROM_POINT_PIN_ASSOCIATION	Gives access to fromPointPinAssociation. Represents a pointer to the sourcePin of the starting point of an arc. FROM_POINT_PIN_ASSOCIATION shall only be used during the modeling of an arc.
DCM__INPUT_PINS	Gives access to inputPins. Represents an array of pointers of type PIN. The only member of the PIN type known to the DPCM is the first member, which is the pin name.
DCM__INPUT_PIN_COUNT	Gives access to inputPinCount. Represents the count of the input pins in the inputPins array.
DCM__LATE_SLEW	Gives access to slew.late.
DCM__MODEL_DOMAIN	Gives access to the modelDomain, which is set to either timing or power.
DCM__MODEL_NAME	Gives access to model_name. Names the DCL Model Procedure currently in use.
DCM__NODES	Gives access to nodes. Represents an array of pointers of type PIN. The only known member of the PIN type is the first member, which is the pin name.
DCM__NODE_COUNT	Gives access to nodeCount. Represents the count of the input pins in the nodes array.
DCM__OUTPUT_PINS	Gives access to outputPins. Represents an array of pointers of type PIN. The only known member of the PIN type is the first member, which is the pin name.
DCM__OUTPUT_PIN_COUNT	Gives access to outputPinCount. Represents the count of the output pins in the outputPins array.
DCM__PATH	Identifies a path by its name. This predefined variable accesses the pathData within the Standard Structure and is private to the DPCM. The application is only required to store the path data block.
DCM__PATH_DATA	Pointer to path object.

#define (used within C source)	Description
DCM__PHASE	Gives access to phase. This is a combination of the SOURCE_EDGE and SINK_EDGE. When the SOURCE_EDGE and the SINK_EDGE are the same value, PHASE is set to I. If they are not the same value, PHASE is set to O.
DCM__REFERENCE_EDGE	Gives access to sourceEdge and allows the library developer to utilize the reference edge within DCL statements. This generally occurs in TEST and TABLEDEF statements. The application treats this as though it is the SOURCE_EDGE.
DCM__REFERENCE_MODE	Gives access to sourceMode and allows the library developer to utilize the reference edge within DCL statements. This generally occurs in TEST and TABLEDEF statements. The application treats this as though it is the SOURCE_MODE.
DCM__REFERENCE_POINT	Allows the library developer to utilize the reference point within DCL statements. This generally occurs in TEST and TABLEDEF statements. The application treats this as though it is the FROM_POINT.
DCM__REFERENCE_POINT_PIN_ASSOCIATION	
	Allows the library developer to utilize the reference point pin association within DCL statements. REFERENCE_POINT_PIN_ASSOCIATION shall only be used during the modeling of a test arc.
DCM__REFERENCE_SLEW	Gives access to slew.early. From the application's perspective, it is the same data field as EARLY_SLEW.
DCM__SIGNAL_EDGE	Gives access to the computed signalEdge. Allows the library developer to utilize the reference edge within DCL statements. This generally occurs in TEST and TABLEDEF statements. The application treats this as though it is the SINK_EDGE.
DCM__SIGNAL_MODE	Gives access to sinkMode and allows the library developer to utilize the reference edge within DCL statements. This generally occurs in TEST and TABLEDEF statements. The application treats this as though it is the SINK_MODE.
DCM__SIGNAL_POINT	Allows the library developer to utilize the reference point within DCL statements. This generally occurs in TEST and TABLEDEF statements. The application treats this as though it is the TO_POINT.
DCM__SIGNAL_POINT_PIN_ASSOCIATION	
	Allows the library developer to utilize the signal point pin association within DCL statements. SIGNAL_POINT_PIN_ASSOCIATION shall only be used during the modeling of a test arc.
DCM__SIGNAL_SLEW	Gives access to signalSlew. From the application's perspective it is the same data field as LATE_SLEW.
DCM__SINK_EDGE	Gives access to sinkEdge. This is the transition type for the signal on the load pin. The sinkEdge value is represented by the enumeration of rise, fall.
DCM__SINK_MODE	Gives access to the timing mode for the load pin sinkMode. The sinkMode value is represented by the enumeration of early, late, or terminate.
DCM__SOURCE_EDGE	Gives access to sourceEdge. This is the edge type for the signal on the source pin. The sourceEdge value is represented by the enumeration of rise, fall.
DCM__SOURCE_MODE	Gives access to sourceMode. The sourceMode value is represented by early, late, both early and late, terminate, or both edges terminated.
DCM__TO_POINT	Gives access to toPoint. Represents the end point of an arc.
DCM__TO_POINT_PIN_ASSOCIATION	
	Gives access to toPointPinAssociation. Represents the end point of an arc. TO_POINT_PIN_ASSOCIATION shall only be used during the modeling of an arc.
DCM__CELL_DATA	Pointer to cell object.
DCM__CYCLEADJ	Gives access to the pathdata->cycleadj field of the Standard Structure. This field is set by the propagation sequence of a PATH statement.

#define (used within C source)	Description
DCM__EARLY_MODE	Gives access to string value of the sourceMode field in the Standard Structure.
DCM__LATE_MODE	Gives access to string value of the sinkMode field in the Standard Structure.
DCM__EARLY_MODE_SCALAR	Gives access to enumeration value of the sourceMode field in the Standard Structure.
DCM__LATE_MODE_SCALAR	Gives access to enumeration value of the sinkMode field in the Standard Structure.
DCM__SOURCE_MODE_SCALAR	
	Gives access to enumeration value of the sourceMode field in the Standard Structure.
DCM__SINK_MODE_SCALAR	Gives access to enumeration value of the sinkMode field in the Standard Structure.
DCM__CALC_MODE_SCALAR	Gives access to enumeration value of the CalcMode field in the Standard Structure.
DCM__DELAY_FUNC	Gives access to delay function associated with the pathData pointer.
DCM__SLEW_FUNC	Gives access to slew function associated with the pathData pointer.

For the C defines to function properly, C code shall use the name *std_struct* for the pointer to the *DCM_STD_STRUCT* because all the defines are of the form (*std_struct -> XXX*).

10.12.1 Alternate semantics for Standard Structure fields

Some DCL Standard Structure fields may have different meanings depending on the PI function called. For example, when the application is requesting a slew or delay calculation, the slew structure's fields represent the early and late slew values. When the application is requesting a check calculation, those same fields represent the values for the reference and signal slew, respectively. For programming convenience, the *REFERENCE_SLEW* and *SIGNAL_SLEW* defines in the *std_macros.h* header file specify the same Standard Structure field as *EARLY_SLEW* and *LATE_SLEW*. It is the application's responsibility to help ensure the values set in the DCL Standard Structure are appropriate for the context of the function being called.

Table 106 lists the different names used for the same fields in the DCL Standard Structure under the corresponding context headings.

Table 106—Alternate semantics for Standard Structure fields

Standard Structure field name	Delay and slew calculation	Check calculation
slew.early	EARLY_SLEW	REFERENCE_SLEW
slew.late	LATE_SLEW	SIGNAL_SLEW
fromPoint	FROM_POINT	REFERENCE_POINT
fromPointPinAssociation	FROM_POINT_PIN_ASSOCIATION	REFERENCE_POINT_PIN_ASSOCIATION
toPoint	TO_POINT	SIGNAL_POINT
toPointPinAssociation	TO_POINT_PIN_ASSOCIATION	SIGNAL_POINT_PIN_ASSOCIATION
sourceMode	SOURCE_MODE	REFERENCE_MODE
sinkMode	SINK_MODE	SIGNAL_MODE
SourceEdge	SOURCE_EDGE	REFERENCE_EDGE
SinkEdge	SINK_EDGE	SIGNAL_EDGE

10.12.2 Reserved fields

The Standard Structure has several unused fields for internal use, future expansion, and/or elimination of migration impact. These fields include *reserved1*, *reserved2*, *reserved3*, and *reserved4*. Do NOT use these fields!

10.12.3 Standard Structure value restriction

Standard Structure fields of type string shall not be set to the string "*" by the application.

10.13 DCMTransmittedInfo structure

This structure, as defined in the *dcmintf.h* header file, is used to support dynamic linking to the application of explicit *EXPOSE* functions (see 10.31 and implicit Calculation and Modeling functions 10.16.5 and 10.16.6) within the DPCM.

When the application first calls the DPCM entry point (which *dcmRT_BindRule* provides), it passes a pointer to a *DCMTransmittedInfo* structure, along with a table of name/pointer pairs of the *EXTERNAL* and implicit functions the application offers to the DPCM.

The DPCM shall locate pointers to the application's *EXTERNAL* functions from the passed table. The DPCM fills the *DCMTransmittedInfo* structure with pointers to its main calculation entry points: *modelSearch* (see 10.27.1), *delay* (see 10.26.1), *slew* (see 10.26.2), and *check* (see 10.26.3), followed by a table of pointers to functions, each paired with an *EXPOSE* name from the standard interface.

The application can use *dcmRT_FindFunction* (see 10.25.4.5) or *dcmRT_QuiteFindFunction* (see 10.25.4.7) by naming the corresponding *EXPOSE* function pointer in the *DCMTransmittedInfo* structure. The four implicit functions can be called directly by the application.

10.14 Environment or user variables

The environment variables (UNIX) or user variables (Windows NT) *DCMRULEPATH* and *DCMTABLEPATH* may be referenced when subrules or dynamic tables are loaded. These variables shall contain a colon-delimited sequence of file system directory paths. The subrule- and table-loading subsystems search each directory in the path sequence, in left-to-right order, for a matching filename.

10.15 Procedural interface (PI) functions summary

This subclause summarizes the PI functions defined by this standard. The PI functions are grouped into three categories: *EXPOSE* (which start with the prefix *dpcm*), *EXTERNAL* (which start with the prefix *app*), and run time (which start with the prefix *dcm*).

10.15.1 Expose functions

Table 107 shows the explicit *EXPOSE* functions defined in the DPCM and called by the application. Those *EXPOSE* functions for which *METHODS* may be used explicitly call out *pathData* and *cellData* as fields in the Standard Structure. This subclause defines the *EXPOSE* functions used in this standard.

Table 107—Expose functions

EXPOSE function subclause	Description
pcmAddWireLoadModel 10.18.3.9	Add a custom wire load model (write into) DPCM.
pcmAETGetSettlingTime 10.18.9.5	Returns settling time for a list of pins when using the AET power calculation method.

EXPOSE function subclause	Description
pcmAETGetSimultaneousSwitchTime 10.18.9.6	Returns simultaneous switch times for a list of pins when using the AET power calculation method.
pcmBuildInterconnectModels 10.18.9.2	Returns the interconnect models and the library's updated net models for the interconnect network.
dpcmBuildLoadModels 10.18.9.1	Returns the load models and the library's updated net models for the interconnect network.
dpcmBuildNoiseInterconnectModels 10.22.4.1.3	Returns pointers to the noise interconnect models for a sink pin on an interconnect network, along with updated versions of the library's intermediate noise models for that network.
dpcmBuildNoiseLoadModels 10.22.4.1.4	Returns pointers to the noise load models for a driver of an interconnect network, along with updated versions of the library's intermediate noise models for that network.
dpcmCalcCeff 10.21.12.2	Returns the effective capacitances seen by the passed driver pin (<i>toPoint</i>).
dpcmCalcCouplingCapacitance 10.21.13.1.1	Calculates the capacitance between the aggressor and victim conductors.
dpcmCalcInputNoise 10.22.5.1.1	Calculates a set of composite noise pulses at the input of a cell, given the activity on the direct current (dc)-connected drivers and the aggressor drivers on coupled nets. The function can also be called to calculate noise propagated to a bidirectional pin that is acting as an input.
dpcmCalcOutputNoise 10.22.6.1.1	Calculates a set of composite noise pulses at the output of a cell, given the activity on the related pins. The function can also be called to calculate noise propagated to a bidirectional pin that is acting as an output.
dpcmCalcOutputResistances 10.21.15.6	Returns output resistances for the passed pin corresponding to the early and late slews and transition waveform (XWF) structures provided.
dpcmCalcPartialSwingEnergy 10.18.9.9	Returns the energy of a partial logic swing for AET or group power calculation methods.
dpcmCalcSegmentResistance 10.21.13.1.3	Calculates the resistance.
dpcmCalcSteadyStateResistanceRange 10.21.12.3	Returns the maximum and minimum driving resistances possible for a cell that is not transitioning, regardless of the states where those resistances can occur.
dpcmCalcSubstrateCapacitance 10.21.13.1.2	Computes the plate and fringe capacitances between a conductor on any given layer and the substrate.
dpcmCalcTristateResistanceRange 10.21.12.4	Returns the maximum and minimum driving resistances possible for a cell that is in the high impedance state.
dpcmCalcXWF 10.23.8.3.3	Returns pointers to XWF structures for the passed pin and edge direction corresponding to the early and late slews.
dpcmCopyNWFarray 10.22.3.3	Allocates space for a new array of noise waveforms (NWFs) and copies the contents of the source NWF array passed to it into this new array.
dpcmCopyPWFarray 10.22.3.4	Allocates space for a new pulse waveform (PWF) array and then copies the contents of the source PWF array passed to it into this new array.
dpcmCreatePWF 10.22.3.2	Creates a PWF with points in which the waveform type, the offsets, and the time and voltage values for each point are all uninitialized.
dpcmCreatePWFdriverModel 10.22.3.5	Creates a <i>PWFdriverModel</i> that incorporates the PWF passed to it and includes a resistance array whose size is the same as the number of points in the PWF.

EXPOSE function subclause	Description
dpcmCreateSubnetStructure 10.21.6.12	Allocates a new subnet structure and initializes its link pointers.
dpcmDebug 10.18.8.16	Allows the application to control the debug level setting if the DPCM is compiled with debugging enabled.
dpcmFillPinCache 10.18.8.16	Supplies the load and slew of the specified pin.
dpcmFreePinCache 10.18.8.17	Frees the load and slew cache.
dpcmGetAETCellPowerWithSensitivity 10.18.9.3	Returns static power per rail, dynamic energy per rail, total energy, and total static power.
dpcmGetAllResources 10.23.18.1.2	Queries the library for all resources.
dpcmGetBaseCellRailVoltageArray 10.23.9.2	Returns the default values for the voltage rails of the cell identified by the <i>cellData</i> field in the Standard Structure.
dpcmGetBaseCellTemperature 10.23.9.3	Returns the default temperature value for the cell identified by the <i>cellData</i> field in the Standard Structure.
dpcmGetBaseFunctionalMode 10.18.5.2	Returns the index number of the default functional mode for the cell specified in the Standard Structure.
dpcmGetBaseOpPoint 10.23.10.1.2	Obtains the index for the default operating point.
dpcmGetBasePinFrequency 10.23.13.8	Returns the default pin frequency value.
dpcmGetBaseProcessPoint 10.23.11.1.2	Obtains a base process point from the library.
dpcmGetBaseOpRange 10.18.5.16	Returns the name of the base operating range modeled in the DPCM.
dpcmGetBaseRailVoltage 10.18.5.10	Returns a default voltage value for the specified rail.
dpcmGetBaseTemperature 10.18.5.15	Returns the base operating temperature for the library.
dpcmGetBaseWireLoadModel 10.18.5.13	Returns the index number of the default wire load model for the library.
dpcmGetCapacitanceLimit 10.18.4.1	Returns the capacitance limits for the specified pin.
dpcmGetCellList 10.18.6.1	Return a list of cells that exist in the library.
dpcmGetCellIOlists 10.18.1.14	Return inputs, outputs, and bidirectional pins for the given cell.
dpcmGetCellNonScanCell 10.23.12.4.4	Identifies the nonscan flipflop used to construct the scan flipflop.
dpcmGetCellPowerInfo 10.18.9.1	Returns the power calculation methods supported by the DPCM.
dpcmGetCellPowerWithState 10.18.9.2	Returns power and energy for the group power calculation methodology.
dpcmGetCellRailVoltageRangeArray 10.23.11.3.2	Returns the extrema defining the range of legal values for each voltage rail of the cell identified by the <i>cellData</i> field in the Standard Structure.
dpcmGetCellRestrictClass 10.23.12.4.1	Returns the set of restrict class values that apply to a cell.
dpcmGetCellRestrictClassArray 10.23.12.4	Returns an array of all possible values for the restrict class within a technology.
dpcmGetCellScanType 10.23.12.4.3	Returns the set of the scan type property indices for a cell.

EXPOSE function subclause	Description
dpcmGetCellScanTypeArray 10.23.12.4.2	Returns an array of all possible scan type values used within a technology.
dpcmGetCellSwapClass 10.23.12.3.4	Returns the set of swap class values that apply to a cell.
dpcmGetCellSwapClassArray 10.23.12.3.3	Returns all possible <i>SwapClass</i> names.
dpcmGetCellTemperatureRange 10.23.11.4.2	Returns the extrema defining a range of legal values for the temperature of the cell identified by the <i>cellData</i> field in the Standard Structure.
dpcmGetCellTestProcedure 10.23.24	Obtains the test procedure defined for a cell and returns an array of path data blocks, the functionality of which, as specified by the associated function graphs and when executed in the returned order, performs the desired test sequence.
dpcmGetCellType 10.23.12.3.2	Returns an array of the cell type indices for a cell.
dpcmGetCellTypeArray 10.23.12.3.1	Returns all possible cell types used within a technology.
dpcmGetCellVectorPower 10.23.13.15	Returns the power associated with a cell for a given state vector.
dpcmGetControlExistence 10.18.5.4	Returns information that controls the existence of the segment identified by <i>pathData</i> .
dpcmGetDefaultInterconnectTechnology 10.21.6.14	Returns a technology the application can use when it cannot otherwise determine which technology owns a particular interconnect subnet back to the application.
dpcmGetDefCellSize 10.18.1.7	Returns cell's size metric for interconnect load estimation.
dpcmGetDefPinSlews 10.23.17.1	Returns the default pin slews for the passed <i>pinName</i> of the cell in the Standard Structure.
dpcmGetDifferentialPairPin 10.23.12.1.15	Returns the name of the pin that shall be paired with the known pin.
dpcmGetDefPortCapacitance 10.18.2.6	Returns default capacitance for a chip primary output.
dpcmGetDefPortSlew 10.18.2.5	Returns default slew for a chip primary input.
dpcmGetDelayGradient 10.18.2.2	Returns the rate of change of the delay at the specified point.
dpcmGetEstLoadCapacitance 10.18.1.10	Returns an estimated load capacitance for the specified pin.
dpcmGetEstimateRC 10.18.2.4	Returns an RC value for a specified pin pair. Called when the application does not know the RC value for the current pin pair.
dpcmGetEstWireCapacitance 10.18.1.11	Returns an estimated interconnect capacitance for the interconnect that the specified pin drives.
dpcmGetEstWireResistance 10.18.1.12	Returns the estimated wire resistance to which the specified pin is connected.
dpcmGetExistenceGraph 10.23.15.1	Returns a function graph of the existence condition for the segment identified by the <i>pathData</i> field in the Standard Structure.
dpcmGetExposePurityAndConsistency 10.18.5.8	Allows the application to determine which <i>EXPOSE</i> functions provide data the application can cache.
dpcmGetFrequencyLimit 10.23.13.6	Returns the minimum and maximum frequency limit for a given path or pin.
dpcmGetFunctionalModeArray 10.18.5.1	Returns the functional mode names for the cell specified in the Standard Structure.

EXPOSE function subclause	Description
dpcmGetInductanceLimit 10.23.13.10	Returns the minimum and maximum series inductance permitted on the pin.
dpcmGetLayerArray 10.21.13.2.1	Returns an ordered array of layer names and layer types.
dpcmGetLevelShifter 10.23.12.4.6	Instantiates level shifters between the different chip voltage islands.
dpcmGetLibraryAccuracyLevelArrays 10.18.5.6	Returns the accuracy levels.
dpcmGetLibraryConnectClassArray 10.23.14.2	Returns the list of all possible <i>ConnectClass</i> names contained within a library representing a technology.
dpcmGetLibraryConnectivityRules 10.23.14.3	Returns a matrix of connectivity rules between all connect classes in the library.
dpcmGetLibraryNoiseTypesArray 10.22.1.1.1	Returns an ordered array of strings identifying the custom noise types defined by the library.
dpcmGetLogicalBISTMap 10.23.23.2	Returns the corresponding logical address and bit index where the data word maps, as well as indicating whether the logical data are inverted with respect to the physical data value.
dpcmGroupGetSettlingTime 10.18.9.7	Returns settling time for a list of pins when using the group power calculation method.
dpcmGroupGetSimultaneousSwitchTime 10.18.9.6	Returns simultaneous switch time for a list of pins when using the group power calculation method.
dpcmGetNetEnergy 10.18.9.13	Returns net energy.
dpcmGetNodeSensitivity 10.23.4.3	Returns a list of strand states that shall be monitored.
dpcmGetNoiseViolationDetails 10.22.8.1.2	Returns detailed information about the noise of various types contributed by each related pin to a violation reported via <i>appSetNoiseViolation</i> .
dpcmGetOpPointArray 10.23.10.1.1	Obtains the names of the operating points within a library.
dpcmGetOpRangeArray 10.18.5.17	Returns the operating range names that are modeled in the DPCM.
dpcmGetOutputSourceResistances 10.23.13.11	Returns two output source resistance values, an early value and a late value, for a modeled arc.
dpcmGetPathLabel 10.23.12.2.1	Returns the SDF label for a vector expression that represents at least one state for a path.
dpcmGetPhysicalBISTMap 10.23.23.1	Returns the corresponding physical row, column, and bank where the data word maps, as well as indicating whether the physical data are inverted with respect to the logical data value.
dpcmGetPinAction 10.23.12.1.7	Returns all possible values for the <i>Action</i> property.
dpcmGetPinActionArray 10.23.12.1.6	Returns an array of indices, which comprise the synchronous/asynchronous properties of a signal at the pin of interest.
dpcmGetPinCapacitance 10.18.1.13	Returns the cell's pin capacitance for the specified pin.
dpcmGetPinCellConnectivityArrays 10.23.14.1	Returns arrays of pins, which have defined cell-level connectivity rules relative to the specified pin.
dpcmGetPinConnectClass 10.23.12.1.11	Returns an index into the array of library connect classes.
dpcmGetPinDriveStrength 10.23.13.14	Returns the relative pin drive strength.

EXPOSE function subclause	Description
dpcmGetPinEnablePin 10.23.12.1.10	Returns the list of pins that enable the pin identified by the pin pointer argument.
dpcmGetPinIndexArrays 10.23.22.1	Returns the indices of the input, output, and bidirectional pins of the cell specified in the Standard Structure.
dpcmGetPinJitter 10.23.13.9	Returns the jitter for a given pin.
dpcmGetPinPinType 10.23.12.1.3	Returns the <i>PinType</i> property for a pin.
dpcmGetPinPinTypeArray 10.23.12.1.2	Returns all possible values for the <i>PinType</i> property that can be used by a technology.
dpcmGetPinPolarity 10.23.12.1.9	Returns the <i>Polarity</i> property for a pin.
dpcmGetPinPolarityArray 10.23.12.1.8	Returns a string array of pin polarities that can be used in the technology.
dpcmGetPinPower 10.18.9.4	Returns static power per rail, dynamic energy per rail, total energy, and total static power for a specific pin state change.
dpcmGetPinReadPolarity 10.23.13.2	Returns the <i>ReadPolarity</i> property for a pin.
dpcmGetPinScanPosition 10.23.12.1.12	Returns the position of a pin in the scan for a cell.
dpcmGetPinSignalType 10.23.12.1.5	Returns an array of indices into the signal type array (which constitutes the complete signal type for the pin of interest).
dpcmGetPinSignalTypeArray 10.23.12.1.4	Returns all possible values for the <i>SignalType</i> property used by the technology.
dpcmGetPinStuck 10.23.12.1.14	Returns a list of the stuck failure types for a pin.
dpcmGetPinStuckArray 10.23.12.1.13	Returns all possible stuck-at-fault values used within the technology.
dpcmGetPinTiePolarity 10.23.13.1	Returns the <i>TiePolarity</i> property for a pin.
dpcmGetPinWritePolarity 10.23.13.3	Returns the <i>WritePolarity</i> property for a pin.
dpcmGetPortNames 10.21.7.3	Returns the list of the port names associated with a pin.
dpcmGetPowerStateLabel 10.23.12.2.2	Returns the label for the particular group and condition.
dpcmGetProcessPointRange 10.23.11.2.1	Returns the extrema defining a range of legal values for a process point other than one represented by the best-case, nominal, or worst-case calculation modes.
dpcmGetPull 10.23.13.13	Determines the pull up/down resistance in a series with rail voltages.
dpcmGetPWFArrary 10.22.3.1	Converts an array of library-proprietary NWFs into an array of PWFs, which are suitable for interpretation by an application or by a different technology than that which created the NWF.
dpcmGetPWFdriverModelArray 10.22.3.6	Creates an array of PWF driver models using an array of library-proprietary NWFs and output resistances for the driver pin passed to it.
dpcmGetRailVoltageArray 10.18.5.9	Returns the voltage rail names that are modeled in the DPCM.
dpcmGetRuleUnitToFarads 10.18.8.6	Returns the basic units of capacitance assumed by the technology library.
dpcmGetRuleUnitToHenries 10.18.8.7	Returns the basic units of inductance assumed by the technology library.

EXPOSE function subclause	Description
dpcmGetRuleUnitToJoules 10.18.8.9	Returns the basic units of energy assumed by the technology library.
dpcmGetRuleUnitToOhms 10.18.8.5	Returns the basic units of resistance assumed by the technology library.
dpcmGetRuleUnitToSeconds 10.18.8.4	Returns the basic units of time assumed by the technology library.
dpcmGetRuleUnitToWatts 10.18.8.8	Returns the basic units of power assumed by the technology library.
dpcmGetRuleUnitsToMeters 10.21.13.2.2	Returns the basic units of distance assumed by the technology.
dpcmGetParasiticCoordinateTypes 10.18.8.11	Returns the type of coordinate measure used for parasitic elements by the technology.
dpcmGetRailVoltageRangeArray 10.23.11.3.1	Returns the extrema defining the range of legal values for each rail voltage in a library.
dpcmGetRuleUnitToAmps 10.21.13.2.3	Returns the basic current units the library assumes, expressed as an integer power of 10.
dpcmGetSimultaneousSwitchTimes 10.23.13.4	Returns an array of skew limits associated with a vector expression graph.
dpcmGetSinkPinNoiseParasitics 10.22.4.1.1	Returns two pointers to parasitic subnets for a sink pin, which contain element values for minimum and maximum on-chip process variation.
dpcmGetSinkPinParasitics 10.21.7.1	Returns the parasitic subnets for a sink pin.
dpcmGetSlewGradient 10.18.2.3	Returns the rate of change of the slew at the specified point.
dpcmGetSlewLimit 10.18.4.2	Returns the slew limit for the specified pin.
dpcmGetSourcePinNoiseParasitics 10.22.4.1.2	Returns two pointers to parasitic subnets for a source pin, which contain element values for minimum and maximum on-chip process variation.
dpcmGetSourcePinParasitics 10.21.7.2	Returns the parasitic subnets for the source pin.
dpcmGetSupplyPins 10.23.22.2	Returns the power and ground pins for a cell.
dpcmGetTemperatureRange 10.23.11.4.1	Returns the extrema defining a range of legal values for temperature.
dpcmGetThresholds 10.18.7.1	Returns voltage and transition and delay points.
dpcmGetTimeResolution 10.18.8.10	Returns the coarsest resolution for time values to be used by the application to help ensure accurate interaction with the specified technology.
dpcmGetTimingStateGraphs 10.23.15.2	Returns function graphs of the state condition expressions for the segment identified by the <i>pathData</i> field in the <i>Standard Structure</i> .
dpcmGetTimingStateStrings 10.23.15.3	Returns the condition expression and the labels for the timing state identified by the <i>pathData</i> field in the <i>Standard Structure</i> .
dpcmGetVectorEdgeNumbers 10.23.15.4	Returns numbers for the from-pin and to-pin edges in a timing vector for the segment identified by the <i>pathData</i> field in the <i>Standard Structure</i> .
dpcmGetVersionInfo 10.18.8.14	Returns the version identification for the technology library and the version of IEEE Std 1481 with which the library is compliant.
dpcmGetWireLoadModel 10.18.3.10	Returns wire load model to application.

EXPOSE function subclause	Description
dpcmGetWireLoadModelArray 10.18.5.13	Returns the wire load model names modeled in the DPCM.
dpcmGetWireLoadModelForBlockSize 10.18.3.11	Returns the index number of the appropriate wire load model given a specified area.
dpcmFreeStateCache 10.18.9.11	Frees the state cache whose handle is passed in.
dpcmGetXovers 10.18.4.3	Returns the capacitance limits for cell drive strengths of the cell to which the specified pin is connected.
dpcmHoldControl 10.18.8.15	Returns a signal from the DPCM that allows hold control to be performed.
dpcmIdentifyInternalNode 10.21.8.2	Returns the <i>nodeMap</i> index of the node within the appropriate subnet that corresponds to the internal node identified by the <i>pathData</i> pointer in the Standard Structure.
dpcmIsSlewTime 10.18.8.12	Returns the units for calculated slews as absolute time or rate of change.
dpcmModelMoreFunctionDetail 10.23.5.1.1	Expands the hierarchical node.
dpcmPassivateInterconnectModels 10.21.11.1.2	Converts a pair of library interconnect models to a contiguous collection of bytes that can be persistently stored.
dpcmPassivateLoadModels 10.21.11.1.1	Converts a pair of library load models to a contiguous collection of bytes that can be persistently stored.
dpcmPerformPrimitive 10.23.4.1	Returns a single dimensional array of integers, each representing a bit of the primitive's value.
dpcmRestoreInterconnectModels 10.21.11.1.4	Converts a pair of library passivated interconnect models to the <i>DCM_STRUCT</i> format used by the library during calculations made using the models.
dpcmRestoreLoadModels 10.21.11.1.3	Converts a pair of library passivated load models to the <i>DCM_STRUCT</i> format used by the library during calculations made using the models.
dpcmScaleParasitics 10.21.6.15	Scales the parasitics within a subnet to compensate for changes in the operating point, process point, voltages, and temperature.
dpcmSetCurrentOpPoint 10.23.10.1.3	Sets an operating point.
dpcmSetCurrentProcessPoint 10.23.11.1.1	Sets the current process point.
dpcmSetInitialState 10.18.9.10	Set the initial state for a cell.
dpcmSetLevel 10.18.5.5	Instructs the DPCM to perform calculations for performance (execution speed) or accuracy, and for the scope for derating supported by the application.
dpcmSetLibraryAccuracyLevel 10.18.5.7	Sets the accuracy levels.
dpcmSetNoiseLimit 10.22.7.1	Specifies an application budget for the specified noise type.
dpcmSetParallelRelatedNoise 10.22.6.2.2	Called once for each driver pin on the net to which initiatingPin is connected that is parallel to initiatingPin. For each call, minLoadModel, maxNoiseLoadModel, and noises shall be passed to enable the library to propagate noise simultaneously across all of the parallel drivers.
dpcmSetResource 10.23.18.1.1	Sets the value of a resource.
dpcmGetTimingStateArray 10.18.5.20	Returns an array of strings that represent the valid states for the given segment.

10.15.2 External functions

These functions are defined in the application and declared by the DPCM via the *EXTERNAL* function, as shown in Table 108.

Table 108—External functions

EXTERNAL function subclause	Description
appForEachNoiseParallelDriver 10.22.6.2.1	When called, the application shall respond by calling dpcmSetParallelRelatedNoise once for each of the other parallel driver pins on the net to which the passed pin is connected.
appForEachParallelDriverByName 10.18.3.4	Returns the additional number of logically redundant parallel drivers to the specified driver and a DPCM computed value for driver cells connected in parallel on the specified interconnect (by pin name).
appForEachParallelDriverByPin 10.18.3.3	Returns the additional number of logically redundant parallel drivers to the specified driver and a DPCM computed value for driver cells connected in parallel on the specified interconnect (by pin pointer).
appGetAggressorOverlapWindows 10.21.15.1	Obtains arrays of arrival-window information about the aggressor drivers coupled to a net for which propagation-arc calculation (delay, slew, etc.) is being done.
appGetArcStructure 10.23.4.2	Returns the actual bit pattern present at the time of the call, as well as the data type and any strand information.
appGetArrivalOffsetArraysByName 10.23.2.1	Returns arrays of arrival offsets and slews for the signal associated with the edge and pin name passed as the arguments back to the library.
appGetArrivalOffsetsByName 10.23.2	Returns the arrival offsets and slews for the signal associated with the edge and pin name passed as the arguments.
appGetCeFF 10.21.12.1	Returns the effective capacitances seen by the passed driver pin (toPoint).
appGetCellName 10.18.6.2	Returns the cell name to which the specified pin is connected.
appGetCellCoordinates 10.18.1.8	Returns the x and y coordinates of the cell.
appGetCellOrientation 10.18.1.9	Returns how the cell is orientated on the grid.
appGetCurrentFunctionalMode 10.18.5.3	Returns the current functional mode of the cell instance identified in the Standard Structure.
appGetCurrentOpRange 10.18.5.19	Returns the current operating range.
appGetCurrentRailVoltage 10.18.5.11	Returns a voltage value for the specified rail index number.
appGetCurrentTemperature 10.18.5.18	Returns the desired temperature value to be used for calculations.
appGetCurrentTimingState 10.18.5.21	Returns the current state or condition.
appGetCurrentWireLoadModel 10.18.5.13	Returns the current wire load model.
appGetDriverThresholds 10.21.14.1	Finds the threshold of the driver cells on the net. If there is more than one driver, returns all the driver thresholds.
appGetExternalDelayByPin 10.23.21.1	Returns the total delay and path status, from an output pin to an input pin.

EXTERNAL function subclause	Description
appGetExternalDelayByName 10.23.21.2	Returns the total delay and path status, from an output pin to an input pin.
appGetExternalStatus 10.18.8.1	Returns whether and to what degree the application implemented an EXTERNAL.
appGetHierBlockName 10.18.6.4	Returns the hierarchical instance name for the instance to which the specified pin is connected.
appGetHierNetName 10.18.6.5	Returns the hierarchical interconnect name for the interconnect to which the specified pin is connected.
appGetHierPinName 10.18.6.3	Returns the hierarchical pin name for the specified pin.
appGetInstanceCount 10.18.3.12	Returns the cell instance count for the interconnect region containing the driving the specified pin.
appGetInterconnectModels 10.21.9.3	Returns the interconnect models created by the library for the path between the pins specified in the Standard Structure.
appGetInterfaceVersion 10.23.17.2	Returns an array of strings, identifying the versions of the IEEE 1481 interface that the application supports.
appGetLoadModels 10.21.9.4	Returns the load models created by the library for the pin specified in the toPoint field of the Standard Structure.
appGetLogicLevelByName 10.23.21.3	Returns the logic level on the pin specified by the pin name.
appGetLogicLevelByPin 10.23.21.4	Returns the logic level on the pin.
appGetOverlapNWFs 10.21.15.4	Called by the library to obtain arrays of noise waveforms for dc-connected drivers and aggressor drivers coupled to a net for which propagation-arc calculation (delay, slew, etc.) is being done.
appGetParasiticNetworksByName 10.21.10.2	Returns the parasitic networks for the net connected to the named pin on the block supplied in the Standard Structure.
appGetParasiticNetworksByPin 10.21.10.1	Returns the parasitic elements structures that represent the non-reduced parasitic networks for that net.
appGetNumDriversByName 10.18.3.2	Returns the number of source (driving) pins (including bidirectional pins) on the interconnect to which the named pin is connected.
appGetNumDriversByPin 10.18.3.1	Returns the number of source (driving) pins (including bidirectional pins) on the interconnect to which the specified pin is connected.
appGetNumPinsByName 10.18.3.6	Returns the total number of pins on the interconnect to which the named pin is connected.
appGetNumPinsByPin 10.18.3.5	Returns the total number of pins on the interconnect to which the specified pin is connected.
appGetNumSinksByName 10.18.3.8	Returns the number of sinks (including bidirectional pins) on the interconnect to which the named pin is connected.
appGetNumSinksByPin 10.18.3.7	Returns the number of sinks (including bidirectional pins) on the interconnect to which the specific pin is connected.
appGetPinFrequency 10.23.13.7	Returns the current frequency on a particular pin.
appGetRC 10.18.2.1	Returns the RC value for the specified pin pair.
appGetResource 10.18.8.3	Returns a user-supplied value for an arbitrary keyword passed from the DPCM.
appGetSwitchingBits 10.23.13.5	Returns the number of bits switching on a bus during processing of a dpcmGetCellPowerWithState or a dpcmGetCellVectorPower, respectively.
appGetThresholds 10.18.7.2	Returns voltage and transition and delay points.

EXTERNAL function subclause	Description
appGetStateCache 10.18.9.12	Retrieves the cache handle associated with a cell instance and created with the call dpcmSetInitialState
appGetSourcePinCapacitanceByPin 10.18.1.5	Returns the capacitance of the identified pin. The identified pin shall be a pin that drives an interconnect network.
appGetSourcePinCapacitanceByName 10.18.1.6	Returns the capacitance of the named pin. The named pin shall be a pin that drives an interconnect network.
appGetTotalLoadCapacitanceByName 10.18.1.2	Returns the total capacitance (sum of capacitance on all pins plus wire capacitance) for the interconnect to which the named pin is connected.
appGetTotalLoadCapacitanceByPin 10.18.1.1	Returns the total capacitance (sum of capacitance on all pins plus wire capacitance) for the interconnect to which the specified pin is connected.
appGetTotalPinCapacitanceByName 10.18.1.4	Returns the total pin capacitance (sum of capacitance on all pins on the interconnect) for the interconnect to which the named pin is connected.
appGetTotalPinCapacitanceByPin 10.18.1.3	Returns the total pin capacitance (sum of capacitance on all pins on the interconnect) for the interconnect to which the specified pin is connected.
appGetVersionInfo 10.18.8.2	Returns the version of IEEE Std 1481 with which the application is compliant.
appGetXWF 10.23.8.3.2	Returns the XWF data structures obtained from a previous call to appSetXWF (which were applied to the pin at the beginning of this arc).
appNewNoiseCone 10.22.2.1	Describes the set of pins that are related to (that can have noise affecting) an output pin on a cell.
appRegisterCellInfo 10.18.8.18	This call allows the application to supply load and slew information.
appSetAggressorInteractWindows 10.21.15.2	Stores arrays of interaction windows, one window in each array for each aggressor driver, in the application.
appSetCeff 10.21.12.5	Passes the effective load capacitances seen by and output resistances of the driver pin and for the edge identified by the toPoint and sinkEdge fields, respectively, in the Standard Structure
appSetDriverInteractWindows 10.21.15.5	Used to store arrays of interaction windows, one window in each array for each dc-connected driver, in the application
appSetNoiseViolation 10.22.8.1.1	Describes violations detected by the library during noise calculation.
appSetParallelOutputNoise 10.22.6.2.3	Called by the library when noise waveforms are computed for the output of a parallel driver as a side effect of the calculations done by dpcmCalcOutputNoise for an initiating driver.
appSetPull 10.23.13.12	Passes the enumeration DCM_PullType, which indicates the pull-up, pull-down, or both arrangement
appSetSignalDivision 10.23.16.1	Sends data describing input signal sampling or output signal rate division characteristics for the segment last defined during timing model elaboration or for which calculation is currently being performed.
appSetSignalGeneration 10.23.16.3	Sends data describing input signal generation characteristics for the segment last defined during timing model elaboration or for which calculation is currently being performed.
appSetSignalMultiplication 10.23.16.2	Sends data describing output signal rate multiplication characteristics for the segment last defined during timing model elaboration or for which calculation is currently being performed.
appSetVectorOperations 10.23.12.4.5	Requests the application store an array of vector operations, which are associated with the vector specified by the segment last defined during vector timing model elaboration.

EXTERNAL function subclause	Description
appSetXWF 10.23.8.3.1	Sets computed XWF data structures for the pin, edge, and timing arc as identified by the toPoint, sinkEdge, and pathData fields in the Standard Structure.

10.15.3 Deprecated functions

Several functions were defined in the previous version of this standard that are considered to be deprecated. They are presented here with the understanding that in a future release, these functions may no longer be supported. They are preserved in this standard for the purpose of providing existing applications and libraries a period of time in which to switch over to using the new versions. The deprecated functions are defined in Table 109.

Table 109—Deprecated functions

Function subclause	Description
appGetPiModel 10.28.1.1	Returns the pi model capacitance and resistance values for the interconnect to which the specified pin is connected.
appGetPolesAndResidues 10.28.1.2	Returns poles and residues for the specified load.
appGetCeffective 10.28.1.3	Returns the value of C-effective of the load seen by the specified driver.
appGetRLCnetworkByName 10.28.1.5	Returns the RLC elements representing the interconnect to which the named pin is connected.
appGetRLCnetworkByPin 10.28.1.4	Returns RLC elements representing the interconnect to which the specified pin is connected.
dpcmCalcPiModel 10.28.1.6	Calculates a pi model for the interconnect driven by the specified pin drives.
dpcmCalcPolesAndResidues 10.28.1.7	Calculates poles and residues for the specified load.
dpcmCalcCeffective 10.28.1.8	Returns the effective capacitance for the load seen by the specified driving pin.
dpcmSetRLCmember 10.28.1.9	A function that is called to pass R, L, C, and M elements within the specified interconnect.
dpcmAppendPinAdmittance 10.28.1.10	Adds the admittance of a receiver pin to the RLC tree for the interconnect.
dpcmDeleteRLCnetwork 10.28.1.11	Deletes previously created RLC network.
dcmBindRule 10.28.4.6	Loads and links the specified DCL subrule and returns the initialization entry point.
dcmSetMessageIntercept 10.28.4.14	Allows the application to supply a message interceptor, which controls the printing of DPCM messages.
dcmSetNewStorageManager 10.28.4.2	Allows the application to assert its storage management system on the DPCM.
dcmAddRule 10.28.4.7	Adds additional DCL subrules to the DPCM after dcmBindRule has been called.
dcmCellList 10.28.4.1	Returns the list of cell names in the current library.

Function subclause	Description
dcmFindAppFunction 10.28.4.10	Determines whether the application defined the indicated EXTERNAL function.
dcmFindFunction 10.28.4.9	Returns a pointer to the function matching the specified name. Issues an error message if the specified function name cannot be found.
dcmFree 10.28.4.4	Instructs the DPCM to free memory using the storage management function currently in effect.
dcmIssueMessage 10.28.4.15	Instructs the DPCM to print a message using the dcmSetMessageIntercept currently in effect.
dcmMakeRC 10.28.4.12	Returns an error code constructed from the message number and severity arguments, which shall not conflict with internal DCL reserved codes (such as those returned from dcmHardErrorRC).
dcmMalloc 10.28.4.3	Instructs the DPCM to allocate memory using the storage management function currently in effect.
dcmQuietFindFunction 10.28.4.11	Returns a pointer to the function matching the specified name. No error message is issued if the specified function name cannot be found.
dcmRealloc 10.28.4.5	Instructs the DPCM to reallocate memory using the storage management function currently in effect.
dcmUnbindRule 10.28.4.8	Unloads the DPCM from memory and releases any memory it may have used.
dcm_freeAllTechs 10.28.4.22	Frees the technology list created by dcm_getAllTechs.
dcm_getAllTechs 10.28.4.21	Returns a list of all technologies loaded as part of the current DPCM.
dcm_getTechnology 10.28.4.20	Returns the technology name where the Standard Structure is mapped.
dcm_isGeneric 10.28.4.23	Indicates whether the current Standard Structure is pointing to a generic technology.
dcm_mapNugget 10.28.4.24	Returns the current technology mapping structure.
dcm_registerUserObject 10.28.4.26	Registers a pointer to a user-defined object with the Standard Structure so that it may be deleted by dcm_DeleteRegisteredUserObjects or dcm_DeleteOneUserObject.
dcm_rule_init 10.28.4.16	Entry called to initialize the DPCM previously loaded.
dcm_setTechnology 10.28.4.19	Set the technology mapping in the Standard Structure to the specified technology name.
dcm_takeMappingOfNugget 10.28.4.25	Sets the Standard Structure to point to the technology contained in the nugget.
dcm_DeleteRegisteredUserObjects 10.28.4.27	Deletes the user objects associated with the specified Standard Structure.
dcm_DeleteOneUserObject 10.28.4.28	Deletes a single user object that was registered to the specified Standard Structure.
dcmHardErrorRC 10.28.4.13	Returns the constructed return code “hard error rc.”
DCM_new_DCM_STD_STRUCT 10.28.4.17	Allocates and initializes a new Standard Structure.

Function subclause	Description
DCM_delete_DCM_STD_STRUCT 10.28.4.18	Deletes a previously allocated Standard Structure.
dcm_copy_DCM_ARRAY 10.28.2.1	Copies the contents from one DCM_ARRAY to another DCM_ARRAY.
dcm_new_DCM_ARRAY 10.28.2.2	Allocates a new DCM_ARRAY.
dcm_sizeof_DCM_ARRAY 10.28.2.3	Calculates the size of a DCM_ARRAY.
dcm_lock_DCM_ARRAY 10.28.2.4	Locks a DCM_ARRAY.
dcm_unlock_DCM_ARRAY 10.28.2.5	Decrements the claim count of a DCM_ARRAY.
dcm_lock_DCM_STRUCT 10.28.3.1	Increments the claim count of a structure.
dcm_unlock_DCM_STRUCT 10.28.3.2	Decrements the claim count of a structure.
dcm_getNumDimensions 10.28.3.3	Returns the number of dimensions of a DCM_ARRAY.
dcm_getNumElementsPer 10.28.3.4	Returns the number of elements in each dimension of a DCM_ARRAY.
dcm_getNumElements 10.28.3.5	Returns the number of elements in a specific dimension of a DCM_ARRAY.
dcm_getElementType 10.28.3.6	Returns the element type contained in a DCM_ARRAY.
dcm_arraycmp 10.28.3.7	Tests to determine if two DCM_ARRAYs have the identical contents.

10.16 Implicit functions

Implicit functions are identified as such because their names do not appear explicitly in the DCL source. They can be categorized as *libdcmlr* functions, initialization (or run-time library) functions, calculation functions (main calculation entry points), and modeling functions. These functions are also available for use by the application.

10.16.1 libdcmlr

Table 110 lists the *libdcmlr* functions; calls to these functions shall precede the use of any other DPCM calls.

Table 110—libdcmlr functions

libdcmlr function subclause	Description
dcmRT_InitRuleSystem 10.25.4.1	Initializes the DCL run-time system. The application shall call this function before calling any other run-time or library calls.
dcmRT_BindRule 10.25.4.2	Loads and links the specified DCL subrule and returns the initialization entry point.
dcmRT_SetMessageIntercept 10.25.4.10	Allows the application to supply a message intercept, which controls the printing of DPCM messages.

10.16.2 Run-time library utility functions

Once a DPCM is loaded, various information query and setup calls may be made. These functions shall be dynamically linkable to the application. Run-time functions shall be separate from the DPCM so multiple subrules can be linked without conflict.

10.16.2.1 Module control functions

Table 111 describes the module control functions.

Table 111—Module control functions

Initialization function subclause	Description
dcmRT_AppendRule 10.25.4.3	Adds additional DCL subrules to the DPCM after dcmRT_BindRule has been called.
dcmRT_FindAppFunction 10.25.4.6	Determines whether the application defined the indicated EXTERNAL function.
dcmRT_FindFunction 10.25.4.5	Returns a pointer to the function matching the specified name and Issues an error message if the specified function name cannot be found.
dcmRT_QuietFindFunction 10.25.4.7	Returns a pointer to the function matching the specified name. No error message is issued if the specified function name cannot be found.
dcmRT_UnbindRule 10.25.4.4	Unloads the DPCM from memory and releases any memory it may have used.
dcmRT_freeAllTechs 10.25.4.17	Frees the technology list created by dcmRT_getAllTechs.
dcmRT_getAllTechs 10.25.4.16	Returns a list of all technologies loaded as part of the current DPCM.
dcmRT_getTechnology 10.25.4.15	Returns the technology name where the Standard Structure is mapped.
dcmRT_isGeneric 10.25.4.18	Indicates whether the current Standard Structure is pointing to a generic technology.
dcmRT_setTechnology 10.25.4.14	Set the technology mapping in the Standard Structure to the specified technology name.
dcmRT_takeMappingOfNugget 10.25.4.19	Sets the Standard Structure to point to the technology contained in the nugget.

10.16.3 Memory control functions

Memory control functions (Table 112) control the allocation, freeing, and serializing access to structures and arrays.

Table 112—Memory control functions

Initialization function subclause	Description
dcmRT_claim_DCM_STRUCT 10.25.3.1	Increments the claim count of a DCM_STRUCT.
dcmRT_disclaim_DCM_STRUCT 10.25.3.2	Decrements the claim count of a DCM_STRUCT. The application shall not call dcmRT_disclaim_DCM_STRUCT more time than it has called dcmRT_claim_DCM_STRUCT on any DCM_STRUCT.
dcmRT_longlock_DCM_STRUCT 10.25.3.3.1	Requests serialized access to a DCM_STRUCT with the SYNC attribute (passed to it by the library).

Initialization function subclause	Description
dcmRT_longunlock_DCM_STRUCT 10.25.3.3.2	Releases the serialized access to a <i>DCM_STRUCT</i> that was granted earlier through a call to dcmRT_longlock_DCM_STRUCT. The structure being unlocked had the SYNC attribute (passed to it by the library).
dcmRT_registerUserObject 10.25.4.20	Registers a pointer to a user-defined object with the Standard Structure so that it may be deleted by dcmRT_DeleteRegisteredUserObjects or by dcmRT_DeleteOneUserObject.
dcmRT_DeleteRegisteredUserObjects 10.25.4.21	Deletes the user objects associated with the specified Standard Structure.
dcmRT_DeleteOneUserObject 10.25.4.22	Deletes a single user object that was registered to the specified Standard Structure.
dcmRT_new_DCM_STD_STRUCT 10.25.4.12	Allocates and initializes a new Standard Structure.
dcmRT_delete_DCM_STD_STRUCT 10.25.4.13	Deletes a previously allocated Standard Structure.
dcmRT_copy_DCM_ARRAY 10.25.1.1	Copies the contents from one DCM_ARRAY to another DCM_ARRAY.
dcmRT_new_DCM_ARRAY 10.25.1.2	Allocates a new DCM_ARRAY.
dcmRT_sizeof_DCM_ARRAY 10.25.1.3	Calculates the size of a DCM_ARRAY.
dcmRT_claim_DCM_ARRAY 10.25.1.4	Increments the claim count of a DCM_ARRAY.
dcmRT_disclaim_DCM_ARRAY 10.25.1.5	Decrements a claim count of a DCM_ARRAY.
dcmRT_getNumDimensions 10.25.3.4	Returns the number of dimensions of a DCM_ARRAY.
dcmRT_getNumElementsPer 10.25.3.5	Returns the number of elements in each dimension of a DCM_ARRAY.
dcmRT_getNumElements 10.25.3.6	Returns the number of elements in a specific dimension of a DCM_ARRAY.
dcmRT_getElementType 10.25.3.7	Returns the element type contained in a DCM_ARRAY.
dcmRT_arraycmp 10.25.3.8	Tests to determine if two DCM_ARRAYs have the identical contents.

10.16.4 Message and error control functions

Message and error control functions issue messages and set error return code values (Table 113).

Table 113—Message and error control functions

Initialization function subclause	Description
dcmRT_IssueMessage 10.25.4.11	Instructs the DPCM to print a message using the dcmSetMessageIntercept currently in effect.
dcmRT_MakeRC 10.25.4.8	Returns an error code constructed from the message number and severity arguments, which shall not conflict with internal DCL reserved codes (such as those returned from dcmHardErrorRC).
dcmRT_HardErrorRC 10.25.4.9	Returns the constructed return code "hard error rc".

10.16.5 Calculation functions

The main calculation entry point functions are called by the application to perform calculation of delay, slew and timing checks, as shown in Table 114.

Table 114—Calculation functions

Calculation function subclause	Description
check 10.26.3	Calculates values to be used for timing checks for the specified timing arc.
elay 10.26.1	Calculates the delay for the specified timing arc.
slew 10.26.2	Calculates the slew for the specified timing arc.

10.16.6 Modeling functions

An application's call to *modelSearch* (see 10.27.1) causes implicit callbacks from the DPCM to the application. *modelSearch* is used to find a model and convey its structure to an application. Pointers to these functions paired with the following names are presented to the DPCM, along with the list of *EXTERNAL* entries in the call to *dcmRT_InitRuleSystem* (see 10.25.4.1). The application shall define all the functions described in Table 115.

Table 115—Modeling functions

Modeling function subclause	Description
modelSearch 10.27.1	Called by the application for each instance of a cell that has to be modeled. Initiates a sequence of calls to the application by the DPCM to convey the model's structure.
newAltTestSegment 10.27.12	Called by the DPCM to create a timing check arc for the specified from-to pin pair.
newDelayMatrixRow 10.27.7	Called by the DPCM to describe the propagation characteristics for arcs created by PATH, BUS, INPUT, OUTPUT, DO: NODE IMPORT, and DO: NODE EXPORT.
newNetSinkPropagateSegments 10.27.8	Called by the DPCM to create delay arcs to a specified pin from all sources.
newNetSourcePropagateSegments 10.27.9	Called by the DPCM to create delay arcs from a specified pin to all loads (sinks).
newPropagateSegment 10.27.10	Called by the DPCM to creates a delay arc for a specified from-to pin pair.
newTestMatrixRow 10.27.11	Called by the DPCM to describe the propagation characteristics for timing arcs created by DCL TEST statements.
newTimingPin 10.27.6	Called by the DPCM to create storage for a timing node internal to a cell.

10.17 PI function table description

The description of each PI functions (see 10.15) includes a two-part table and a function description.

The first part of Table 116 defines the function name, and where applicable, the argument(s), result(s) and Standard Structure field(s) used. The Standard Structure *pathData* pointer can originate from two sources, relative to an arc if created by a *Path*, *Bus*, or *Test* statement, or relative to a pin if created by an *INPUT* or *OUTPUT* statement. In the *Standard Structure* column, when *pathData* is listed as a required field, it indicates whether it originates from a pin (“timing pin specific”), an arc (“timing arc specific”), or either one (“timing”).

The second part of the table illustrates, where applicable, the DCL and/or C syntax of the function call. Table 116 and the subclauses that follow it describe the different characteristics of a PI function.

Table 116—PI function table example

Function name	The interface function name. For application functions, this may be different from the actual name declared in the code (see 10.16).
Arguments	An argument is a value passed by the caller to the function.
Result	A result is a value or values returned through a structure pointer passed in as an argument
Standard Structure fields	The entries listed in this column are those fields within the Standard Structure that may be read by the called function.
DCL syntax	The DCL syntax for the function interface is shown here.
C syntax	The C syntax for the function interface is shown here.

10.17.1 Arguments

When there is duplication of data for arguments of functions, the passed parameters take precedence over data contained in the Standard Structure field.

10.17.1.1 Standard Structure fields

Any names appearing in the Standard Structure fields column of a PI function table are input values to the function being invoked. Values shall be set in the Standard Structure by the caller, although in some cases, not all values have to be reset for every iteration in a loop (see 10.7.1).

Common semantics for values are summarized in Table 117. Refer to the description of a particular function for alternate interpretations of these variables or for descriptions of other variables not included in this table.

Table 117—Standard Structure field semantics

Value	Description
<i>block</i>	The cell instance relevant for the current context. This may be the instance for the pin(s) identified in the Arguments column, or those identified by the <i>FROM_POINT</i> or <i>TO_POINT</i> in other fields of the <i>std_struct</i> .
<i>CellName</i>	<i>CellName</i> represents the following three fields in the Standard Structure: <i>cell</i> , <i>cellQual</i> , and <i>modelDomain</i> . These three fields together uniquely qualify a particular model in the DPCM.
<i>cellData</i> <i>pathData</i>	Pointers to the current <i>PATH</i> and <i>CELL</i> (returned by the DPCM to the application at <i>modelSearch</i> time). If these fields are specified in the PI-specific table, then the application shall supply these values in the Standard Structure, and the DPCM is free to implement the function using METHOD functions. If these fields are not specified, the DPCM shall not use METHODS for that PI call.
<i>calcMode</i>	<i>calcMode</i> defines whether the computation is for the best case, worse case, or nominal. All PI calls that mention <i>calcMode</i> in the list of Standard Structure fields shall return a best case, worse case, or nominal result based on the value of <i>calcMode</i> .

If neither *pathData* nor *cellData* appears in the Standard Structure field list, then that PI function may be called before modeling that cell (using *modelSearch*).

10.17.2 DCL syntax

EXPOSE and *EXTERNAL* functions give only an abbreviated DCL syntax. Depending on the function, there may be zero or more *PASSED* parameters or *RESULT* variables. For a complete syntax, refer to the appropriate description of a function (see Clause 7.).

10.17.3 C syntax

Functions with one or more *Result* value(s) are declared with a pointer to a specially declared *struct*.

10.18 PI function descriptions

This subclause details the DPCS standard PI functions. Each DCL statement can be accessed through a PI function.

10.18.1 Interconnect loading related functions

This subclause describes the interconnect loading related functions.

10.18.1.1 appGetTotalLoadCapacitanceByPin

Table 118 provides information on appGetTotalLoadCapacitanceByPin.

Table 118—appGetTotalLoadCapacitanceByPin

Function name	appGetTotalLoadCapacitanceByPin
Arguments	Pin pointer
Result	Total load capacitance
Standard Structure fields	calcMode
DCL syntax	EXTERNAL (appGetTotalLoadCapacitanceByPin) : passed (pin: outputPin) result (double: loadCapByPin);
C syntax	typedef struct { DCM_DOUBLE loadCap; } T_TotalLoadCapByPin; int appGetTotalLoadCapacitanceByPin (DCM_STD_STRUCT *std_struct, T_TotalLoadCapByPin *rtn, DCM_PIN outputPin);

This returns to the DPCM the total capacitance of the interconnect to which the passed pin is connected. The total capacitance is the sum of capacitance on all pins on the interconnect plus the interconnect's total capacitance.

10.18.1.2 appGetTotalLoadCapacitanceByName

Table 119 provides information on appGetTotalLoadCapacitanceByName.

Table 119—appGetTotalLoadCapacitanceByName

Function name	appGetTotalLoadCapacitanceByName
Arguments	Pin name
Result	Total load capacitance
Standard Structure fields	block, calcMode
DCL syntax	EXTERNAL (appGetTotalLoadCapacitanceByName) : passed(string: outputPin) result(double: loadCapByName);
C syntax	typedef struct { DCM_DOUBLE loadCap; } T_TotalLoadCapByName; int appGetTotalLoadCapacitanceByName (DCM_STD_STRUCT *std_struct, T_TotalLoadCapByName *rtn, STRING outputPin);

This returns to the DPCM the total capacitance of the interconnect to which the passed pin name is connected. The total capacitance is the sum of the capacitance of all pins on the interconnect plus the interconnect's total capacitance.

10.18.1.3 appGetTotalPinCapacitanceByPin

Table 120 provides information on appGetTotalPinCapacitanceByPin.

Table 120—appGetTotalPinCapacitanceByPin

Function name	appGetTotalPinCapacitanceByPin
Arguments	Pin pointer
Result	Total pin capacitance
Standard Structure fields	calcMode
DCL syntax	EXTERNAL (appGetTotalPinCapacitanceByPin) : passed(pin: outputPin) result(double: totalPinCapByPin);
C syntax	typedef struct { DCM_DOUBLE totalPinCap; } T_TotalPinCapByPin; int appGetTotalPinCapacitanceByPin (DCM_STD_STRUCT *std_struct, T_TotalPinCapByPin *rtn, DCM_PIN outputPin);

This returns the total pin capacitance of the interconnect to which the passed pin is connected. The total capacitance is the sum of the capacitances for all pins on the interconnect.

10.18.1.4 appGetTotalPinCapacitanceByName

Table 121 provides information on appGetTotalPinCapacitanceByName.

Table 121—appGetTotalPinCapacitanceByName

Function name	appGetTotalPinCapacitanceByName
Arguments	Pin name
Result	Total pin capacitance
Standard Structure fields	block, calcMode
DCL syntax	EXTERNAL (appGetTotalPinCapacitanceByName) : passed(string: outputPin) result(double: totalPinCapByName);
C syntax	typedef struct { DCM_DOUBLE totalPinCap; } T_TotalPinCapByName; int appGetTotalPinCapacitanceByName (DCM_STD_STRUCT *std_struct, T_TotalPinCapByName *rtn, STRING outputPin);

This returns the total pin capacitance of the interconnect to which the passed pin name is connected. The total capacitance is the sum of the capacitances for all pins on the interconnect.

10.18.1.5 appGetSourcePinCapacitanceByPin

Table 122 provides information on appGetSourcePinCapacitanceByPin.

Table 122—appGetSourcePinCapacitanceByPin

Function name	appGetSourcePinCapacitanceByPin
Arguments	Driver (source) pin pointer
Result	Total sourcePin capacitance
Standard Structure fields	calcMode
DCL syntax	EXTERNAL (appGetSourcePinCapacitanceByPin) : passed(pin: sourcePin) result(double: sourcePinCapByPin);
C syntax	typedef struct { DCM_DOUBLE sourcePinCap; } T_sourcePinCapacitanceByPin; int appGetSourcePinCapacitanceByPin (DCM_STD_STRUCT *std_struct, T_sourcePinCapacitanceByPin *rtn, DCM_PIN sourcePin);

This returns the total capacitance of all driver (source) pins on the interconnect to which the passed driver (source) pin is connected. The total capacitance is the sum of the capacitances for all driver (source) pins on the interconnect (including bidirectional pins).

10.18.1.6 appGetSourcePinCapacitanceByName

Table 123 provides information on appGetSourcePinCapacitanceByName.

Table 123—appGetSourcePinCapacitanceByName

Function name	appGetSourcePinCapacitanceByName
Arguments	Driver (source) pin name
Result	Total driver (source) pin capacitance
Standard Structure fields	block, calcMode
DCL syntax	EXTERNAL (appGetSourcePinCapacitanceByName) : passed(string: sourcePin) result(double: sourcePinCapByName);
C syntax	typedef struct { DCM_DOUBLE sourcePinCap; } T_sourcePinCapacitanceByName; int appGetSourcePinCapacitanceByName (DCM_STD_STRUCT *std_struct, T_sourcePinCapacitanceByName *rtn, STRING sourcePin);

This returns the total capacitance of all driver (source) pins on the interconnect to which the passed driver (source) pin name is connected. The total capacitance is the sum of the capacitances for all driver (source) pins on the interconnect (including bidirectional pins).

10.18.1.7 dpcmGetDefCellSize

Table 124 provides information on dpcmGetDefCellSize.

Table 124—dpcmGetDefCellSize

Function name	dpcmGetDefCellSize
Arguments	None
Result	Cell's size metric for interconnect load estimation
Standard Structure fields	CellName
DCL syntax	EXPOSE (dpcmGetDefCellSize) : result(double: cellSize);
C syntax	typedef struct { DCM_DOUBLE cellSize; } T_defCellSize; int dpcmGetDefCellSize (const DCM_STD_STRUCT *std_struct, T_defCellSize *rtn);

This returns the cell's size metric for interconnect load estimation.

10.18.1.8 appGetCellCoordinates

Table 125 provides information on appGetCellCoordinates.

Table 125—appGetCellCoordinates

Function name	appGetCellCoordinates
Arguments	None
Result	Cell coordinates
Standard Structure fields	CellName, Block

DCL syntax	EXTERNAL(dpcmGetCellCoordinates): result(float: xCoordinate, yCoordinate);
C syntax	typedef struct { FLOAT xCoordinate, yCoordinate; } T_cellCoordinates; int appGetCellCoordinates (DCM_STD_STRUCT *std_struct, T_cellCoordinates *rtn);

This returns the cell's x and y coordinates relative to the lower left corner. The lower left corner shall be defined to be the lower left corner of the smallest possible rectangle that contains the cell.

10.18.1.9 appGetCellOrientation

Table 126 provides information on appGetCellOrientation.

Table 126—appGetCellOrientation

Function name	appGetCellOrientation
Arguments	None
Result	Cell orientation
Standard Structure fields	CellName, Block
DCL syntax	EXTERNAL(appGetCellOrientation): result(int: orientation);
C syntax	typedef struct { INTEGER orientation; } T_cellOrientation; int appGetCellOrientation (DCM_STD_STRUCT *std_struct, T_cellOrientation *rtn);

This returns the cell's orientation. A cell's orientation is relative to the default orientation where the y coordinate is along the left edge of the cell and the x coordinate is along the bottom edge of the cell with the origin at the lower left corner. When permitted by the technology, the cells can exist in up to eight configurations. The configurations consist of rotations and flips. A cell can be rotated up to 270° in 90° increments. A cell can be flipped to its mirror configuration. A cell instantiated in the mirror image is the case where the cell has been flipped in the y axis such that the original x axis is still on the bottom but the original y axis is along the right side of the cell and the original origin is at the lower right corner of the cell. After the operations of flipping and rotating have been completed, the origin is reset to the lower left corner of the new orientation.

The cellOrientation values shall represent the orientation operations performed on the cell. The legal values that may be represented by cellOrientation are the values 0 through 7. All others shall be considered an error. The least significant 2 bits of the cellOrientation (0x3) shall indicate the degrees of rotation in 90° increments. The third least significant bit (0x4) having a value of 1 shall indicate the cell has been flipped.

10.18.1.10 dpcmGetEstLoadCapacitance

Table 127 provides information on dpcmGetEstLoadCapacitance.

Table 127—dpcmGetEstLoadCapacitance

Function name	dpcmGetEstLoadCapacitance
Arguments	Pin pointer
Result	Estimated interconnect load capacitance
Standard Structure fields	CellName, calcMode, pathData (timing-pin-specific), cellData (timing)
DCL syntax	<pre>EXPOSE(dpcmGetEstLoadCapacitance): passed(pin: outputPin) result(double: estLoadCap);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE estLoadCap; } T_estLoadCap; int dpcmGetEstLoadCapacitance (const DCM_STD_STRUCT *std_struct, T_estLoadCap *rtn, DCM_PIN outputPin);</pre>

This returns an estimated loading capacitance taking into account the effects of the interconnects and all pins connected to which the passed pin is connected.

NOTE—This function is used when the loading capacitance value is not otherwise available within the application's model.

10.18.1.11 dpcmGetEstWireCapacitance

Table 128 provides information on dpcmGetEstWireCapacitance.

Table 128—dpcmGetEstWireCapacitance

Function name	dpcmGetEstWireCapacitance
Arguments	Pin pointer
Result	Estimated interconnect wire capacitance
Standard Structure fields	CellName, calcMode, pathData (timing-pin specific), cellData (timing), block
DCL syntax	<pre>EXPOSE(dpcmGetEstWireCapacitance): passed(pin: outputPin) result(double: estWireCap);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE estWireCap; } T_estWireCap; int dpcmGetEstWireCapacitance (const DCM_STD_STRUCT *std_struct, T_estWireCap *rtn, DCM_PIN outputPin);</pre>

This returns the estimated wire capacitance for the interconnect to which the passed pin is connected. The DPCM recognizes an output pin value 0 (zero) as a special indicator to return the technology-wide default.

10.18.1.12 dpcmGetEstWireResistance

Table 129 provides information on dpcmGetEstWireResistance.

Table 129—dpcmGetEstWireResistance

Function name	dpcmGetEstWireResistance
Arguments	Pin pointer
Result	Estimated interconnect wire resistance
Standard Structure fields	CellName, calcMode, pathData (timing-pin-specific), cellData (timing), block
DCL syntax	<pre>EXPOSE(dpcmGetEstWireResistance): passed(pin: outputPin) result(double: estWireResistance);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE estWireResistance; } T_estWireResistance; int dpcmGetEstWireResistance (const DCM_STD_STRUCT *std_struct, T_estWireResistance *rtn, DCM_PIN outputPin);</pre>

This returns the estimated wire resistance for the interconnect to which the passed pin is connected. The DPCM takes the estimated RC time constraint and divides it by the estimated wire capacitance to determine the estimated wire resistance. The DPCM recognizes an output pin value 0 (zero) as a special indicator to return the technology-wide default.

10.18.1.13 dpcmGetPinCapacitance

Table 130 provides information on dpcmGetPinCapacitance.

Table 130—dpcmGetPinCapacitance

Function name	dpcmGetPinCapacitance
Arguments	Pin name
Result	Rise pin capacitance, Fall pin capacitance
Standard Structure fields	CellName, calcMode, pathData, (timing-pin-specific), cellData (timing), block
DCL syntax	<pre>EXPOSE(dpcmGetPinCapacitance): passed(string: pinName) result(double: risePinCap, fallPinCap);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE risePinCap, fallPinCap; } T_riseFallCap; int dpcmGetPinCapacitance (const DCM_STD_STRUCT *std_struct, T_riseFallCap *rtn, STRING pinName);</pre>

This returns the pin capacitance for the passed *pinName* of the *cellName* in the Standard Structure.

10.18.1.14 dpcmGetCellIOlists

Table 131 provides information on dpcmGetCellIOlists.

Table 131—dpcmGetCellIOlists

Function name	dpcmGetCellIOlists
Arguments	
Result	Input pins, Output pins, Bidirectional pins
Standard Structure fields	CellName
DCL syntax	EXPOSE(dpcmGetCellIOlists): result(string[*]: inputPins, outputPins, bidiPins);
C syntax	typedef struct { DCM_STRING_ARRAY *inputPins, *outputPins, *bidiPins; } T_IO_results; int dpcmGetCellIOlists (const DCM_STD_STRUCT *std_struct, T_IO_results *rtn);

This returns the names of the input, output, and the bidirectional pins of the cell specified in the Standard Structure.

Where any input, output, or bidirectionals of a cell is a bus, the result returned by this function shall separately enumerate every bit of a bus. The pin names returned by this function call shall be used by the application for all calls referencing the specified cell name. A zero-length array is returned if the specified cell does not have any pins of the returned types (inputs, output, or bidirectionals).

10.18.2 Interconnect delay related functions

This subclause describes the interconnect delay related functions.

10.18.2.1 appGetRC

Table 132 provides information on appGetRC.

Table 132—appGetRC

Function name	appGetRC
Arguments	Driver (source) pin pointer, receiver (sink) pin pointer
Result	Interconnect RC delay
Standard Structure fields	block, cellName, calcMode
DCL syntax	EXTERNAL(appGetRC): passed(pin: fromPin, toPin) result(double: netSegRC);
C syntax	typedef struct { DCM_DOUBLE rcDelay; } T_rc; int appGetRC (DCM_STD_STRUCT *std_struct, T_rc *rtn, DCM_PIN fromPin, DCM_PIN toPin);

This returns the equivalent RC (Resistor Capacitor time constant, also known as the Elmore delay for pin to pin interconnect) value for the interconnect between *fromPin* and *toPin*.

BLOCK and *CellName* fields in the Standard Structure shall describe the driving circuit (*FROM_POINT* pin pointer).

The application shall return the value available from its model (e.g., a SPEF file). If no such value exists, the application shall call dpcmGetEstimateRC (see 10.18.2.4) to get an estimated value for the RC.

10.18.2.2 dpcmGetDelayGradient

Table 133 provides information on dpcmGetDelayGradient.

Table 133—dpcmGetDelayGradient

Function name	dpcmGetDelayGradient
Arguments	
Result	Rate of change of delay
Standard Structure fields	block
DCL syntax	<pre>EXPOSE(dpcmGetDelayGradient): result(double: LateRateofChangeWithRespectToOutputCapacitance, LateRateofChangeWithRespectToInputSlew, EarlyRateofChangeWithRespectToOutputCapacitance, EarlyRateofChangeWithRespectToInputSlew);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE LateRateofChangeWithRespectToOutputCapacitance, LateRateofChangeWithRespectToInputSlew, EarlyRateofChangeWithRespectToOutputCapacitance, EarlyRateofChangeWithRespectToInputSlew; } T_RecDel; int dpcmGetDelayGradient (const DCM_STD_STRUCT *std_struct, T_RecDel *rtn);</pre>

This call shall be made directly following a call for *delay*. It returns the delay gradient (rate of change for the delay) associated with the most recent calculation of delay.

10.18.2.3 dpcmGetSlewGradient

Table 134 provides information on dpcmGetSlewGradient.

Table 134—dpcmGetSlewGradient

Function name	dpcmGetSlewGradient
Arguments	None
Result	Rate of change of slew
Standard Structure fields	block
DCL syntax	<pre>EXPOSE(dpcmGetSlewGradient): result(double: LateRateofChangeWithRespectToOutputCapacitance, LateRateofChangeWithRespectToInputSlew, EarlyRateofChangeWithRespectToOutputCapacitance, EarlyRateofChangeWithRespectToInputSlew);</pre>
C syntax	<pre>typedef struct {DCM_DOUBLE LateRateofChangeWithRespectToOutputCapacitance, LateRateofChangeWithRespectToInputSlew, EarlyRateofChangeWithRespectToOutputCapacitance, EarlyRateofChangeWithRespectToInputSlew; } T_RecSlew; int dpcmGetSlewGradient(const DCM_STD_STRUCT *std_struct, T_RecSlew *rtn);</pre>

This call shall be made directly following a call for *slew*. It returns the slew gradient (rate of change for the slew) associated with the most recent calculation of slew.

10.18.2.4 dpcmGetEstimateRC

Table 135 provides information on dpcmGetEstimateRC.

Table 135—dpcmGetEstimateRC

Function name	dpcmGetEstimateRC
Arguments	Driver (source) pin pointer, Receiver (sink) pin pointer
Result	Interconnect RC delay
Standard Structure fields	block, CellName, calcMode, pathData (timing-pin-specific), cellData (timing)
DCL syntax	<pre>EXPOSE(dpcmGetEstimateRC): passed(pin: fromPin,toPin) result(double: netSegRC);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE netSegRC; } T_estRC; int dpcmGetEstimateRC (const DCM_STD_STRUCT *std_struct, T_estRC *rtn, DCM_PIN fromPin, DCM_PIN toPin);</pre>

This returns a estimated interconnect RC delay value when the application does not know what the interconnect RC delay value is for the current pin pair.

BLOCK and *CellName* fields in the Standard Structure shall describe the driving circuit (*FROM_POINT* pin pointer).

10.18.2.5 dpcmGetDefPortSlew

Table 136 provides information on dpcmGetDefPortSlew.

Table 136—dpcmGetDefPortSlew

Function name	dpcmGetDefPortSlew
Arguments	None
Result	Default slew
Standard Structure fields	calcMode
DCL syntax	<pre>EXPOSE(dpcmGetDefPortSlew): result(double: defSlew);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE defSlew; } T_defPortSlew; int dpcmGetDefPortSlew (const DCM_STD_STRUCT *std_struct, T_defPortSlew *rtn);</pre>

This returns a default value for the slew of a signal presented to a chip primary input.

10.18.2.6 dpcmGetDefPortCapacitance

Table 137 provides information on dpcmGetDefPortCapacitance.

Table 137—dpcmGetDefPortCapacitance

Function name	dpcmGetDefPortCapacitance
Arguments	None
Result	Default capacitance
Standard Structure fields	calcMode
DCL syntax	EXPOSE(dpcmGetDefPortCapacitance): result(double: defPortCap);
C syntax	typedef struct { DCM_DOUBLE defPortCap; } T_defPortCap; int dpcmGetDefPortCapacitance (const DCM_STD_STRUCT *std_struct, T_defPortCap *rtn);

This returns a default value of the capacitance load on a chip primary output.

10.18.3 Functions accessing netlist information

This subclause lists the functions that access netlist information.

10.18.3.1 appGetNumDriversByPin

Table 138 provides information on appGetNumDriversByPin.

Table 138—appGetNumDriversByPin

Function name	appGetNumDriversByPin
Arguments	Pin pointer
Result	Number of source pins on the interconnect
Standard Structure fields	None
DCL syntax	EXTERNAL(appGetNumDriversByPin): passed(pin: inputPin) result(int: numDrivers);
C syntax	typedef struct { INTEGER drivers; } T_numDriversByPin; int appGetNumDriversByPin (DCM_STD_STRUCT *std_struct, T_numDriversByPin *rtn, DCM_PIN inputPin);

This returns the number of driver (source) pins on the interconnect to which the passed pin is connected. This count includes all interconnect drivers, including bidirectional pins.

10.18.3.2 appGetNumDriversByName

Table 139 provides information on appGetNumDriversByName.

Table 139—appGetNumDriversByName

Function name	appGetNumDriversByName
Arguments	Pin name
Result	Number of driver (source) pins on the interconnect
Standard Structure fields	block

DCL syntax	EXTERNAL (appGetNumDriversByName) : passed(string: inputPin) result(int: numDrivers);
C syntax	typedef struct { INTEGER drivers; } T_numDriversByName; int appGetNumDriversByName (DCM_STD_STRUCT *std_struct, T_numDriversByName *rtn, STRING inputPin);

This returns the number of driver (source) pins on the interconnect to which the passed pin name is connected. This count includes all interconnect drivers, including bidirectional pins.

10.18.3.3 appForEachParallelDriverByPin

Table 140 provides information on appForEachParallelDriverByPin.

Table 140—appForEachParallelDriverByPin

Function name	appForEachParallelDriverByPin
Arguments	Pin pointer, function pointer to call at each parallel driver pin, function pointer to call to perform an operation on the data, initial value
Result	Integer count of drivers working in parallel with the pin passed as the argument. Computed value
Standard Structure fields	calcMode
DCL syntax	<pre>FORWARD CALC (EXPOSE STATEMENT) : passed(pin: parallelPin) result(double: exposeValue); FORWARD CALC (OPERATOR STATEMENT) : passed(double: initialValue, exposeValue) result(double: computedValue); EXTERNAL (appForEachParallelDriverByPin) : passed(pin: outputPin; exposeStatement() : anyStatementThatHasTheSamePrototypeSignature; operatorStatement() : anyStatementThatHasTheSamePrototypeSignature; double: initialValue) result(int: parallelDriverCount; double: computedValue);</pre>

C syntax	<pre> typedef struct { INTEGER parallelDriverCount; DCM_DOUBLE computedValue; } T_ParaDrvPin; typedef int(exposeType*) (DCM_STD_STRUCT *std_struct, DCM_DOUBLE *exposeValue, DCM_PIN parallelPin); typedef int(operatorType*) (DCM_STD_STRUCT *std_struct, DCM_DOUBLE *resultValue, DCM_DOUBLE initialValue, DCM_DOUBLE exposeValue); int appForEachParallelDriverByPin (DCM_STD_STRUCT *std_struct, T_ParaDrvPin *rtn, DCM_PIN outputPin, exposeType expose_function, operatorType operator_function, DCM_DOUBLE initialValue); </pre>

This performs a callback (to DPCM) for each driver on the interconnect to which the passed pin is connected that is parallel to the pin. Parallel drivers are those pins on the interconnect that belong to cells on the interconnect and are wired identically to the cell of the passed pin.

For example, in Figure 5, *PinA* and *PinB* are parallel drivers, whereas *PinC* and *PinD* are not.

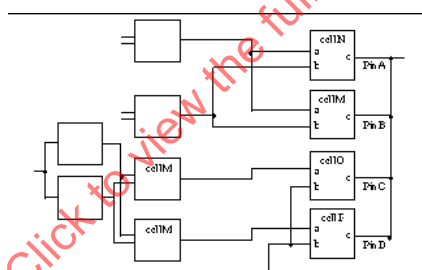


Figure 5—Parallel drivers example

appForEachParallelDriverByPin is called with a pin, two function pointers, and an initial value. The first function pointer is for an *EXPOSE* function, which shall accept one passed argument and return a double. The second function pointer is for an *OPERATOR* function, which shall accept two doubles representing the initial accumulated value (which is generated from the previous call to this function) and the value returned from an expression value (computed by an *EXPOSE* function). It returns a double representing the computed value. The application, at each parallel driver pin, calls the DPCM-supplied *EXPOSE* function if the pointer is not 0 (zero).

After the *EXPOSE* function computes its value, the application calls the *OPERATOR* function and passes it two values. The first value (*initialValue*) represents the initial value, which may have been derived from the last call to the *OPERATOR* function or, on the first call, the initial value passed as arguments to *appForEachParallelDriverByPin*. The next parameter (*exposeValue*) is the return value from the most recently called *EXPOSE* function.

The purpose of the *EXPOSE* and *OPERATOR* function pointers is to allow the DPCM to define computations that shall be performed on the parallel driver pins and to have that processing be executed as part of *appForEachParallelDriverByPin*.

Example

To compute the average slew value for the parallel drivers, the following shall occur:

- a) *appForEachParallelDriverByPin* is called (from the DPCM) with the following:
 - 1) A specified source pin.
 - 2) A function pointer to a DPCM function (*EXPOSE*) that computes the slew for the pin specified to it.
 - 3) A function pointer to a DPCM function (*OPERATOR*) that adds its passed *initialWorstCaseValue* to its passed *exposeValue*, adds its passed *initialValue* to its passed *exposeValue*, and returns the sum.
 - 4) A new value for the slew of the specified pin.
- b) On return from the completed *appForEachParallelDriverByPin* function, the DPCM is passed the *computedValue* result argument that represents the sum of the slew for all the parallel drivers and the number of parallel drivers (*parallelDriverCount*). The DPCM can then divide the resulting slew values by the number of parallel drivers to compute an average value for the slews.

appForEachParallelDriverByPin also records the number of parallel drivers it encounters and returns the number of parallel drivers (*parallelDriverCount*) and the last values returned by *computedValue*. If the function pointers passed to *appForEachParallelDriverByPin* are 0 (zero), the function returns the number of parallel drivers and the passed initial value arguments.

10.18.3.4 appForEachParallelDriverByName

Table 141 provides information on *appForEachParallelDriverByName*.

Table 141—appForEachParallelDriverByName

Function name	<i>appForEachParallelDriverByName</i>
Arguments	Pin name, Function pointer to call at each parallel driver pin, Function pointer to call to perform an operation on the data, Initial value
Result	Integer count of drivers working in parallel with the pin passed as the argument, Computed value
Standard Structure fields	calcMode, block
DCL syntax	<pre> FORWARD CALC (EXPOSE_STATEMENT) : passed(string: outputPin) result(double: resultValue); FORWARD CALC (OPERATOR_STATEMENT) : passed(double: initialValue, exposeValue) result(double: computedValue); EXTERNAL (appForEachParallelDriverByName) : passed(string: outputPin; exposeStatement() : anyStatementThatHasTheSamePrototypeSignature; operatorStatement() : anyStatementThatHasTheSamePrototypeSignature; double: initialValue) result(int: parallelDriverCount; double: computedValue); </pre>

C syntax	<pre> typedef struct { DCM_INTEGER parallelDriverCount; DCM_DOUBLE computedValue } T_ParaDrivName; typedef int (*exposeType) (DCM_STD_STRUCT *std_struct, DCM_DOUBLE *resultValue, DCM_STRING outputPin); typedef int (*operatorType) (DCM_STD_STRUCT *std_struct, DCM_DOUBLE *resultValue, DCM_DOUBLE initialValue, DCM_DOUBLE exposeValue); int appForEachParallelDriverByName (DCM_STD_STRUCT *std_struct, T_ParaDrivName *rtn, const DCM_STRING outputPin, exposeType expose_function, operatorType operator_function, DCM_DOUBLE initialValue); </pre>

appForEachParallelDriverByName is identical to *appForEachParallelDriverByPin*, except the argument *outputPin* is of type *STRING* and contains the name of the pin on the cell for which this call applies.

10.18.3.5 appGetNumPinsByPin

Table 142 provides information on *appGetNumPinsByPin*.

Table 142—*appGetNumPinsByPin*

Function name	<i>appGetNumPinsByPin</i>
Arguments	Pin pointer
Result	Total number of pins on the interconnect
Standard Structure fields	None
DCL syntax	<pre> EXTERNAL (appGetNumPinsByPin) : passed (pin: outputPin) result (int: totalPins); </pre>
C syntax	<pre> typedef struct { INTEGER totalPins; } T_NumPinsByPin; int appGetNumPinsByPin (DCM_STD_STRUCT *std_struct, T_NumPinsByPin *rtn, DCM_PIN outputPin); </pre>

This returns the total number of pins (all driver and receiver pins, including the passed pin) on the interconnect to which the passed pin is connected. An *outputPin* value of 0 (zero) shall be legal as it may result from a 0 (zero) pin pointer passed to *dpcmGetEstWireCapacitance* or *dpcmGetEstWireResistance*.

10.18.3.6 appGetNumPinsByName

Table 143 provides information on *appGetNumPinsByName*.

Table 143—appGetNumPinsByName

Function name	appGetNumPinsByName
Arguments	Pin name
Result	Total number of pins on the interconnect
Standard Structure fields	block
DCL syntax	EXTERNAL (appGetNumPinsByName) : passed(string: outputPin) result(integer: totalPins);
C syntax	typedef struct { INTEGER totalPins; } T_NumPinsByName; int appGetNumPinsByName (DCM_STD_STRUCT *std_struct, T_NumPinsByName *rtn, STRING outputPin);

This returns the total number of pins (all driver and receiver pins, including the passed pin) on the interconnect to which the passed pin name is connected.

10.18.3.7 appGetNumSinksByPin

Table 144 provides information on appGetNumSinksByPin.

Table 144—appGetNumSinksByPin

Function name	appGetNumSinksByPin
Arguments	Pin pointer
Result	Total number of sink pins on the interconnect
Standard Structure fields	
DCL syntax	EXTERNAL (appGetNumSinksByPin) : passed(pin: outputPin) result(int: totalLoadPins);
C syntax	typedef struct { INTEGER totalLoadPins; } T_NumSinksByPin; int appGetNumSinksByPin (DCM_STD_STRUCT *std_struct, T_NumSinksByPin *rtn, DCM_PIN outputPin);

This returns the total number of load (sink) pins (including bidirectional pins) on the interconnect to which the passed pin is connected. An outputPin value of 0 (zero) shall be legal as it may result from a 0 (zero) pin pointer passed to dpcmGetEstWireCapacitance or dpcmGetEstWireResistance.

10.18.3.8 appGetNumSinksByName

Table 145 provides information on appGetNumSinksByName.

Table 145—appGetNumSinksByName

Function name	appGetNumSinksByName
Arguments	Pin name
Result	Total number of sink pins on the interconnect
Standard Structure fields	block

DCL syntax	EXTERNAL (appGetNumSinksByName) : passed(string: outputPin) result(int: totalLoadPinsByName);
C syntax	typedef struct { INTEGER totalLoadPins; } T_NumSinksByName; int appGetNumSinksByName (DCM_STD_STRUCT *std_struct, T_NumSinksByName *rtn, STRING outputPin);

This returns the total number of load (sink) pins (including bidirectional pins) on the interconnect to which the passed pin name is connected.

10.18.3.9 dpcmAddWireLoadModel

Table 146 provides information on dpcmAddWireLoadModel.

Table 146—dpcmAddWireLoadModel

Function name	dpcmAddWireLoadModel
Arguments	Model name, Extrapolation constant, Unit resistance, Unit capacitance, Length matrix
Result	Model index
Standard Structure fields	calcMode
DCL syntax	EXPOSE (dpcmAddWireLoadModel) : passed(string: modelName; double: extrapolationConstant; double[*]: unitResistance, unitCapacitance, lengthMatrix) result(int: modelIndex);
C syntax	typedef struct { INTEGER modelIndex; } T_AddWireLoadModel; int dpcmAddWireLoadModel (const DCM_STD_STRUCT *std_struct, T_AddWireLoadModel *rtn, STRING modelName, DCM_DOUBLE extrapolationConstant, DCM_DOUBLE_ARRAY *unitResistance, DCM_DOUBLE_ARRAY *unitCapacitance, DCM_DOUBLE_ARRAY *lengthMatrix);

This function adds custom wire load information to the DPCM. Custom wire load information can be written into the DPCM with this call. Unit resistance, unit capacitance, an extrapolation constant, and a length matrix are passed with this call. The length matrix is indexed by fanout. Thus, for a given fanout, a unit length can be determined. Using this length, resistance and capacitance can be determined. If fanout exceeds the dimensions of the array, then the extrapolation constant shall be used.

The DPCM shall return a model index such that the application can inform the DPCM which custom wire load model it can use. The model index returned shall be unique across all wire load models (custom and default). If unit resistance or unit capacitance does not vary with length, then the respective result array may be of length 1. If unit resistance or unit capacitance array does vary with length, then the respective result array shall be of the same length as the *lengthMatrix* array and is indexed by fanout.

An application-supplied wire load model whose name matches a wire load model name already in this array shall have the following affect:

- If the name matches a wire load model supplied with the library (a default wire load model), then an error is generated and no change in the data occurs.
- If the name matches a wire load model previously supplied by the application, then the previous wire load model data are replaced and the same index number in this array is used for this new wire load model.

The DPCM shall not choose a custom wire load model that was added by the application.

All elements of the *lengthMatrix* array shall have valid data.

10.18.3.10 dpcmGetWireLoadModel

Table 147 provides information on dpcmGetWireLoadModel.

Table 147—dpcmGetWireLoadModel

Function name	dpcmGetWireLoadModel
Arguments	Model index
Result	Extrapolation constant, Unit resistance per length, Unit capacitance per length, Length matrix, Size
Standard Structure fields	calcMode
DCL syntax	<pre>EXPOSE(dpcmGetWireLoadModel): passed(int: modelIndex) result(double: extrapolationConstant; double[*]: unitResistance, unitCapacitance, lengthMatrix; double: size);</pre>
C syntax	<pre>typedef struct{ double extrapolationConstant; DCM_DOUBLE_ARRAY *unitResistance, *unitCapacitance, *lengthMatrix; DCM_DOUBLE size; } T_GetWireLoadModel; int dpcmGetWireLoadModel (const DCM_STD_STRUCT *std_struct, T_GetWireLoadModel *rtn, INTEGER modelIndex);</pre>

This function transfers a wire load model to the application. A wire load model is selected by *modelIndex*, which indicates an element of the array returned by *dpcmGetWireLoadModelArray*. The data that are given to the application consist of unit resistance, unit capacitance, and length matrix arrays that are indexed by fanout, an extrapolation constant, and the number of cells in a block. If unit resistance or unit capacitance does not vary with length, then the respective result array may be of length 1. If unit resistance or unit capacitance array does vary with length, then the respective result array shall be of the same length as the *lengthMatrix* array. The size shall be zero for user-supplied wire load models.

Size is an abstract quantity associated with a physical block for which the wire load model, estimated capacitance, estimated resistance, and estimated RC apply. Size is an abstract measure for the block area, height, width, perimeter, or any combination of these. Size is dimensionless, as opposed to area, length, width, perimeter, or height. A calculation algorithm for size shall be defined in the library but not be

exposed to the application. Size can be defined as a function of the physical block area, width and height; in which case, size shall be monotonically increasing with any of these quantities. Verification of the monotonicity shall be the task of the library's quality assurance, not the application's.

10.18.3.11 dpcmGetWireLoadModelForBlockSize

Table 148 provides information on dpcmGetWireLoadModelForBlockSize.

Table 148—dpcmGetWireLoadModelForBlockSize

Function name	dpcmGetWireLoadModelForBlockSize
Arguments	Size
Result	Array index of current wire load models according to size
Standard Structure fields	calcMode
DCL syntax	<pre>EXPOSE (dpcmGetWireLoadModelForBlockSize) : passed(double: size) result(int: modelIndex);</pre>
C syntax	<pre>typedef struct { int modelIndex; } T_GetWireLoadModelForBlockSize; int dpcmGetWireLoadModelForBlockSize (const DCM_STD_STRUCT *std_struct, T_GetWireLoadModelForBlockSize *rtn, DCM_DOUBLE size);</pre>

This call requests the index number of the appropriate wire load model given the specified size from the DPCM. This function only considers the wire load models delivered with the library when determining which model to return. Application-supplied wire load models shall not be selected via this function.

The wire load model is used to estimate interconnect delay. To determine the size to be passed into this function, the application shall determine the highest level hierarchical block that contains all the endpoints of the net for which the model is being requested. The area is determined by summing the area of each cell in this hierarchical unit.

10.18.3.12 appGetInstanceCount

Table 149 provides information on appGetInstanceCount.

Table 149—appGetInstanceCount

Function name	appGetInstanceCount
Arguments	None
Result	Count of instances
Standard Structure fields	toPoint
DCL syntax	<pre>EXTERNAL (appGetInstanceCount) : result(int: countOfInstances);</pre>
C syntax	<pre>typedef struct { INTEGER countOfInstances; } T_InstanceCount; int appGetInstanceCount (DCM_STD_STRUCT *std_struct, T_InstanceCount *rtn);</pre>

This returns the number of cell instances found in the cluster(s) or floorplanned region(s) in which the interconnect specified by the *toPoint* resides.

NOTE—This particular PI function is used by the DPCM to estimate interconnect delay.

10.18.4 Functions exporting limit information

This subclause lists the functions that export limit information.

10.18.4.1 dpcmGetCapacitanceLimit

Table 150 provides information on dpcmGetCapacitanceLimit.

Table 150—dpcmGetCapacitanceLimit

Function name	dpcmGetCapacitanceLimit
Arguments	Pin name
Result	Maximum capacitance value, Minimum capacitance value
Standard Structure fields	CellName, block, calcMode, pathData (timing-pin-specific), cellData (timing)
DCL syntax	EXPOSE(dpcmGetCapacitanceLimit): passed(string: pinName) result(double: lowerLimit, upperLimit);
C syntax	typedef struct { DCM_DOUBLE lowerLimit, upperLimit; } T_minMaxCap; int dpcmGetCapacitanceLimit (const DCM_STD_STRUCT *std_struct, T_minMaxCap *rtn, STRING pinName);

This returns the minimum and maximum capacitance the passed pin name is allowed to drive. The passed pin shall be either an output or a bidirectional pin.

NOTE—The intent of this function is to enable an application to help ensure a cell operates within its design limits.

10.18.4.2 dpcmGetSlewLimit

Table 151 provides information on dpcmGetSlewLimit.

Table 151—dpcmGetSlewLimit

Function name	dpcmGetSlewLimit
Arguments	Pin name, Transition type
Result	Minimum slew value, Maximum slew value
Standard Structure fields	CellName, block, pathData (timing-pin-specific), cellData (timing), calcMode
DCL syntax	EXPOSE(dpcmGetSlewLimit): passed(string: pinName, transitionType) result(double: lowerLimit, upperLimit);
C syntax	typedef struct { DCM_DOUBLE lowerLimit, upperLimit } T_slewLimits; int dpcmGetSlewLimit (const DCM_STD_STRUCT *std_struct, T_slewLimits *rtn, STRING pinName, STRING transitionType);

This returns the maximum and minimum slew limits for the passed pin name. Transition type is *F*, *R*, or *B* representing falling, rising, or both, respectively.

NOTE—The intent of this function is to help ensure a cell operates within its design limits.

10.18.4.3 dpcmGetXovers

Table 152 provides information on dpcmGetXovers.

Table 152—dpcmGetXovers

Function name	dpcmGetXovers
Arguments	Pin name
Result	Nominal capacitance, Slow capacitance, Fast capacitance
Standard Structure fields	CellName, block, pathData (timing-pin-specific), cellData (timing)
DCL syntax	<pre>EXPOSE(dpcmGetXovers): passed(string: pinName) result(double: nominal, slow, fast);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE nominal, slow, fast; } T_nsfCap; int dpcmGetXovers (const DCM_STD_STRUCT *std_struct, T_nsfCap *rtn, STRING pinName);</pre>

This returns the drive strengths (load capacitance limits at which design applications, such as synthesis, switch to specific cell drive strengths) for the cell identified in the Standard Structure with which the passed pin name is associated. The three capacitance values are alternatives for three different process/voltage/temperature (PVT) cases chosen by the library developer.

10.18.5 Functions getting/setting model information

This subclause describes the functions that get or set model information.

10.18.5.1 dpcmGetFunctionalModeArray

Table 153 provides information on dpcmGetFunctionalModeArray.

Table 153—dpcmGetFunctionalModeArray

Function name	dpcmGetFunctionalModeArray
Arguments	none
Result	Array of functional mode group names, Array of functional mode names
Standard Structure fields	CellName
DCL syntax	<pre>EXPOSE(dpcmGetFunctionalModeArray): result(string[*]:modeGroupArray, modeNameArray);</pre>
C syntax	<pre>typedef struct { DCM_STRING_ARRAY *modeGroupArray, *modeNameArray; } T_fModes; int dpcmGetFunctionalModeArray (const DCM_STD_STRUCT *std_struct, T_fModes *rtn);</pre>

A cell may have zero or more groups of functional modes. Each group and each functional mode in that group has a name. The functional mode of a cell identified by specifying the value of *modeNameArray* for all mode groups requested by the DPCM. For example, if the DPCM requests the mode group values for all mode groups of a cell, then the cell's mode is the aggregate of the *modeNameArray* values specified.

This call requests the names of the functional mode groups and the names of the functional modes for the cell specified in the Standard Structure from the DPCM. The DPCM can elect to return two zero-length arrays to indicate the cell has only one mode.

For cells with multiple functional modes, the *i*th element of *modeGroupArray* shall contain the name of the *i*th functional mode group and the *i*th element of *modeNameArray* shall contain a comma-delimited list of the names of functional modes in that group. Group and function mode names shall not contain embedded white space.

When the application requests the default functional mode by calling *dpcmGetBaseFunctionalMode* or the DPCM requests the current functional mode by calling *appGetCurrentFunctionalMode*, the *functionalModeGroup* is specified as an index into *modeGroupArray*. The returned index selects an element from the corresponding *modeNameArray*, where the first mode has an index value of 0.

Example

Consider a cell that contains the functional mode groups *rw* (containing modes *read* and *write*) and *latch_type* (containing modes *latching* and *transparent*). A call to *dpcmGetFunctionalModeArray* returns:

```
modeGroupArray[0] = "rw"
modeNameArray[0] = "read,write"
modeGroupArray[1] = "latch_type"
modeNameArray[1] = "latching,transparent"
```

10.18.5.2 dpcmGetBaseFunctionalMode

Table 154 provides information on *dpcmGetBaseFunctionalMode*.

Table 154—*dpcmGetBaseFunctionalMode*

Function name	<i>dpcmGetBaseFunctionalMode</i>
Arguments	<i>FuncModeGroupIndex</i>
Result	Index of the default functional mode
Standard Structure fields	<i>CellName</i>
DCL syntax	<pre>EXPOSE(dpcmGetBaseFunctionalMode) : passed(int: FuncModeGroupIndex) result(int: modeIndex);</pre>
C syntax	<pre>typedef struct { INTEGER modeIndex; } T_BaseFunctionalMod; int dpcmGetBaseFunctionalMode (const DCM_STD_STRUCT *std_struct, T_BaseFunctionalMod *rtn, INTEGER FuncModeGroupIndex);</pre>

This call specifies a cell (in the Standard Structure) and a functional mode group index (which indicates one of the functional mode groups returned by *dpcmGetFunctionalModeArray*) and returns the index number of the default functional mode for that cell and functional mode group. The returned *modeIndex* value shall be between 0 and *n* – 1 (where *n* is the number of modes for the functional mode group in the specified cell) and the first mode has index value 0.

NOTE—This number can be used to index into the *modeNameArray* returned by *dpcmGetFunctionalModeArray* to retrieve the name of the default mode.

10.18.5.3 appGetCurrentFunctionalMode

Table 155 provides information on *appGetCurrentFunctionalMode*.

Table 155—appGetCurrentFunctionalMode

Function name	<i>appGetCurrentFunctionalMode</i>
Arguments	<i>FuncModeGroupIndex</i>
Result	Index of the current functional mode
Standard Structure fields	block
DCL syntax	<pre>EXTERNAL (appGetCurrentFunctionalMode) : passed (int: FuncModeGroupIndex) result (int: modeIndex);</pre>
C syntax	<pre>typedef struct { INTEGER modeIndex; } T_modeIndex; int appGetCurrentFunctionalMode (DCM_STD_STRUCT *std_struct, T_modeIndex *rtn, INTEGER FuncModeGroupIndex);</pre>

This call requests the current functional mode for the specified functional mode group index (which indicates one of the functional mode groups returned by *dpcmGetFunctionalModeArray*) and of the cell instance identified in the Standard Structure. The returned index selects an element from the *modeNameArray* corresponding to the specified functional mode group, where the first mode has index value 0. If no functional modes are defined for this cell instance, then a *modeIndex* value of *−1* shall be returned.

10.18.5.4 dpcmGetControlExistence

Table 156 provides information on *dpcmGetControlExistence*.

Table 156—dpcmGetControlExistence

Function name	<i>dpcmGetControlExistence</i>
Arguments	none
Result	Functional Modes, Expression
Standard Structure fields	fromPoint, toPoint, pathData (timing-arc-specific), cellData (timing), block
DCL syntax	<pre>EXPOSE (dpcmGetControlExistence) : result (integer[*]: FunctionalModes; string: Expression);</pre>
C syntax	<pre>typedef struct { DCM_INTEGER_ARRAY *FunctionalModes; char *Expression; } T_ControlExistence; int dpcmGetControlExistence (const DCM_STD_STRUCT *std_struct, T_ControlExistence *rtn);</pre>

This returns to the following application information that controls the existence of the segment identified by *pathData*:

- a) The integers returned through the *FunctionalModes* result in encoding the elements of the *modeGroupArray* and *modeNameArray* returned by *dpcmGetFunctionalModeArray* for which the segment exists.
- b) The *FunctionalModes* array contains zero or more contiguous integer sequences. For each sequence, the following occurs:
 - 1) The first element value, *v1*, indicates functional mode group *modeGroupArray*[*v1*].
 - 2) The second element value, *v2*, specifies how many functional modes follow in the sequence.
 - 3) The remaining elements (equal in number to *v2*) indicate modes in *modeNameArray*[*v1*].
- c) The string returned through the *Expression* result is a *ConditionalExpression* using the syntax and semantics of the “Group Condition Language” (see 8.11) and the segment shall exist unless the expression evaluates to FALSE.

A zero-length *FunctionalModes* result indicates there is no controlling functional mode or the controlling functional mode information is not known. A zero-length *Expression* indicates a controlling expression does not exist or is not known.

If possible, an expression shall be used to decide the existence of a segment. If the application can evaluate expressions and a non-zero-length *Expression* is returned, the application shall use that *Expression* to decide the existence of the segment.

10.18.5.5 dpcmSetLevel

Table 157 provides information on *dpcmSetLevel*.

Table 157—dpcmSetLevel

Function name	dpcmSetLevel
Arguments	Desired DPCM computation mode (performance or accuracy), PVT derating and other scopes
Result	Previous DPCM computation mode (performance or accuracy) and scopes
Standard Structure fields	calcMode
DCL syntax	<pre>EXPOSE(dpcmSetLevel): passed(int: perfLevel, temperatureScope, voltageScope, functionalModeScope, wire load, ModelScope) result(int: oldPerfLevel, oldtemperatureScope, oldvoltageScope, oldfunctionalModeScope, oldwireloadModelScope);</pre>

C syntax	<pre> typedef struct old_per_level { DCM_INTEGER oldPerfLevel, oldtemperatureScope, oldvoltageScope, oldfunctionalModeScope, oldwireloadModelScope; } T_oldPerfLevels; int dpcmSetLevel (const DCM_STD_STRUCT *std_struct, T_oldPerfLevels *rtn, DCM_INTEGER perfLevel, DCM_INTEGER temperatureScope, DCM_INTEGER voltageScope, DCM_INTEGER functionalModeScope, DCM_INTEGER wireloadModelScope); </pre>

The application can use *dpcmGetExposePurityAndConsistency* to determine which *EXPOSE* functions provide data the application can cache.

This function instructs the DPCM in two ways. The first parameter (*perfLevel*) instructs the DPCM to perform calculations to maximum supported accuracy or at a lesser accuracy in favor of computation speed. The *perfLevel* switch affects both timing and power calculations. The subsequent *Scope* parameters can be used to indicate to the DPCM how constant the temperature, voltage, functional mode, and wire load model settings are for this run.

The values of the *perfLevel* parameter are as follows:

- 0—Indicates calculations for maximum performance
- 1—Indicates calculations for maximum accuracy

The values of the *Scope* parameters are as follows:

- 0—This condition applies to all cell instances equally.
- 1—This condition can apply to each cell instance uniquely.

For *Scope* parameters set to zero, the DPCM can choose to cache the value it receives on the first callback to the application for this information and avoid subsequent callbacks.

Whenever a call to this function (*dpcmSetLevel*) is made, the following actions shall occur:

- a) The DPCM shall invalidate its caching, if any, of the *Scope* parameter values and is required to query the application again for this information prior to any calculations using this information.
- b) If the DPCM supports multiple operating ranges, then the DPCM shall query the application for the current operating range value (via *appGetCurrentOpRange*).

Calls to *dpcmSetLevel* shall not cause any changes in the DPCM that require model elaboration.

10.18.5.5.1 Accuracy levels

Additional control of library accuracy has been enhanced with the *dpcmSetLibraryAccuracyLevel* API. The DPCM can implement various algorithms that trade off performance versus accuracy. The list of these algorithms is exposed to the application via the *dpcmGetAccuracyLevelArrays* API. The DPCM is responsible for the semantic description of each level. The application shall obtain the names of the accuracy levels to use during analysis from the user.

The library shall ignore *dpcmSetLibraryAccuracyLevel* if *perfLevel* has not been set to 1 by a call to *dpcmSetLevel*. If *perfLevel* is set to 1, the *dpcmSetLibraryAccuracyLevel* values apply. If the application does not set *dpcmSetLibraryAccuracyLevel*, the DPCM shall pick a default behavior. This provides a backward compatibility to IEEE Std 1481-2009 for applications that do not support the new APIs.

NOTE 1—For example, if different delay equations are used between high accuracy mode and high performance mode, then both of these equations shall be modeled during the initial elaboration.

NOTE 2—Since no re-elaboration of models is required due to a change in `dpcmSetLevel`, the model writer shall consider what is STORED to support high accuracy versus high performance modes. Setting levels in the library.

This subclause details how to set levels within a library.

10.18.5.6 dpcmGetLibraryAccuracyLevelArrays

Table 158 provides information on `dpcmGetLibraryAccuracyLevelArrays`.

Table 158—dpcmGetLibraryAccuracyLevelArrays

Function name	<code>dpcmGetLibraryAccuracyLevelArrays</code>
Arguments	none
Result	List of cell accuracy levels, List of interconnect accuracy levels
Standard Structure fields	None
DCL syntax	<pre>EXPOSE(dpcmGetLibraryAccuracyLevelArrays): result(string[*]: cellAccuracyLevel, interconnectAccuracyLevel);</pre>
C syntax	<pre>typedef struct { DCM_STRING_ARRAY *cellAccuracyLevel; DCM_STRING_ARRAY *interconnectAccuracyLevel; } T_GetLibraryAccuracyLevelArrays; int dpcmGetLibraryAccuracyLevelArrays (const DCM_STD_STRUCT *std_struct, T_GetLibraryAccuracyLevelArrays *rtn);</pre>

This returns the accuracy levels back to the application.

10.18.5.7 dpcmSetLibraryAccuracyLevel

Table 159 provides information on `dpcmSetLibraryAccuracyLevel`.

Table 159—dpcmSetLibraryAccuracyLevel

Function name	<code>dpcmSetLibraryAccuracyLevel</code>
Arguments	Cell accuracy level, Interconnect accuracy level
Result	Previous cell accuracy level, Previous interconnect accuracy level
Standard Structure fields	None
DCL syntax	<pre>EXPOSE(dpcmSetLibraryAccuracyLevel): passed(int: cellAccuracyLevel, interconnectAccuracyLevel) result(int: oldCellAccuracyLevel, oldInterconnectAccuracyLevel);</pre>

C syntax	<pre>typedef struct { DCM_INTEGER oldCellAccuracyLevel; DCM_INTEGER oldInterconnectAccuracyLevel; } T_SetLibraryAccuracyLevel; int dpcmSetLibraryAccuracyLevel (const DCM_STD_STRUCT *std_struct, T_SetLibraryAccuracyLevel *rtn, DCM_INTEGER cellAccuracyLevel, DCM_INTEGER interconnectAccuracyLevel);</pre>

This sets the accuracy levels to be used by the library until the next call to *dpcmSetLibraryAccuracyLevel*. The arguments are used as the indices for the arrays returned by *dpcmGetLibraryAccuracyLevelArrays*.

10.18.5.8 dpcmGetExposePurityAndConsistency

Table 160 provides information on dpcmGetExposePurityAndConsistency.

Table 160—dpcmGetExposePurityAndConsistency

Function name	dpcmGetExposePurityAndConsistency
Arguments	EXPOSE API name
Result	Purity, Consistency
Standard Structure fields	None
DCL syntax	<pre>EXPOSE(dpcmGetExposePurityAndConsistency): passed(string: exposeName) result(int: purity, consistency);</pre>
C syntax	<pre>typedef enum DCM_Purity { DCM_IMPURE, DCM_PURE } DCM_Purity; typedef enum DCM_Consistency { DCM_INCONSISTENT, DCM_CONSISTENT } DCM_Consistency; typedef struct { DCM_Purity purity; DCM_Consistency consistency; } T_GetExposePurityAndConsistency; int dpcmGetExposePurityAndConsistency (const DCM_STD_STRUCT *std_struct, T_GetExposePurityAndConsistency *rtn, DCM_STRING exposeName);</pre>

This allows the application to determine which *EXPOSE* functions provide data the application can cache. The resulting integer values are used where the application passes in the expose name and the DCM returns the enumerated values, corresponding to *PURE* or *IMPURE* for purity and *CONSISTENT* or *INCONSISTENT* for consistency (Table 161 and Table 162).

Table 161—DCM_Purity

Enumerator	Enumeration	Description
DCM_IMPURE	0	Impure
DCM_PURE	1	Pure

Table 162—DCM_Consistency

Enumerator	Enumeration	Description
DCM_INCONSISTENT	0	Inconsistent
DCM_CONSISTENT	1	Consistent

10.18.5.9 dpcmGetRailVoltageArray

Table 163 provides information on dpcmGetRailVoltageArray.

Table 163—dpcmGetRailVoltageArray

Function name	dpcmGetRailVoltageArray
Arguments	None
Result	Array of rail voltage
Standard Structure fields	None
DCL syntax	EXPOSE (dpcmGetRailVoltageArray) : result(string[*]:railArray);
C syntax	typedef struct rail_array { DCM_STRING_ARRAY *railArray; } T_railArray; int dpcmGetRailVoltageArray (const DCM_STD_STRUCT *std_struct, T_railArray *rtn);

This call requests the voltage rail names that are modeled in the DPCM. A zero-length array can be returned by a library that does not model voltage.

When requesting the default voltage value for a particular voltage rail (via *dpcmGetBaseRailVoltage*) or when the DPCM is asking for the current voltage value for a particular voltage rail (via *appGetCurrentRailVoltage*), the index number into this array is used to identify the rail.

10.18.5.10 dpcmGetBaseRailVoltage

Table 164 provides information on dpcmGetBaseRailVoltage.

Table 164—dpcmGetBaseRailVoltage

Function name	dpcmGetBaseRailVoltage
Arguments	Integer index value for the rail
Result	Voltage value for the requested rail (volts)
Standard Structure fields	calcMode
DCL syntax	EXPOSE (dpcmGetBaseRailVoltage) : passed(int: railIndex) result(double:railVoltage);
C syntax	typedef struct { DCM_DOUBLE railVoltage; } T_railVoltage; int dpcmGetBaseRailVoltage (const DCM_STD_STRUCT *std_struct, T_railVoltage *rtn, INTEGER railIndex);

This call requests the default voltage value for the specified voltage rail. This value may be different for the different operating ranges.

10.18.5.11 appGetCurrentRailVoltage

Table 165 provides information on appGetCurrentRailVoltage.

Table 165—appGetCurrentRailVoltage

Function name	appGetCurrentRailVoltage
Arguments	Integer index value for the rail
Result	Voltage value for the current rail (volts)
Standard Structure fields	calcMode, block
DCL syntax	<pre>EXTERNAL (appGetCurrentRailVoltage) : passed(int: railIndex) result(double: railVoltage);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE railVoltage; } T_railVoltage; int appGetCurrentRailVoltage (DCM_STD_STRUCT *std_struct, T_railVoltage *rtn, INTEGER railIndex);</pre>

This call requests the voltage value for the specified rail from the application. If provided, this value shall then be used by the DPCM for its calculations and shall override the default or base voltage value for this rail.

10.18.5.12 dpcmGetWireLoadModelArray

Table 166 provides information on dpcmGetWireLoadModelArray.

Table 166—dpcmGetWireLoadModelArray

Function name	dpcmGetWireLoadModelArray
Arguments	None
Result	An array of wire load models
Standard Structure fields	calcMode
DCL syntax	<pre>EXPOSE (dpcmGetWireLoadModelArray) : result(string[*]: modelArray);</pre>
C syntax	<pre>typedef struct { DCM_STRING_ARRAY *modelArray; } T_WireLoadModelArray; int dpcmGetWireLoadModelArray (const DCM_STD_STRUCT *std_struct, T_WireLoadModelArray *rtn);</pre>

This call requests the wire load model names from the DPCM that are modeled in the DPCM. A zero-length array shall be returned by a library that does not contain any wire load models.

If the application supplies additional wire load models to the DPCM, then these additional models shall be added to the end of this array by the DPCM and returned on subsequent calls to this function. Adding new wire load models to the DPCM shall not affect the order of the previous wire load models in this array.

When requesting the default wire load model (*dpcmGetBaseWireLoadModel*) from the DPCM or when the DPCM is asking for the current wire load model (*appGetCurrentWireLoadModel*), the index number into this array is used to identify it.

10.18.5.13 dpcmGetBaseWireLoadModel

Table 167 provides information on *dpcmGetBaseWireLoadModel*.

Table 167—*dpcmGetBaseWireLoadModel*

Function name	<i>dpcmGetBaseWireLoadModel</i>
Arguments	None
Result	Array index of the default wire load models
Standard Structure fields	calcMode
DCL syntax	EXPOSE(<i>dpcmGetBaseWireLoadModel</i>) : result(int: modelIndex);
C syntax	typedef struct { INTEGER modelIndex; } T_BaseWireLoadModel; int <i>dpcmGetBaseWireLoadModel</i> (const DCM_STD_STRUCT *std_struct, T_BaseWireLoadModel *rtn);

This call requests the index number of the default wire load model for the library from the DPCM. If there are no wire load models, or if the library does not wish to specify a default, then a value of -1 shall be returned in model index. Otherwise, an index number between 0 and $n - 1$ (where n is the number of wire load models) is returned.

This number can be used to index into the array returned by *dpcmGetWireLoadModelArray* (see 10.18.5.12) to retrieve the default wire load model name.

10.18.5.14 appGetCurrentWireLoadModel

Table 168 provides information on *appGetCurrentWireLoadModel*.

Table 168—*appGetCurrentWireLoadModel*

Function name	<i>appGetCurrentWireLoadModel</i>
Arguments	Pin pointer
Result	Array index of the current wire load models to use
Standard Structure fields	calcMode
DCL syntax	EXTERNAL(<i>appGetCurrentWireLoadModel</i>) : passed(pin: pinPointer) result(int: modelIndex);
C syntax	typedef struct { INTEGER modelIndex; } T_CurrentWireLoadModel; int <i>appGetCurrentWireLoadModel</i> (DCM_STD_STRUCT *std_struct, T_CurrentWireLoadModel *rtn, DCM_PIN pinPointer);

This call requests from the application the current wire load model to be used in the DPCM's calculations. The index number (into the array returned by *dpcmGetWireLoadModelArray*) of the current wire load

model shall be returned. If no wire load models are defined for this library, then a value of -1 shall be returned in the model index.

10.18.5.15 dpcmGetBaseTemperature

Table 169 provides information on dpcmGetBaseTemperature.

Table 169—dpcmGetBaseTemperature

Function name	dpcmGetBaseTemperature
Arguments	None
Result	Default temperature
Standard Structure fields	calcMode
DCL syntax	EXPOSE (dpcmGetBaseTemperature) : result(double: temperature);
C syntax	typedef struct { DCM_DOUBLE temperature; } T_BaseTemperature; int dpcmGetBaseTemperature (const DCM_STD_STRUCT *std_struct, T_BaseTemperature *rtn);

This call requests the base temperature for the modeled library from the DPCM. This value may change depending on opRange.

10.18.5.16 dpcmGetBaseOpRange

Table 170 provides information on dpcmGetBaseOpRange.

Table 170—dpcmGetBaseOpRange

Function name	dpcmGetBaseOpRange
Arguments	None
Result	The index (from array) of the base operating range
Standard Structure fields	None
DCL syntax	EXPOSE (dpcmGetBaseOpRange) : result(int: opRangeIndex);
C syntax	typedef struct { INTEGER opRangeIndex; } T_BaseOpRange; int dpcmGetBaseOpRange (const DCM_STD_STRUCT *std_struct, T_BaseOpRange *rtn);

This call requests the index number of the default operating range name for the library from the DPCM. If no distinct operating range names are defined for this library, a value of -1 is returned as the index. Otherwise, an index number between 0 and $n - 1$ (where n is the number of operating ranges) is returned. This number can be used to index into the array returned by *dpcmGetOpRangeArray* to retrieve the default operating range name.

10.18.5.17 dpcmGetOpRangeArray

Table 171 provides information on dpcmGetOpRangeArray.

Table 171—dpcmGetOpRangeArray

Function name	dpcmGetOpRangeArray
Arguments	None
Result	Array of operating ranges
Standard Structure fields	None
DCL syntax	EXPOSE(dpcmGetOpRangeArray): result(string[*]:opRangeArray);
C syntax	typedef struct { DCM_STRING_ARRAY *opRangeArray; } T_OpRangeArray; int dpcmGetOpRangeArray (const DCM_STD_STRUCT *std_struct, T_OpRangeArray *rtn);

This call requests the operating range names modeled in the DPCM from the DPCM. Operating ranges refer to different characterization points of the library. For example, a library may be modeled at different PVT values for *MILITARY*, *COMMERCIAL*, or *INDUSTRIAL* applications.

There are no predefined operating range names. If the library does not have distinct operating ranges defined, the DPCM can elect to return a zero-length array. For libraries with multiple operating ranges, the returned array shall contain the identifying range names. Changing the operating range shall not require re-elaboration of the timing and power models.

10.18.5.18 appGetCurrentTemperature

Table 172 provides information on appGetCurrentTemperature.

Table 172—appGetCurrentTemperature

Function name	appGetCurrentTemperature
Arguments	None
Result	Current temperature
Standard Structure fields	block, calcMode
DCL syntax	EXTERNAL(appGetCurrentTemperature): result(double: temperature);
C syntax	typedef struct { DCM_DOUBLE temperature; } T_CurrentTemperature; int appGetCurrentTemperature (DCM_STD_STRUCT *std_struct, T_CurrentTemperature *rtn);

This call requests the temperature so the DPCM can use the temperature for its own calculations. This value, if provided, shall override the default or base temperature.

10.18.5.19 appGetCurrentOpRange

Table 173 provides information on appGetCurrentOpRange.

Table 173—appGetCurrentOpRange

Function name	appGetCurrentOpRange
Arguments	None
Result	Array index of current operating range
Standard Structure fields	None
DCL syntax	EXTERNAL (appGetCurrentOpRange) : result (int:opRangeIndex) ;
C syntax	typedef struct { INTEGER opRangeIndex; } T_CurrentOpRange; int appGetCurrentOpRange (DCM_STD_STRUCT *std_struct, T_CurrentOpRange *rtn);

This call requests the current operating range. The index number (into the array returned by *dpcmGetOpRangeArray*) of the current operating range shall be returned. If no operating ranges are defined for this library, then a value of *–1* shall be returned as the index.

10.18.5.20 dpcmGetTimingStateArray

Table 174 provides information on dpcmGetTimingStateArray.

Table 174—dpcmGetTimingStateArray

Function name	dpcmGetTimingStateArray
Arguments	None
Result	Array of valid states and cells
Standard Structure fields	CellName, pathData (timing-arc-specific), cellData (timing), block
DCL syntax	EXPOSE (dpcmGetTimingStateArray) : result (string[*]: states) ;
C syntax	typedef struct { DCM_STRING_ARRAY *states; } T_timingStateArray; int dpcmGetTimingStateArray (const DCM_STD_STRUCT *std_struct, T_timingStateArray *rtn);

Returns an array of strings that represent the states for the given segment. The syntax and semantics of the state array elements are described in Group condition language (see 8.11), except where modified, as follows.

The state array is an ordered list of states; the application shall scan the list from the first array element to the last and stop at the state index containing the first *TRUE* state.

If both a condition expression string and a double-quoted “state label” are present within a state array element and the application can process the condition expression language, then the application shall use the condition expression string to determine the state of the cell.

The state array elements have the following syntactical requirements:

- a) Each array element shall contain at least one conditionExpression.

- b) Each array element may contain at most two conditionExpressions, separated by a comma and optionally surrounded by whitespace.

The semantics of the state array element *conditionExpressions* are the same as the “Group condition language” semantics (see 8.11.2) except when the following occurs:

- If two conditionExpressions are present, one shall be double-quoted (a state label) and the other shall NOT.
- The state array conditionExpression evaluation is independent of whether the cell is in a steady state or is transitioning into this state.
- If one conditionExpression is present, it shall not be a state label.

Example

The following is an example of a returned array:

```
States[0] = "\"chartreuse\", !B"
States[1] = "A&!B, \"green\""
States[3] = "*"

```

10.18.5.21 appGetCurrentTimingState

Table 175 provides information on appGetCurrentTimingState.

Table 175—appGetCurrentTimingState

Function name	appGetCurrentTimingState
Arguments	None
Result	Index of current state
Standard Structure fields	block, pathData (timing-arc-specific), cellData (timing)
DCL syntax	EXTERNAL (appGetCurrentTimingState) : result (int: stateIndex);
C syntax	typedef struct { INTEGER stateIndex; } T_currentState; int appGetCurrentTimingState (DCM_STD_STRUCT *std_struct, T_currentState *rtn);

Returns an index into the timing state array (returned via *dpcmGetTimingStateArray*).

10.18.6 Functions importing instance name information

This subclause describes importing instance name information functions.

10.18.6.1 dpcmGetCellList

Table 176 provides information on dpcmGetCellList.

Table 176—dpcmGetCellList

Function name	dpcmGetCellList
Arguments	None
Result	Array of cell names, Array of cell name qualifiers, Array of model domains
Standard Structure fields	None
DCL syntax	<pre>EXPOSE(dpcmGetCellList): result(string[*]: cellNameArray, cellQualArray, model_domainArray);</pre>
C syntax	<pre>typedef struct dcm_T_dcmCellList { DCM_STRING_ARRAY *cellNameArray; DCM_STRING_ARRAY *cellQualArray; DCM_STRING_ARRAY *model_domainArray; } T_dcmCellList; int dpcmGetCellList (const DCM_STD_STRUCT *std_struct, T_dcmCellList *rtn);</pre>

This function returns to the application three parallel arrays containing the cell names, cell name qualifiers, and model domains loaded with this DPCM. The cell name, cell name qualifier, and model domain fields at the same array index identify a cell modeled in this DPCM.

Each cell name is a string containing the name of a cell modeled in this DPCM.

Each cell name qualifier is a string whose value is either a cell qualifier string or an asterisk (*). An asterisk returned for the cell name qualifier means that this cell has no cell name qualifier.

Each model domain is a string whose value is either *timing* or *power* or an asterisk (*). The model domain value of *timing* indicates this particular cell supports timing calculations. The model domain value of *power* indicates this particular cell supports power calculations. An asterisk returned for the model domain indicates there are not separate timing and power calculation models.

The application shall call *dpcmGetCellList* to retrieve the list of cells in the DPCM. It is a requirement that any cell name returned by this function shall be found if passed to *modelSearch*.

NOTE—This function gives the DPCM the opportunity to modify the cell list returned by the function *dcmCellList* (which shall not be called directly by the application). This may be required by the DPCM if there are *MODEL* names that model multiple cells and the DPCM wants to return a fully enumerated cell list.

10.18.6.2 appGetCellName

Table 177 provides information on appGetCellName.

Table 177—appGetCellName

Function name	appGetCellName
Arguments	Pin pointer
Result	Cell name, cell qualifier, model domain
Standard Structure fields	None
DCL syntax	EXTERNAL (appGetCellName): passed(pin: cellPin) result(string:cellName, cellQual, modelDomain);
C syntax	typedef struct { STRING cellName, cellQual, modelDomain; } T_CellName; int appGetCellName (DCM_STD_STRUCT *std_struct, T_CellName *rtn, DCM_PIN cellPin);

This returns the cell name to which the passed pin belongs.

10.18.6.3 appGetHierPinName

Table 178 provides information on appGetHierPinName.

Table 178—appGetHierPinName

Function name	appGetHierPinName
Arguments	Pin pointer
Result	Hierarchical pin name
Standard Structure fields	None
DCL syntax	EXTERNAL (appGetHierPinName): passed(pin: pinPointer) result(string:hierPinName);
C syntax	typedef struct { STRING hierPinName; } T_HierPinName; int appGetHierPinName (DCM_STD_STRUCT *std_struct, T_HierPinName *rtn, DCM_PIN pinPointer);

This returns the full hierarchical pin name for the passed pin.

10.18.6.4 appGetHierBlockName

Table 179 provides information on appGetHierBlockName.

Table 179—appGetHierBlockName

Function name	appGetHierBlockName
Arguments	Pin pointer
Result	Hierarchical cell instance name
Standard Structure fields	None
DCL syntax	EXTERNAL (appGetHierBlockName) : passed (pin: pinPointer) result (string: hierBlockName);
C syntax	typedef struct { STRING hierBlockName; } T_HierBlockName; int appGetHierBlockName (DCM_STD_STRUCT *std_struct, T_HierBlockName *rtn, DCM_PIN pinPointer);

This returns the full hierarchical name of the instance to which the passed pin is connected.

10.18.6.5 appGetHierNetName

Table 180 provides information on appGetHierNetName.

Table 180—appGetHierNetName

Function name	appGetHierNetName
Arguments	Pin pointer
Result	Hierarchical interconnect name
Standard Structure fields	None
DCL syntax	EXTERNAL (appGetHierNetName) : passed (pin: pinPointer) result (string: hierNetName);
C syntax	typedef struct { STRING hierNetName; } T_HierNetName; int appGetHierNetName (DCM_STD_STRUCT *std_struct, T_HierNetName *rtn, DCM_PIN pinPointer);

This returns the full hierarchical name of the electrical net to which the passed pin is connected.

10.18.7 Process information functions

This subclause lists the process information functions.

10.18.7.1 dpcmGetThresholds

Table 181 provides information on dpcmGetThresholds.

Table 181—dpcmGetThresholds

Function name	dpcmGetThresholds
Arguments	pin pointer
Result	Voltage transition delay points
Standard Structure fields	calcMode, CellName, block, pathData (timing-pin-specific), cellData (timing)
DCL syntax	<pre>EXPOSE(dpcmGetThresholds): passed(pin: pinPointer) result(double: vol, voh, lowerTransitionThreshold, upperTransitionThreshold, riseSwitchLevel, fallSwitchLevel);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE vol, voh, lowerTransitionThreshold, upperTransitionThreshold, riseSwitchLevel, fallSwitchLevel; } T_thresholds; int dpcmGetThresholds (const DCM_STD_STRUCT *std_struct, T_thresholds *rtn, DCM_PIN pinPointer);</pre>

This function requests voltage, transition, and delay points. This capability can be used to communicate threshold information between voltage islands and between different technologies.

If a zero value is passed for *pinPointer*, technology-wide defaults are returned.

upperTransitionThreshold and *lowerTransitionThreshold* are defined as the points of transition characterization. *RiseSwitchLevel* and *fallSwitchLevel* are defined as the points of delay characterization. *voh* and *vol* are defined as the maximum/minimum voltage swing at which a particular pin was modeled.

10.18.7.2 appGetThresholds

Table 182 provides information on appGetThresholds.

Table 182—appGetThresholds

Function name	appGetThresholds
Arguments	pin pointer
Result	voltage low, voltage high, low transition threshold, high transition threshold, rise switch level, fall switch level
Standard Structure fields	None
DCL syntax	<pre>EXTERNAL(appGetThresholds): passed(pin: pinPointer) result (double: vol, voh, lowerTransitionThreshold, upperTransitionThreshold, riseSwitchLevel, fallSwitchLevel);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE vol, voh, lowerTransitionThreshold, upperTransitionThreshold, riseSwitchLevel, fallSwitchLevel; } T_thresholds; int appGetThresholds (DCM_STD_STRUCT *std_struct, T_thresholds *rtn, DCM_PIN pinPointer);</pre>

This function allows the DPCM to retrieve voltage, transition, and delay points. The application shall call *dpcmGetThresholds* to get this information (see 10.18.7.1).

NOTE—If the pin for which thresholds are being requested is in a different technology, this PI call enables the application to switch to that technology before calling dpcmGetThresholds (and to switch it back when it returns the answer to the requesting DPCM).

10.18.8 Miscellaneous standard interface functions

This subclause shows the miscellaneous standard interface functions.

10.18.8.1 appGetExternalStatus

Table 183 provides information on appGetExternalStatus.

Table 183—appGetExternalStatus

Function name	appGetExternalStatus
Arguments	String containing the name of the EXTERNAL
Result	Integer encoding status
Standard Structure fields	None
DCL syntax	EXTERNAL (appGetExternalStatus) : passed(string: externalName) result(int: externalStatus);
C syntax	typedef struct { INTEGER externalStatus; } T_externalStatus; int appGetExternalStatus (DCM_STD_STRUCT *std_struct, T_externalStatus *rtn, STRING externalName);

appGetExternalStatus is an application-supplied function that returns whether, and to what extent, an application implemented a particular *EXTERNAL*. The value returned for *externalStatus* is as follows:

- 0—if the *EXTERNAL* is not implemented by the application.
- 1—if the *EXTERNAL* is implemented by the application but with code that always returns a return code of severity *ERROR* (a “stub”).
- 2—if the *EXTERNAL* is truly implemented by the application.

10.18.8.2 appGetVersionInfo

Table 184 provides information on appGetVersionInfo.

Table 184—appGetVersionInfo

Function name	appGetVersionInfo
Arguments	None
Result	Version of IEEE Std 1481 with which application is compliant
Standard Structure fields	None
DCL syntax	EXTERNAL (appGetVersionInfo) : result(string: 1481 version);
C syntax	typedef struct { STRING 1481_version; } T_VersionInfo; int appGetVersionInfo (DCM_STD_STRUCT *std_struct, T_VersionInfo *rtn);

This returns the version of IEEE Std 1481 with which the application is compliant. The string for an application compliant with this version of IEEE Std 1481 shall be “IEEE 1481-2019.”

10.18.8.3 appGetResource

Table 185 provides information on appGetResource.

Table 185—appGetResource

Function name	appGetResource
Arguments	String containing the name of the resource desired, string containing the resource's description.
Result	String containing the value of the named resource.
Standard Structure fields	None
DCL syntax	EXTERNAL (appGetResource): passed(string: resourceName, resourceDescription) result(string: resourceValue);
C syntax	typedef struct { STRING resourceValue; } T_Resource; int appGetResource (DCM_STD_STRUCT *std_struct, T_Resource *rtn, STRING resourceName, STRING resourceDescription);

This returns a string value for the passed resource name. The passed resource description may be used within an application message to prompt the user for the value.

10.18.8.4 dpcmGetRuleUnitToSeconds

Table 186 provides information on dpcmGetRuleUnitToSeconds.

Table 186—dpcmGetRuleUnitToSeconds

Function name	dpcmGetRuleUnitToSeconds
Arguments	None
Result	Scale factor power
Standard Structure fields	None
DCL syntax	EXPOSE (dpcmGetRuleUnitToSeconds): result(integer: scaleFactorPower);
C syntax	typedef struct { INTEGER scaleFactorPower; } T_RuleUnitToSecond; int dpcmGetRuleUnitToSeconds (const DCM_STD_STRUCT *std_struct, T_RuleUnitToSecond *rtn);

This returns the basic time units the library assumes, expressed as an integer power of 10. The value 10 *scaleFactorPower*, when multiplied by a time value, changes the time value's units to seconds.

The following example shows how a DPCM indicates the time unit in nanoseconds:

```
EXPOSE calc(dpcmGetRuleUnitToSeconds): result(integer: -9);
```

10.18.8.5 dpcmGetRuleUnitToOhms

Table 187 provides information on dpcmGetRuleUnitToOhms.

Table 187—dpcmGetRuleUnitToOhms

Function name	dpcmGetRuleUnitToOhms
Arguments	None
Result	Scale factor power
Standard Structure fields	None
DCL syntax	EXPOSE (dpcmGetRuleUnitToOhms) : result(integer: scaleFactorPower);
C syntax	typedef struct { INTEGER scaleFactorPower; } T_RuleUnitToOhms; int dpcmGetRuleUnitToOhms (const DCM_STD_STRUCT *std_struct, T_RuleUnitToOhms *rtn);

This returns the basic resistance units the library assumes, expressed as an integer power of 10. The value 10 *scaleFactorPower*, when multiplied by a resistance value, changes the resistance value's units to ohms.

The following example shows how a DPCM indicates the resistance unit in Kohms:

```
EXPOSE calc(dpcmGetRuleUnitToOhms): result(int: 3);
```

10.18.8.6 dpcmGetRuleUnitToFarads

Table 188 provides information on dpcmGetRuleUnitToFarads.

Table 188—dpcmGetRuleUnitToFarads

Function name	dpcmGetRuleUnitToFarads
Arguments	None
Result	Scale factor power
Standard Structure fields	None
DCL syntax	EXPOSE (dpcmGetRuleUnitToFarads) : result(int: scaleFactorPower);
C syntax	typedef struct { INTEGER scaleFactorPower; } T_RuleUnitToFarads; int dpcmGetRuleUnitToFarads (const DCM_STD_STRUCT *std_struct, T_RuleUnitToFarads *rtn);

This returns the basic capacitance units the library assumes, expressed as an integer power of 10. The value 10 *scaleFactorPower*, when multiplied by a capacitance value, changes the capacitance value's units to Farads.

The following example shows how a DPCM indicates the capacitance unit in picoFarads:

```
EXPOSE calc(dpcmGetRuleUnitToFarads): result(integer: -12);
```

10.18.8.7 dpcmGetRuleUnitToHenries

Table 189 provides information on dpcmGetRuleUnitToHenries.

Table 189—dpcmGetRuleUnitToHenries

Function name	dpcmGetRuleUnitToHenries
Arguments	None
Result	Scale factor power
Standard Structure fields	None
DCL syntax	EXPOSE (dpcmGetRuleUnitToHenries): result(integer: scaleFactorPower);
C syntax	typedef struct { INTEGER scaleFactorPower; } T_RuleUnitToHenries; int dpcmGetRuleUnitToHenries (const DCM_STD_STRUCT *std_struct, T_RuleUnitToHenries *rtn);

This returns the basic inductance units the library assumes, expressed as an integer power of 10. The value 10 *scaleFactorPower*, when multiplied by an inductance value, changes the inductance value's units to Henries.

The following example demonstrates how a DPCM indicates the inductance unit in microHenries:

```
EXPOSE calc(dpcmGetRuleUnitToHenries): result(integer: -6);
```

10.18.8.8 dpcmGetRuleUnitToWatts

Table 190 provides information on dpcmGetRuleUnitToWatts.

Table 190—dpcmGetRuleUnitToWatts

Function name	dpcmGetRuleUnitToWatts
Arguments	None
Result	Scale factor power
Standard Structure fields	None
DCL syntax	EXPOSE (dpcmGetRuleUnitToWatts): result(int: scaleFactorPower);
C syntax	typedef struct { INTEGER scaleFactorPower; } T_RuleUnitToWatts; int dpcmGetRuleUnitToWatts (const DCM_STD_STRUCT *std_struct, T_RuleUnitToWatts *rtn);

This returns the basic power units the library assumes, expressed as an integer power of 10. The value 10 *scaleFactorPower*, when multiplied by a power value, changes the power value's units to Watts.

The following example demonstrates how a DPCM indicates the power unit in microWatts:

```
EXPOSE calc(dpcmGetRuleUnitToWatts): result(integer: -6);
```

10.18.8.9 dpcmGetRuleUnitToJoules

Table 191 provides information on dpcmGetRuleUnitToJoules.

Table 191—dpcmGetRuleUnitToJoules

Function name	dpcmGetRuleUnitToJoules
Arguments	None
Result	Scale factor power
Standard Structure fields	None
DCL syntax	EXPOSE(dpcmGetRuleUnitToJoules): result(integer: scaleFactorPower);
C syntax	<pre>typedef struct { INTEGER scaleFactorPower; } T_RuleUnitToJoules; int dpcmGetRuleUnitToJoules (const DCM_STD_STRUCT *std_struct, T_RuleUnitToJoules *rtn);</pre>

This returns the basic energy units the library assumes, expressed as an integer power of 10. The value 10 *scaleFactorPower*, when multiplied by an energy value, changes the energy value's units to Joules.

Example

The following example demonstrates how a DPCM indicates the energy unit in picoJoules:

```
EXPOSE calc(dpcmGetRuleUnitToHenries): result(integer: -12);
```

10.18.8.10 dpcmGetTimeResolution

Table 192 provides information on dpcmGetTimeResolution.

Table 192—dpcmGetTimeResolution

Function name	dpcmGetTimeResolution
Arguments	None
Result	Time resolution power of 10
Standard Structure fields	None
DCL syntax	EXPOSE(dpcmGetTimeResolution): result(int: timeResolutionPower);
C syntax	<pre>typedef struct { INTEGER timeResolutionPower; } T_TimeResolution; int dpcmGetTimeResolution (const DCM_STD_STRUCT *std_struct, T_TimeResolution *rtn);</pre>

This returns the coarsest resolution for time values to be used by the application to help ensure accurate interaction with the technology to which the Standard Structure is set. The result is expressed as an integer representing a power of 10. The value 10 *timeResolutionPower* represents this time value in seconds.

10.18.8.11 dpcmGetParasiticCoordinateTypes

Table 193 provides information on dpcmGetParasiticCoordinateTypes.

Table 193—dpcmGetParasiticCoordinateTypes

Function name	dpcmGetParasiticCoordinateTypes
Arguments	None
Result	Distance types, coordinate conversion
Standard Structure fields	None
DCL syntax	EXPOSE(dpcmGetParasiticCoordinateTypes): result(short: xCoordinateType, yCoordinateType; double: xConversion, yConversion);
C syntax	typedef struct { SHORT xCoordinateType, yCoordinateType; DCM_DOUBLE xStepSize, yStepSize; } T_ParasiticCoordinateTypes; int dpcmGetParasiticCoordinateTypes (const DCM_STD_STRUCT *std_struct, T_ParasiticCoordinateTypes *rtn);

This returns coordinate types, step size conversion scaling factors. The coordinate types shall have a value of zero (0) indicating the coordinate value is a distance or a value of one (1) indicating the coordinate value is in stepping units. The step size is the scaling factor used to convert from stepping units to rule distance units. xConversion and yConversion contain the scaling factor to use in the conversion of stepping units to rule distance units.

Example

float: distanceInRuleUnitsForTheXaxis = numberOfUnits*xConversion;

10.18.8.12 dpcmIsSlewTime

Table 194 provides information on dpcmIsSlewTime.

Table 194—dpcmIsSlewTime

Function name	dpcmIsSlewTime
Arguments	None
Result	Indicator
Standard Structure fields	None
DCL syntax	EXPOSE(dpcmIsSlewTime): result(integer:slewTime);
C syntax	typedef struct { INTEGER slewTime; } T_IsSlewTime; int dpcmIsSlewTime (const DCM_STD_STRUCT *std_struct, T_IsSlewTime *rtn);

This returns the units for calculated slews as absolute time or rate of change. If *slewTime* is nonzero, the slew values are in time units. If the *slewTime* is zero, the slew values are rate of change (time/volts) units.

10.18.8.13 dpcmDebug

Table 195 provides information on dpcmDebug.

Table 195—dpcmDebug

Function name	dpcmDebug
Arguments	Debug level
Result	Previous level
Standard Structure fields	None
DCL syntax	<pre>EXPOSE(dpcmDebug): passed(integer: dpcmDebugLevel) result(integer: prevLevel);</pre>
C syntax	<pre>typedef struct { INTEGER prevLevel; } T_Debug; int dpcmDebug (const DCM_STD_STRUCT *std_struct, T_Debug *rtn, INTEGER dpcmDebugLevel);</pre>

This specifies the debugging level to be used within the DPCM for the current technology family (*TECH_FAMILY*). A zero value for debug level disables the debug tracing. Values greater than zero enable debug trace to a varying degree of detail, where a higher value requests a greater amount of detail. The return value is the debug level current at the time this function call was made.

NOTE—This function allows the library developer to force the DPCM’s execution to produce diagnostic data in order to troubleshoot a problem. The number of debug levels supported by a library is determined by the library developer, as is the association of a debug level value to the diagnostic results produced.

10.18.8.14 dpcmGetVersionInfo

Table 196 provides information on dpcmGetVersionInfo.

Table 196—dpcmGetVersionInfo

Function name	dpcmGetVersionInfo
Arguments	None
Result	Library identifier, version of IEEE Std 1481 with which application is compliant
Standard Structure fields	None
DCL syntax	<pre>EXPOSE(dpcmGetVersionInfo): result(string: libIdentification, 1481_version);</pre>
C syntax	<pre>typedef struct { STRING libIdentification, 1481_version; } T_intVer; int dpcmGetVersionInfo (const DCM_STD_STRUCT *std_struct, T_intVer *rtn);</pre>

This returns strings that identify the technology library and the version of IEEE Std 1481 with which the library is compliant. The library identification is an arbitrary string. The *P1481_version* result variable shall be set to “IEEE 1481-2019.”

10.18.8.15 dpcmHoldControl

Table 197 provides information on dpcmHoldControl.

Table 197—dpcmHoldControl

Function name	dpcmHoldControl
Arguments	None
Result	Pointer to application's node structure
Standard Structure fields	CellName, fromPoint, toPoint, sourceEdge, sinkEdge
DCL syntax	EXPOSE(dpcmHoldControl): result(integer: doHoldControlSnip);
C syntax	typedef struct { INTEGER doHoldControlSnip; } T_HoldControl; int dpcmHoldControl (const DCM_STD_STRUCT *std_struct, T_HoldControl *rtn);

This function allows the application to query whether hold control should be used. The *fromPoint* is the pin from which the signal is launched. The *toPoint* is where the signal returns back to the latch in a feedback loop.

dpcmHoldControl indicates a signal shall have been present at the launch latch's input for the hold control not to have been violated. Zero (0) signifies to not perform hold snip, and one (1) signifies to perform hold snip.

For a detailed exposition of the issues surrounding this PI call, see Annex A.

10.18.8.16 dpcmFillPinCache

Table 198 provides information on dpcmFillPinCache.

Table 198—dpcmFillPinCache

Function name	dpcmFillPinCache
Arguments	Pin pointer, Resistance load, Capacitance load, Slew In, Slew Out, Memory handle in
Result	Memory handle out
Standard Structure fields	CellName, block, cellData (timing or power), pathData (timing or power pin-specific)
DCL syntax	EXPOSE dpcmFillPinCache): passed(pin: pinPointer; double: resistanceLoad; double[*]: capLoad, slewIn, slewOut; void: memoryHandleIn) result(void: memoryHandleOut);
C syntax	typedef struct { VOID memoryHandleOut; } T_FillPinCache; int dpcmFillPinCache (const DCM_STD_STRUCT *std_struct, T_FillPinCache *rtn, DCM_PIN pinPointer, DCM_DOUBLE resistanceLoad, DCM_DOUBLE_ARRAY *capLoad, DCM_DOUBLE_ARRAY *slewIn, DCM_DOUBLE_ARRAY *slewOut, VOID memoryHandleIn);

This function is called by the application to supply the load and slew of the specified pin to the DPCM in response to the DPCM request for this information through the *appRegisterCellInfo* function. This function shall supply the *capLoad*, *resistanceLoad*, and *slew* for all pins of the cell specified in the Standard Structure whose types (inputs, bidirectionals, and outputs) match the types requested on the *appRegisterCellInfo* call.

The *capLoad* and *slewIn* arrays are indexed by the *SINK_EDGE_SCALAR* enumeration. The application shall supply the capacitance and input slew values for each *SINK_EDGE* enumeration. The *slewOut* array is indexed by the *SOURCE_EDGE_SCALAR* enumeration. The application shall supply the output slew value for each *SOURCE_EDGE* enumeration. For requested data that are not known, the application shall supply a value of zero.

The memory handle parameter passed into this function shall be the memory handle most recently passed from the DPCM to the application for this timing or power calculation request.

10.18.8.17 dpcmFreePinCache

Table 199 provides information on dpcmFreePinCache.

Table 199—dpcmFreePinCache

Function name	dpcmFreePinCache
Arguments	Memory handle
Result	Return code
Standard Structure fields	CellName, block, cellData (timing or power)
DCL syntax	<pre>EXPOSE(dpcmFreePinCache): passed(void: memoryHandleIn) result(integer: rc);</pre>
C syntax	<pre>typedef struct { INTEGER rc; } T_FreePinCache; int dpcmFreePinCache (const_DCM_STD_STRUCT *std_struct, T_FreePinCache *rtn, VOID memoryHandleIn);</pre>

This function is called to free the load and slew cache when no longer needed.

10.18.8.18 appRegisterCellInfo

Table 200 provides information on appRegisterCellInfo.

Table 200—appRegisterCellInfo

Function name	appRegisterCellInfo
Arguments	Integers indicating whether capacitance load, resistance load or slew is needed
Result	Memory Handle
Standard Structure fields	block, cellName
DCL syntax	<pre>EXTERNAL(appRegisterCellInfo): passed(integer: FillCapLoad, FillResLoad, FillSlew) result(void: memoryHandleOut);</pre>
C syntax	<pre>typedef struct { VOID memoryHandleOut; } T_memoryHandle; int appRegisterCellInfo (DCM_STD_STRUCT *std_struct, T_memoryHandle *rtn, INTEGER FillCapLoad, INTEGER FillResLoad, INTEGER FillSlew);</pre>

This function may be called by the DPCM when the DPCM is called to calculate power or timing. This call enables the application to supply load and slew information to the DPCM. The application supplies this information by calling the *dpcmFillPinCache* function for all pins of the cell specified in the Standard Structure whose types (inputs, bidirectionals, and outputs) match the types registered on the call to this function.

On delay, slew, check, or power calculations, the DPCM may call back with *appRegisterCellInfo*. The application has a choice to create a new cache if one was never created before, or pass in the memory handle of a previously filled in cache (for the same cell type or instance). If the application has a memory handle with load and slew information for this instance with the requested pins filled in (via previous calls to *dpcmFillPinCache*), then the application need not refill this cache. It may pass this memory handle back to the DPCM in *memoryHandleOut*. If the application has not previously filled a cache, it shall pass in 0 (zero for *memoryHandleIn*) to the first call to a *dpcmFillPinCache*. If on subsequent calls to *dpcmFillPinCache* a different memory handle is returned, the new memory handle shall be passed to either the next call to *dpcmFillPinCache* or returned to the DPCM when returning from *appRegisterCellInfo*.

The DPCM shall pass values for the *FillCapLoad*, *FillResLoad* and *FillSlew* parameters to inform the application of which pins require the requested information (see 8.3).

The following values apply to each of the three flags:

- 0—Indicates this information is not needed for any pins.
- 1—Indicates this information is needed for input and bidirectional pins.
- 2—Indicates this information is needed for output and bidirectional pins.
- 3—Indicates this information is needed for all pins.

10.18.9 Power-related functions

This subclause shows the power-related functions.

10.18.9.1 dpcmGetCellPowerInfo

Table 201 provides information on *dpcmGetCellPowerInfo*.

Table 201—*dpcmGetCellPowerInfo*

Function name	<i>dpcmGetCellPowerInfo</i>
Arguments	None
Result	Group pin list, Group condition list, Sensitivity list, Initial state choices, Supported methods
Standard Structure fields	CellName, cellData (power)
DCL syntax	<pre>EXPOSE(dpcmGetCellPowerInfo): result(string[*]: groupPinList, groupConditionList, sensitivityList, initialStateChoices; integer: aet_supported, group_supported, pin_supported);</pre>
C syntax	<pre>typedef struct { DCM_STRING_ARRAY group_pin_list, *group_condition_list, sensitivity_list, *initial_state_choices; INTEGER aet_supported, group_supported, pin_supported; } T_lists; int dpcmGetCellPowerInfo (const DCM_STD_STRUCT *std_struct, T_lists *rtn);</pre>

Returns the power calculation methods supported by the DPCM for the cell specified in the Standard Structure.

The “*supported*” flags relate to the following *EXPOSE* functions for calculating power:

- a) *aet_supported* : *dpcmGetAETCellPowerWithSensitivity*
- b) *group_supported* : *dpcmGetPowerWithState*
- c) *pin_supported* : *dpcmGetPinPower*

For the return parameters *aet_supported*, *group_supported*, and *pin_supported*, a value of one (1) indicates this method for power computation is supported and a value of zero (0) indicates this method is not supported for this cell.

This function also returns the following arrays of information in support of these power calculation methods:

- The *group_pin_list* and group condition list for *dpcmGetPowerWithState*.
- The *sensitivity_list* for *dpcmAetCellPowerWithSensitivity*.
- The *initial_state_choices* for all three power calculation methods.

A 0 length array is returned for each resultant array (the *group_pin_list*, *group_condition_list*, *sensitivity_list*, and *initial_state_choices*) when this information is not needed or not available.

10.18.9.2 dpcmGetCellPowerWithState

Table 202 provides information on *dpcmGetCellPowerWithState*.

Table 202—dpcmGetCellPowerWithState

Function name	<i>dpcmGetCellPowerWithState</i>
Arguments	Group index, Condition index
Result	Energy/rail(static), Static power/rail(static), Total energy, Total static power
Standard Structure fields	CellName, block, cellData (power), calcMode
DCL syntax	<pre>EXPOSE(dpcmGetCellPowerWithState): passed(integer: groupIndex, conditionIndex) result(double[*]: energyPerRail, staticPowerPerRail; double: totalEnergy, totalStaticPower);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE_ARRAY *energyPerRail, *staticPowerPerRail; DCM_DOUBLE totalEnergy; DCM_DOUBLE totalStaticPower; } T_energy; int dpcmGetCellPowerWithState (const DCM_STD_STRUCT *std_struct, T_energy *rtn, INTEGER groupIndex, INTEGER conditionIndex);</pre>

This returns static power per rail, dynamic energy per rail, total energy, and total static power given a specific group and condition index. The application uses the group pin lists and group condition lists returned by *dpcmGetCellPowerInfo* to determine the group and condition index based on a pin change event.

10.18.9.3 dpcmGetAETCellPowerWithSensitivity

Table 203 provides information on *dpcmGetAETCellPowerWithSensitivity*.

Table 203—dpcmGetAETCellPowerWithSensitivity

Function name	dpcmGetAETCellPowerWithSensitivity
Arguments	Sensitivity mask
Result	Energy/rail(static), Static power/rail(static), Total energy, Total static power
Standard Structure fields	CellName, block, cellData, calcMode (power)
DCL syntax	<pre>EXPOSE (dpcmGetAETCellPowerWithSensitivity): passed(integer[*]: sensitivityMask) result(double[*]: energyPerRail, staticPowerPerRail; double:totalEnergy, totalStaticPower);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE_ARRAY *energyPerRail, *staticPowerPerRail; DCM_DOUBLE totalEnergy; DCM_DOUBLE totalStaticPower; } T_energy; int dpcmGetAETCellPowerWithSensitivity (const DCM_STD_STRUCT *std_struct, T_energy *rtn, DCM_DOUBLE_ARRAY *sensitivityMask);</pre>

This returns static power per rail, dynamic energy per rail, total energy, and total static power given a specified sensitivity mask array. The mask value is determined based on the state of the pins for the cell in the *Standard Structure*. The elements of the sensitivity mask correspond to the elements of the sensitivity list returned by the call to *dpcmGetCellPowerInfo*.

The mask definitions are as follows:

- When the element of the sensitivity list array (returned by *dpcmGetCellPowerInfo*) contains a single pin, each element of the mask array encodes the “from” and “to” states in the two least-significant bytes of the integer, as shown in Table 204.

Table 204—Integer LSB example

MSB	LSB
	From To

- For both from and to, the encoding is shown in Table 205.

Table 205—Mask encoding

Value	Pin State
0	0
1	1
2	Z
3	X

- When the element of the sensitivity list array (returned by *dpcmGetCellPowerInfo*) contains multiple pins, each element of the mask array encodes the state of these pins in the least-significant byte of the integer, using the following encoding:

0—None of the pins changed state

1—At least one of the pins changed state

2—At least one of the pins went to X

- If a pin in the sensitivity list goes to X, this condition takes precedence over other changes and the mask values shall be set to 2.

10.18.9.4 dpcmGetPinPower

Table 206 provides information on dpcmGetPinPower.

Table 206—dpcmGetPinPower

Function name	dpcmGetPinPower
Arguments	Pin pointer, Ask for registration
Result	Energy/rail(static), Static power/rail(static), totalEnergy, totalStaticPower
Standard Structure fields	CellName, block, cellData (power), pathData (power-pin-specific), calcMode
DCL syntax	<pre>EXPOSE(dpcmGetPinPower): passed(pin: pinPointer; integer: ask_for_registration) result(double[*]: energyPerRail, staticPowerPerRail; double: totalEnergy, totalStaticPower);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE_ARRAY *energyPerRail, *staticPowerPerRail; DCM_DOUBLE totalEnergy; DCM_DOUBLE totalStaticPower; } T_energy; int dpcmGetPinPower (const DCM_STD_STRUCT *std_struct, T_energy *rtn, DCM_PIN pinPointer, INTEGER ask_for_registration);</pre>

This returns static power per rail, dynamic energy per rail, total energy, and total static power for a specific pin state change. If *ask_for_registration* is *FALSE* (set to 0), the DPCM shall not call back to the application (using *appRegisterCellInfo*) for pin load slew, and resistance values of the cell are specified in the Standard Structure.

10.18.9.5 dpcmAETGetSettlingTime

Table 207 provides information on dpcmAETGetSettlingTime.

Table 207—dpcmAETGetSettlingTime

Function name	dpcmAETGetSettlingTime
Arguments	None
Result	Array of pins, Array of settling times
Standard Structure fields	CellName, block, cellData (power), calcMode
DCL syntax	<pre>EXPOSE(dpcmAETGetSettlingTime): result(string[*]: pinList; double[*]: settlingTimes);</pre>
C syntax	<pre>typedef struct { DCM_STRING_ARRAY *pinList; DCM_DOUBLE_ARRAY *settlingTimes; } T_times; int dpcmAETGetSettlingTime (const DCM_STD_STRUCT *std_struct, T_times *rtn);</pre>

This returns two parallel arrays. The first array contains strings, each of which is a pin list whose syntax is defined in 8.8. The second array contains the settling time values, where each value is the settling time for each pin in the associated pin list.

This function shall only be called by the application for AET-related power computation.

See the definition of settling time as well as the method for calculating power in 8.5 and 8.6. Each pin in the pin list array shall be an actual pin name on the cell specified in the Standard Structure.

10.18.9.6 dpcmAETGetSimultaneousSwitchTime

Table 208 provides information on dpcmAETGetSimultaneousSwitchTime.

Table 208—dpcmAETGetSimultaneousSwitchTime

Function name	dpcmAETGetSimultaneousSwitchTime
Arguments	None
Result	Array of pins, Array of simultaneous switch times
Standard Structure fields	CellName, block, cellData (power), calcMode
DCL syntax	EXPOSE(dpcmAETGetSimultaneousSwitchTime): result(string[*]: pinList; double[*]: SimultaneousSwitchTimes);
C syntax	typedef struct { DCM_STRING_ARRAY *pinList; DCM_DOUBLE_ARRAY *SimultaneousSwitchTimes; } T_times; int dpcmAETGetSimultaneousSwitchTime (const DCM_STD_STRUCT *std_struct, T_times *rtn);

This returns two parallel arrays. The first array contains strings, each of which is a pin list whose syntax is defined in 8.8. The second array contains the simultaneous switch time values, where each value is the simultaneous switching time for each pin in the associated pin list.

This function shall only be called by the application for AET-related power computation.

See the definition of simultaneous switching time as well as the method for calculating power in 8.4 and 8.6. Each pin in the pin list array shall be an actual pin name on the cell specified in the Standard Structure.

10.18.9.7 dpcmGroupGetSettlingTime

Table 209 provides information on dpcmGroupGetSettlingTime.

Table 209—dpcmGroupGetSettlingTime

Function name	dpcmGroupGetSettlingTime
Arguments	None
Result	Array of pins, Array of settling times
Standard Structure fields	CellName, block, cellData (power), calcMode
DCL syntax	EXPOSE(dpcmGroupGetSettlingTime): result(string[*]: pinList; double[*]: settlingTimes);
C syntax	typedef struct { DCM_STRING_ARRAY *pinList; DCM_DOUBLE_ARRAY *settlingTimes; } T_times; int dpcmGroupGetSettlingTime (const DCM_STD_STRUCT *std_struct, T_times *rtn);

This returns two parallel arrays. The first array contains strings, each of which is a pin list whose syntax is defined in 8.8. The second array contains the corresponding settling time values, where each value is the settling time between the pins in the associated pin list.

This function shall only be called by the application for group-related power computation.

See the definition of settling time as well as the method for calculating power in 8.5 and 8.6. Each pin in the pin list array shall be an actual pin name on the cell specified in the Standard Structure.

10.18.9.8 dpcmGroupGetSimultaneousSwitchTime

Table 210 provides information on dpcmGroupGetSimultaneousSwitchTime.

Table 210—dpcmGroupGetSimultaneousSwitchTime

Function name	dpcmGroupGetSimultaneousSwitchTime
Arguments	None
Result	Array of pins, Array of simultaneous switch times
Standard Structure fields	CellName, block, cellData (power), calcMode
DCL syntax	<pre>EXPOSE (dpcmGroupGetSimultaneousSwitchTime): result(string[*]: pinList; double[*]: SimultaneousSwitchTimes);</pre>
C syntax	<pre>typedef struct { DCM_STRING_ARRAY *pinList; DCM_DOUBLE_ARRAY *SimultaneousSwitchTimes; } T_times; int dpcmGroupGetSimultaneousSwitchTime (const DCM_STD_STRUCT *std_struct, T_times *rtn);</pre>

This returns two parallel arrays. The first array contains strings, each of which is a pin list whose syntax is defined in 8.8. The second array contains the simultaneous switch time values, where each value is the simultaneous switching time for each pin in the associated pin list.

This function shall only be called by the application for group-related power computation.

See the definition of simultaneous switching time as well as the method for calculating power in 8.5 and 8.6. Each pin in the pin list array shall be an actual pin name on the cell specified in the Standard Structure.

10.18.9.9 dpcmCalcPartialSwingEnergy

Table 211 provides information on dpcmCalcPartialSwingEnergy.

Table 211—dpcmCalcPartialSwingEnergy

Function name	dpcmCalcPartialSwingEnergy
Arguments	Pin pointer, Pin group index, Pin condition index, Width of occurrence
Result	Energy per rail, Total energy
Standard Structure fields	CellName, block, cellData (power), calcMode
DCL syntax	<pre>EXPOSE dpcmCalcPartialSwingEnergy): passed(pin: pinPointer; integer: group_index, condition_index; double: width) result(double[*]: energyPerRail; double: totalEnergy);</pre>

C syntax	<pre>typedef struct { DCM_DOUBLE_ARRAY *energyPerRail; DCM_DOUBLE totalEnergy; } T_CalcPartialSwingEnergy; int dpcmCalcPartialSwingEnergy (const DCM_STD_STRUCT *std_struct, T_CalcPartialSwingEnergy *rtn, DCM_PIN pinPointer, INTEGER Group_index, INTEGER Condition_index, DCM_DOUBLE width);</pre>

This returns the energy of a partial logic swing for a particular pin group. Width is defined as the time for the pin to transition from a threshold and back to the same threshold.

If the application is using *dpcmGetAETCellPowerWithSensitivity*, pass in a *–1* for the *group_index* and *Condition_index*.

NOTE—No static power is returned by this call; the application shall use the static power associated with the proceeding cell state.

10.18.9.10 dpcmSetInitialState

Table 212 provides information on *dpcmSetInitialState*.

Table 212—dpcmSetInitialState

Function name	<i>dpcmSetInitialState</i>
Arguments	Initial state index, cache handle
Result	Static power/rail, Total static power, cacheHandle
Standard Structure fields	CellName, block, cellData (power)
DCL syntax	<pre>EXPOSE(dpcmSetInitialState): passed(integer: initialStateIndex; void: cacheHandleIn) result(double[*] staticPowerPerRail; double: totalStaticPower; void: cacheHandleOut);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE_ARRAY *staticPowerPerRail; double totalStaticPower; VOID cacheHandleOut; } T_energy; int dpcmSetInitialState (const DCM_STD_STRUCT *std_struct, T_energy *rtn, INTEGER initialStateIndex, VOID cacheHandleIn);</pre>

This function is used by the application to set the initial state of the instance specified in the Standard Structure. The *initialStateIndex* is the index number of the desired initial state from the *initialStateChoices* array returned by the *dpcmGetCellPowerInfo* function for this cell.

The *initialStateIndex* shall be a valid index into the *initialStateChoices* array.

The static power per rail and the total static power consumed by the specified initial state is returned to the application. This function may call back for filling of load and slew caches if needed by using *appRegisterCellInfo*.

For cells with initial states, the DPCM creates a state cache and returns a handle to this cache back to the application. This state cache along with the use of the *cacheHandleIn* and *cacheHandleOut* are described in 8.3. The application shall associate the returned state cache handle with the instance specified in the

Standard Structure. During a power calculation request for an instance that has initial states, the DPCM shall call *appGetStateCache* to retrieve this state cache handle.

10.18.9.11 dpcmFreeStateCache

Table 213 provides information on dpcmFreeStateCache.

Table 213—dpcmFreeStateCache

Function name	dpcmFreeStateCache
Arguments	cache handle
Result	Return code
Standard Structure fields	CellName, block, cellData
DCL syntax	EXPOSE(dpcmFreeStateCache): passed(void: cacheHandleIn) result(integer: rc);
C syntax	typedef struct { INTEGER rc; } T_FreeStateCache; int dpcmFreeStateCache (const DCM_STD_STRUCT *std_struct, T_FreeStateCache *rtn, VOID cacheHandleIn);

This function is called to free the state cache when no longer needed (see 8.2).

10.18.9.12 appGetStateCache

Table 214 provides information on appGetStateCache.

Table 214—appGetStateCache

Function name	appGetStateCache
Arguments	None
Result	cache Handle out
Standard Structure fields	block, cellName
DCL syntax	EXTERNAL(appGetStateCache): result(void: cacheHandleOut);
C syntax	typedef struct { VOID cacheHandleOut; } T_cacheHandle; int appGetStateCache (DCM_STD_STRUCT *std_struct, T_cacheHandle *rtn);

During a power calculation request on an instance with initial state choices, the DPCM shall call this function to retrieve the instance's state cache handle (see 8.2). It is the application's responsibility to request this cache be created and initialized (via *dpcmSetInitialState*), and to associate the returned state cache handle with the instance specified in the Standard Structure.

10.18.9.13 dpcmGetNetEnergy

Table 215 provides information on dpcmGetNetEnergy.

Table 215—dpcmGetNetEnergy

Function name	dpcmGetNetEnergy
Arguments	None
Result	Net energy
Standard Structure fields	CellName, block, fromPoint, cellData (power), pathData (power-pin-specific), calcMode
DCL syntax	EXPOSE (dpcmGetNetEnergy) : result(double: netEnergy);
C syntax	typedef struct { DCM_DOUBLE netEnergy; } T_NetEnergy; int dpcmGetNetEnergy (const DCM_STD_STRUCT *std_struct, T_NetEnergy *rtn);

This returns the energy consumed by a transition on the net connected to *fromPoint*.

10.19 Application context

Once an application starts an exchange with a library by calling one of the latter's functions, and the library in turn calls an application function during this process, the application might need context information associated with the library call it made originally to respond correctly to the library's request. To do this effectively, the application can attach context data to the *Standard Structure* passed to the library as part of the original function call. The application can set the *applicationInfo* pointer in the *Standard Structure* to point to this context data.

The library shall not modify the *applicationInfo* pointer in any *Standard Structure* it receives from the application nor the application data to which this field points. In addition, the library shall help to ensure this same pointer value is present in all *Standard Structures* passed in calls to application functions until the original function call the application made is complete.

10.19.1 pathData association

The library shall associate a maximum of one *pathData* pointer with each arc, pin, or internal node in a cell model. The *pathData* pointer associated with an arc is used to provide access to information specific to that arc cached by the library. Similarly, the *pathData* pointer associated with a pin or node is used to provide access to information specific to that pin or node or any arcs that radiate from the pin or node.

The application shall be responsible for keeping the input and output parts of a bidirectional pin separate and distinguishable, so they can be viewed as separate pins by the library. The library shall be able to associate a unique *pathData* pointer with each of these parts and the application shall store this pointer with the appropriate part when requested to do so by the library.

Any attempt by the library to associate (via any API function call) multiple, different *pathData* pointers with a given pin, node, or arc shall be an error.

10.20 Application and library interaction

Libraries may contain five model domains, *timing*, *power*, *behavior*, *vectorTiming* and *vectorPower*. An application interacting with a library shall maintain separate cell models for each domain it chooses to represent within the library.

Example

```

/** 2 input AND gate behavior description */
model (AND2_behavior) : defines (AND2.*.behavior);

modelproc (AND2_behavior) :
  do: function(
    Z = A `& B
  );
end;

/** 2 input AND gate timing description */
model (AND2_timing) : defines (AND2.*.timing);

modelproc (AND2_timing) :
  path(): from(A,B) to(Z) propagate(rise<->rise & fall<->fall); end;

```

The application shall only request the library elaborate a cell model for a single domain at a time. The application shall not use the default operator *** in the domain field of the *Standard Structure* when calling for model elaboration.

The application shall determine the types of cells and model domains supported in the library by calling the interface function *dpcmGetCellList*. For each cell modeled in multiple domains, the DPCM shall return as many entries in the cell list for that cell as the number of domains where the cell is modeled. Once the model domains have been determined, the application can then call *modelSearch* for a given *cell*, a *cellQual*, a list of inputs, a list of outputs, and a specific *modelDomain*, which is supported by the library.

Model elaboration begins when the application calls *modelSearch*. The library can call an appropriate sequence of callback functions (during model elaboration), including the implicit functions *newPropagateSegment()*, *newDelayMatrixRow()*, and so on. The sequence of these callbacks and the data associated with them are unique to the cell being elaborated. The application shall store the *pathData* pointers supplied by the implicit callbacks. The library shall not construct a model for a *cell*, *cellQual*, and *modelDomain* combination that requires the application to store more than one *pathData* pointer for any single pin, arc, or internal node.

The model elaboration process for the *behavior*, *vectorTiming*, and *vectorPower* domains shall describe the cell's function or state in a graph format. The graph format transferred to the application uses nodes to represent Boolean operations, arcs to represent data sources, and arcs to represent data results of these Boolean operations. Once the library has transferred the graph to the application, it is the responsibility of the application to evaluate the graph for its own purposes.

The application can choose to defer the evaluation or directly use the representation transferred by the library. In either case, the application shall save the *pathData* pointers returned by the implicit callbacks and the *cellData* set in the *Standard Structure* when the call to *modelSearch* ends. When the application no longer needs the *pathData* or *cellData* pointers, it received from the library, and the application shall call *dcmDeletePathDataBlock* or *dcmDeleteCellDataBlock*, respectively.

10.20.1 behavior model domain

The *behavior* domain uses a graph to represent the Boolean functions of a cell. Each graph transferred during model elaboration of a cell represents an independent function performed by that cell.

10.20.2 vectorTiming and vectorPower model domains

The model elaboration process shall describe a cell's set of vector expressions in a graph format. Each vector that evaluates to *true* represents a cell state change. Each cell state change has associated with it a

delay, slew, and check calculation. Once elaboration is complete, the application can evaluate each graph that evaluates to true and call the library for delay, slew, and check. The application requesting (for the library to calculate) delay, slew, or check shall provide the *pathData* pointer associated with the *DCM_PRIMITIVE_VECTOR_DELAY_TARGET* or *DCM_PRIMITIVE_VECTOR_CHECK_TARGET* to that library. Once the application requests vector delay and vector slew evaluations, the library shall return the early and late delay and slew values.

Unlike in IEEE Std 1481™-1999, vectors permit description of multiple transitions. For example, the following describes a sequence of transitions for a given from and to point:

Example

```
/** 2 input AND gate vector power description */
model (AND2_vectorTiming) : defines (AND2.*.vectorTiming);

modelproc (AND2_vectorTiming) :
  do: vector(-|+A->-|+B->-|+Z)
    from(B) to(Z) propagate(rise->rise);
  do: vector(-|+B->-|+A->-|+Z)
    from(A) to(Z) propagate(rise->rise);
end;
```

For the timing and power domains, the delay and output slew calculation may be functions of input slew at pin *a*. However, as the previous example indicates, delay and output slew calculations can be functions of more than two pins. Therefore, a library that contains *vectorTiming* or *vectorPower* models can call the application for input slews, output capacitance loads, and resistance on multiple pins via the *appRegisterCellInfo()*. In response to this call, the application shall provide slews and loads for the specified pins by calling *dpcmFillPinCache()* for those pins listed in the vector expression. The library then uses those slews and loads for computing the delay, slew, check, or power.

10.20.3 Power unit conversion

The library can represent its power modeling in either energy (Joules), power (Watts), or any other proprietary unit. The library needs to help ensure the appropriate energy units are passed for all the functions in the *power* and *vectorPower* domains. These units shall match the exponents returned by *dpcmGetRuleUnitToJoules* and *dpcmGetRuleUnitToWatts*.

10.20.4 Vector power calculation

The model elaboration process shall describe a cell's set of vector expressions in a graph format. Each vector represents either a cell state change or a steady state of a cell. Each vector has a power calculation associated with it. Once the model elaboration process for a cell is complete, the application has a set of vectors and their associated *pathData* pointers. The application can evaluate the graph for each vector to determine which occur during simulation. For those that evaluate to *true*, the application shall call *dpcmGetCellVectorPower* (see 10.23.13.5) for the power associated with each vector. The application shall provide the *pathData* pointer associated with the *DCM_PRIMITIVE_VECTOR_POWER_TARGET* when calling the library to calculate the power.

Once the application has set up the models of power vectors and determined that the conditions described by the vector function graph are met, it can call the DPCM to calculate power for a given vector. The application shall call *dpcmGetCellVectorPower*. The application needs to pass the *cellData* information for the cell and the *pathData* for the vector.

To obtain slewrates and capacitance data for the instance, the DPCM shall call back to the application via the *EXTERNAL* API *appRegisterCellInfo()*. This function passes in three flags indicating the type of information being requested (capacitance, resistance, and/or slew) and the types of pins for which the

requested information is needed (i.e., inputs, outputs, bidirectionals, or all). This callback (*appRegisterCellInfo()*) enables the application to update (if necessary) the load and slew cache for this instance prior to the power calculation via the *EXPOSEAPI dpcmFillPinCache()*.

Use the following calls to request voltage and temperature parameters:

- *appGetCurrentRailVoltage*
- *appGetCurrentTemperature*

The process point is set by the application by calling *dpcmSetCurrentProcessPoint* (see 10.23.11.1.1).

The switching bits information is requested by calling *appGetSwitchingBits* (see 10.23.13.5).

10.21 Parasitic analysis

Parasitic analysis typically requires the calculation of the interconnect and driver models associated with an interconnect network that connects drivers and receivers.

10.21.1 Assumptions

The following represents a procedural interface between an application and a library for the purpose of exchanging parasitic information. The interface is capable of handling both linear and nonlinear devices. Some applications and libraries are not able nor desire to process all the potential element types the interface is capable of processing. An application that encounters devices it is not equipped to process shall use a suitable default and optionally issue an informative message.

10.21.2 Parasitic networks

A *parasitic network* is a collection of network fragments called *subnets*. Each subnet can contain a collection of interconnected parasitic elements such as resistors, capacitors, inductors, and clamping diodes. The application needs to assemble the collection of subnets into a single system representing the entire parasitic network that exists between the driver and receiver(s). The library needs to supply subnets for those portions of the network that exist within a library element and calculate the reduced models required for its calculation of delay and slew.

When analyzing the network a cell is driving, the application shall gather all the component parts (including pin parasitics) and assemble them into a complete network. This network is then presented to the driving cell for analysis.

A *parasitic element* is a device that alters the electrical characteristics of a network. A parasitic network is comprised of a collection of parasitic elements that are connected to terminals called *nodes*. During the calculation of interconnect characteristics such as delay and slew, the parasitic network is analyzed to determine the effect it has on the calculation. Each cell that represents a driver or sink has some type of parasitic network. So the effects of these parasitics on delay and slew are properly considered, and the designed interconnect network, as well as driver and sink networks, shall be merged into a single unified system for analysis.

10.21.3 Basic definitions

This subclause defines some basic terms used during parasitic analysis.

10.21.3.1 Logical pins and internal nodes

A cell can have a logical pin that represents the entry and/or exit point of a logical signal. A pin can be an entry or exit point, but not both at the same time. This logical entry point shall have one or more connection points called *ports*, where the signal retains the same logical function if it is connected to any of these physical locations. In many situations, there can be cells where the logical pin is not the only location where analysis measurements shall take place. For example, a very large cell or macro can have a single logical pin that is routed to several internal points for use by the cell.

10.21.3.2 Physical ports

A *physical port* is a physical attachment point where an electrical signal can enter or exit the cell. Like logical pins, the ports can be both an entry and exit point, but not at the same time. Because there can be many physical ports associated with a pin, the library shall supply a mapping system (*port numbers*) that correlates the physical ports with their associated nodes in the parasitic representation of that pin.

The application shall use port numbers to connect nodes in the parasitic network external to the cell to nodes in the internal parasitic network for the pin. Port names for a pin are contained within an array associated with each pin. They are identified by their index in the array for rapid mapping. The index of the first port shall be zero (0).

10.21.3.3 Nodes

Nodes represent points where electrical elements, such as resistors, capacitors, inductors and clamping diodes, can be connected. They are represented by positive integers beginning at zero (0). Each node has a type, such as intermediate node or termination node. A *termination node* is a point where both elements can be connected and measurements taken. An *intermediate node* is a point where only elements can be connected. The node number zero (0) is assumed to be an internal connection to *ground*.

10.21.3.4 Terminating points

A *terminating point* is where some measurement can be viewed. It can be a source or a sink. The collection of interconnect parasitic subnets and parasitic subnets within the source or sink pins comprise the parasitic network under analysis. Each subnet shall be thought of by its creator as a complete network. The application shall take this collection of subnets and merge them into a single uniform parasitic network. To do this, both the application and library shall use a consistent data arrangement that can be viewed and manipulated.

10.21.3.5 Parasitic elements

A parasitic element is a device contained within a network that alters the performance of a circuit. Each parasitic element contains the type of element it represents; such as a resistor, capacitor, inductor, or clamping diode; the list of nodes to which it is connected; and the values associated with that element.

10.21.3.6 Subnets

A subnet consists of an ordered list of node types, a *node map* (mapping nodes in the subnet to the overall network's nodes), a list of elements, a *port map* (an ordered list of the nodes in the subnet that represent ports), a change flag, and a pair of linking pointers to other subnets.

An *interconnect network* is composed of many subnets. At a minimum, there shall be one subnet for each driver, one for each receiver, and one for the interconnect between the drivers and receivers. The interconnect subnet can be subdivided to account for voltage and temperature variations and for different interconnect technologies.

The library shall supply the subnets for its drivers and receivers, which are scaled correctly to account for the current voltages, temperature, and process point. The application is responsible for creating the interconnect subnets. In some situations, an extraction was performed at voltage, temperature, and process points different from the current conditions. To account for these changes and any on-chip process variation, the application shall call *EXPOSE* APIs, allowing the library to scale each interconnect subnet the application created.

Each element in a subnet has a collection of attached nodes. Each node is represented by a unique positive integer. Node zero is reserved for *ground*; in the absence of a port information, node one shall be mapped to the logical pin associated with the subnet. Each node also has an associated node type that indicates whether the node is a measurement point or just an intermediate point in the network. The terminus of each timing arc that begins or ends at this pin shall be mapped to a measurement point in the subnet.

10.21.4 Parasitic element data structure

Each parasitic element is represented by a data structure (*parasiticElement*). The contents of this structure vary depending on the parasitic element being represented. A member of the data structure (*elementType*) identifies the type of the parasitic element. Each type of parasitic element has a different content. The structure defined for a parasitic element is large enough to handle the most complex parasitic element, while being space conscientious for simpler elements, such as resistors. The data structure is composed of several fields, as shown in Table 216; some fields are undefined for certain types of parasitic elements.

Table 216—parasiticElement structure

DCL syntax
<pre>typedef(ivcurve): result(double var [*] var: voltage, current); typedef(parasiticElement): result(int var: elementType, node0Index, node1Index, node2Index, node3Index; double var: value0, value1, value2, value3, value4; var ivcurve var: ivCurve; string var : modelName; int var : railIndex; int var: node0Position, node1Position, node2Positon, nodePposiiton; void var : ownerPrivate);</pre>
C syntax
<pre>enum DCM_ElementTypes { DCM_RESISTOR, DCM_CAPACITOR, DCM_INDUCTOR, DCM_MUTUAL_INDUCTANCE, DCM_LOSSLESS_TRANSMISSION_LINE_TIME_DELAY_BASED, DCM_LOSSLESS_TRANSMISSION_LINE_FREQUENCY_BASED, DCM_LOSSY_TRANSMISSION_LINE_RLC, DCM_LOSSY_TRANSMISSION_LINE_RC, DCM_LOSSY_TRANSMISSION_LINE_LC, DCM_LOSSY_TRANSMISSION_LINE_LG, DCM_DIODE, DCM_VOLTAGE_SOURCE }; typedef DCM_STRUCT DCM_ivCurve;</pre>

```

typedef struct {
    DCM_ElementTypes elementType;
    DCM_INTEGER node0Index, node1Index, node2Index, node3Index;
    DCM_DOUBLE value0, value1, value2, value3, value4;
    const DCM_ivCurve *ivCurve;
    const DCM_STRING modelName;
    DCM_INTEGER railIndex;
    DCM_INTEGER node0Position, node1Position, node2Position,
    node3Position;
    DCM_VOID ownerPrivate;
} DCM_ParasiticElement;

typedef DCM_ParasiticElement *DCM_ParasiticElement_ARRAY;

```

Table 217 defines the semantics associated with the fields of the *parasiticElement* structure. The structure shall contain the complete set of structure fields (i.e., the structure never varies in size), but the number of members that are valid vary by *elementType*.

Table 217—Parasitic element variables

Variable name	Definition
elementType	The enumerated type of the parasitic element
node0	The node index of the first terminal on the parasitic element
node1	The node index of the second terminal on the parasitic element
node2	The node index of the third terminal on the parasitic element
node3	The node index of the fourth terminal on the parasitic element
value0	The first value associated with the parasitic element
value1	The second value associated with the parasitic element
value2	The third value associated with the parasitic element
value3	The fourth value associated with the parasitic element
value4	The fifth value associated with the parasitic element
modelName	The clamping diode model name
ivCurve	A pointer to the current versus voltage curve that models the clamping diode
railIndex	The voltage-source rail index
position0	The physical position of node0
position1	The physical position of node1
position2	The physical position of node2
position3	The physical position of node3
ownerPrivate	A pointer pointing to an owner private collection of data

10.21.4.1 elementType

elementType is an enumerated type that identifies the kind of parasitic element a structure represents, as shown in Table 218.

Table 218—DCM_ElementTypes

Enumerated name	Value	Definition
DCM_RESISTOR	0	Resistor
DCM_CAPACITOR	1	Capacitor
DCM_INDUCTOR	2	Inductor
DCM_MUTUAL_INDUCTANCE	3	Mutual inductance
DCM_LOSSLESS_TRANSMISSION_LINE_TIME_DELAY_BASED	4	Lossless time delay transmission line
DCM_LOSSLESS_TRANSMISSION_LINE_FREQUENCY_BASED	5	Lossless frequency based transmission line
DCM_LOSSY_TRANSMISSION_LINE_RLC	6	Lossy RLC transmission line
DCM_LOSSY_TRANSMISSION_LINE_RC	7	Lossy RC transmission line
DCM_LOSSY_TRANSMISSION_LINE_LC	8	Lossy LC transmission line
DCM_LOSSY_TRANSMISSION_LINE_LG	9	Lossy LG transmission line
DCM_DIODE	10	Clamping diode
DCM_VOLTAGE_SOURCE	11	Voltage source

10.21.4.2 Node index variable values

A parasitic subnet contains an array of nodes. Each parasitic element in the subnet can reference up to four of these nodes. Each of those nodes is represented by the corresponding index in the subnet's node map. Each type of parasitic element shall use preassigned structure variables for its terminals as defined in Table 219.

Table 219—Node variables

Element type	Indices
Resistor	<i>node0Index</i> = one end of the resistor <i>node1Index</i> = the other end of the resistor
Capacitor	<i>node0Index</i> = positive end of the capacitor <i>node1Index</i> = negative end of the capacitor
Inductor	<i>node0Index</i> = positive or dotted end of the inductor <i>node1Index</i> = other end of the inductor
Mutual inductance	<i>node0Index</i> = one inductor <i>node1Index</i> = other inductor Inductors are tracked by element number; that is, they are counted beginning at zero (0). The first inductor seen in the parasitic elements is identified as zero, the next as one, and so on.
All types of transmission line	<i>node0Index</i> = input port 1 of the transmission line <i>node1Index</i> = input port 2 of the transmission line <i>node2Index</i> = output port 1 of the transmission line <i>node3Index</i> = output port 2 of the transmission line
Diode	<i>node0Index</i> = positive end of the diode <i>node1Index</i> = negative end of the diode
Voltage source	<i>node0Index</i> = the positive node <i>node1Index</i> = the negative node

10.21.4.3 Parasitic element values

Each parasitic element can contain one or more values. The value variable assignments for each type of parasitic element are defined in Table 220.

Table 220—Value variables

Element type	Values
Resistor	<i>value0</i> = resistance
Capacitor	<i>value0</i> = capacitance
Inductor	<i>value0</i> = inductance
Mutual inductance	<i>value0</i> = coupling coefficient
Lossless transmission line time delay based	<i>value0</i> = characteristic impedance <i>value1</i> = time delay
Lossless transmission line frequency based	<i>value0</i> = characteristic impedance <i>value1</i> = frequency <i>value2</i> = normalized electrical length
Lossy transmission lines	<i>value0</i> = resistance/unit length <i>value1</i> = inductance/unit length <i>value2</i> = conductance/unit length <i>value3</i> = capacitance/unit length <i>value4</i> = length of the line — For RLC networks, <i>value0</i> , <i>value1</i> , <i>value3</i> , and <i>value4</i> shall have valid values. — For RC networks, <i>value0</i> , <i>value3</i> , and <i>value4</i> shall have valid values. — For LC networks, <i>value1</i> , <i>value3</i> , and <i>value4</i> shall have valid values. — For LG networks <i>value1</i> , <i>value2</i> and <i>value4</i> shall have valid values.
Diode	<i>value0</i> = cut-in voltage
Voltage source	<i>value0</i> -- <i>value4</i> are undefined. <i>railIndex</i> carries the index of the rail voltage to be used as the voltage source.

10.21.4.4 Clamping diodes

A *clamping diode* has a nonlinear current-voltage characteristic. There are three alternatives for modeling a clamping diode. The first order approximation of this characteristic is the cut-in voltage. When the voltage observed across the diode is less than the cut-in voltage ($V_{plus} - V_{minus}$) the current passing through the diode is zero (0). When the voltage is greater than or equal to this voltage, the diode's resistance is assumed to be zero (0).

For more accurate modeling of a diode, a voltage-versus-current curve can be represented in an IV-curve structure (*ivcurve* in DCL and *DCM_ivCurve* in C). This structure contains a pair of synchronized arrays, one containing voltages and the other containing the current passing through the diode at the corresponding voltage. The voltage array contains a monotonically increasing list of voltages. Each member in the voltage array *voltage[i]* represents a measurement point, whereas the corresponding member of the current array *current[i]* is the current flowing through the diode when that voltage is applied. There shall be the same number of elements in the voltage array as in the current array and *ith* element of the voltage array shall be paired with the *ith* element of the current array.

Finally, the most accurate modeling approach is to use *modelName* to represent the name of a clamping diode model. It shall be the responsibility of the library to locate this model.

10.21.4.5 ownerPrivate pointer

ownerPrivate is a pointer-sized space that the owner of the subnet is allowed to use for its own purposes. This member can be set to any legal value, but if it is a pointer, the data to which it points shall be managed and destroyed by its creator. When an element structure is deleted, this member shall not be altered in any way.

NOTE—The application can use this field as a pointer to the element in the application's memory.

10.21.5 Coordinates

Each element that is a member of a parasitic subnet has a position indicating where that component is located on the chip. The location of the element shall be its physical location. The location shall consist of x , y coordinates and the interconnect level. The x and y coordinates shall be the distance from the lower left corner of the chip. The interconnect level is the level of interconnect counted vertically from the substrate.

It shall be the responsibility of the library to adjust parasitic coordinates to reflect their position relative to the chip when presenting parasitic subnets to the application (Table 221).

Table 221—Coordinate structure

DCL syntax
<pre>typedef (Coordinate): result(float var: x,y; int var: interconnectLayer);</pre>
C syntax
<pre>typedef struct { float x, y; int z; } Coordinate;</pre>

10.21.6 Parasitic subnets

A *parasitic subnet* consists of a collection of parasitic elements that represents a portion of the interconnect network, including subnets contained within cells, as shown in Table 222.

Table 222—parasiticSubnet structure

DCL syntax
<pre>typedef (parasiticSubnet): result(int var: changed; var parasiticElement var[*] var : parasiticElementArray; int var[*] var: portMap, nodeMap, nodeTypeList; var parasiticSubnet var: nextSubnet, prevSubnet; TECH_TYPE var: techFamily; var Coordinates var[*]var: elementPosition; void var: ownerPrivate);</pre>
C syntax
<pre>enum DCM_NodeTypes { DCM_INTERMEDIATE_NODE, DCM_SINK_NODE, DCM_SOURCE_NODE, DCM_AGGRESSOR_SOURCE_NODE }; typedef DCM_NodeTypes DCM_NodeTypes_ARRAY; typedef struct { DCM_INTEGER changed; DCM_ParasiticElement_ARRAY *parasiticElementArray; DCM_INTEGER_ARRAY *portMap, *nodeMap; DCM_NodeTypes_ARRAY *nodeTypeList; DCM_ParasiticSubnet *nextSubnet, *prevSubnet; DCM_TechFamilyNugget *techFamily; DCM_VOID ownerPrivate; } DCM_ParasiticSubnet;</pre>

Table 223 shows the set of possible node types (for *DCM_NodeType*).

Table 223—DCM_NodeTypes

Node type	Value	Definition
DCM_INTERMEDIATE_NODE	0	The typical internal node in a parasitic graph; this is a point where parasitic elements with a common connection point shall be attached. Intermediate nodes are not measurement points where delays and slews are calculated.
DCM_SINK_NODE	1	DCM_SINK_NODE represents a node where a receiver is attached. Sink nodes are measurement points to which delays and slews are calculated.
DCM_SOURCE_NODE	2	DCM_SOURCE_NODE represents a node where a driver is attached. Source nodes are measurement points from which delays and slews are measured.
DCM_AGGRESSOR_SOURCE_NODE	3	DCM_AGGRESSOR_SOURCE_NODE represents a node where a coupled aggressor driver is attached.
DCM_PARALLEL_DRIVER_NODE	4	DCM_PARALLEL_DRIVER_NODE represents a node to which one of a set of drivers acting in unison is attached.

10.21.6.1 Changed

changed represents whether this parasitic subnet has been modified since the last time the subnet was presented to the library. A change includes adding or deleting parasitic elements, changing the value of a parasitic element, changing the node ordering, or changing the configuration of the subnet or its relative placement with respect to other parasitic subnets. If a change has occurred, then the value of *changed* is 1 (one); otherwise it is 0 (zero).

10.21.6.2 parasiticElementArray

parasiticElementArray is a pointer to a *DCM_ARRAY* of pointers to *DCM_ParasiticElement* structures.

10.21.6.3 portMap

Port names shall be defined in an ordered list so they can be referenced by their position within that list. *portMap* is an array of node indices; it contains one element for each port associated with a logical pin. The node number contained at *portMap* index *n* is the node associated with port *n*. The name corresponding to *portMap* index *n* is the *n*th port name associated with the logical pin.

Because an interconnect subnet is not associated with a cell, it has no ports. The application shall help to ensure the port(s) of each subnet, which represent a pin's parasitics are correctly attached to the subnet(s) representing the design net where that pin is connected.

10.21.6.4 nodeMap

nodeMap is an array of nodes used by the subnet. *nodeMap* element 0 (zero) shall have a value of 0 (zero) and shall represent *ground*. *nodeMap* element 1 (one) shall represent a logical pin's default connection point. All other *nodeMap* values are consecutive positive integers.

10.21.6.5 nodeTypeList

nodeTypeList is an array of integers (*DCM_NodeTypes* enum in C) representing the type of each node in the subnet. The *ithnodeType* value in *nodeTypeList* represents the type of node corresponding to the *ith* entry in the node map of the subnet. The zeroth and first node types correspond to *ground* and the default port for a logical pin, respectively.

10.21.6.6 Link pointers

Parasitic subnets are connected in a linked list to form a parasitic network. *nextSubnet* and *prevSubnet* point to the next and previous subnets in the list, respectively. For the first or last subnet in the list, *prevSubnet* or *nextSubnet* shall have a value of 0 (zero).

10.21.6.7 techFamily

techFamily is a structure containing the *TECH_FAMILY* to which the subnet belongs. *Its contents shall not be modified once the subnet is created.*

10.21.6.8 elementPosition

The *elementPosition* array contains an array of coordinates. Each coordinate in the array represents a physical location within the design. Parasitic elements contain one or more indexes into this array. Parasitic elements shall have a position index for each valid node.

The array of element positions allows the sharing of coordinate information for those parasitic element nodes that have a common physical location.

10.21.6.9 ownerPrivate pointer

ownerPrivate is a pointer-sized space that the owner of the subnet is allowed to use for its own purposes. This member can be set to any legal value, but if it is a pointer, the data to which it points shall be managed and destroyed by its creator. When a subnet structure is deleted, this member shall not be altered in any way. When the library is the creator, the user private member may be an abstract type. The library may achieve this through casting or carrying parallel type definitions.

NOTE—The application can use this field as a pointer to the element in the application's memory.

10.21.6.10 Linked lists of subnets

A parasitic network is a linked list of subnets, with each subnet containing an array of parasitic elements. Each subnet shall be linked into the list by the application, which shall convert its parasitic element values to the units of the driving cell.

For parasitics that it owns, such as those between cells in an interconnect network, the application shall request that the library create a new subnet structure by calling *dpcmCreateSubnetStructure* (see 10.21.6.12). The number of parasitic elements to be placed in the subnet shall be specified in this call, and the subnet returned shall not contain *port-map*, *node-map*, nor *node-type* arrays (pointers to these arrays shall be set to zero (0) by the library).

The application shall then fill in the element array in this subnet. It shall also request that the library allocate appropriately sized *node-map* and *node-type* arrays (via *dcm_new_DCM_ARRAY*), set the corresponding pointers in the subnet to point to them, and fill in them in. The application shall not claim these arrays, nor shall it disclaim them when the subnet is disclaimed.

The value of the *portMap* pointer in the subnet shall be left at zero (0), and the application shall set the *ownerPrivate* pointer as it deems fit.

The application shall partition any network that traverses more than one technology, more than one voltage for any voltage rail, or different temperatures into multiple subnets, so that each subnet is confined to a single voltage set, temperature, and technology.

The application shall also ask the library for the parasitic subnets associated with the sink and source pins connected to the network, one pin at a time. In response, the library shall build a *parasiticSubnet* structure for each such pin. The parasitic values in a pin subnet provided by the library shall be scaled based on the current voltage, temperature, and process settings for the design. The library shall return a complete structure containing *port-map* (if needed), *node-map*, and *node-type* arrays.

The application (during the process of assembling the interconnect network) shall organize the individual subnets into a double-linked list by setting *nextSubnet* in each subnet to point to the next subnet in the list and *prevSubnet* in that subnet to point to the previous subnet in the list. The first subnet's *prevSubnet* shall have a value of zero (0), and the last subnet's *nextSubnet* shall also have a value of zero (0). There shall be no specific order assumed for or imposed on this list.

The application shall also adjust each subnet's *nodeMap* so the node numbers are unique within the network as a whole, except for entry 0 (zero), which shall be mapped to node 0, representing ground. All of this is depicted in Figure 6.

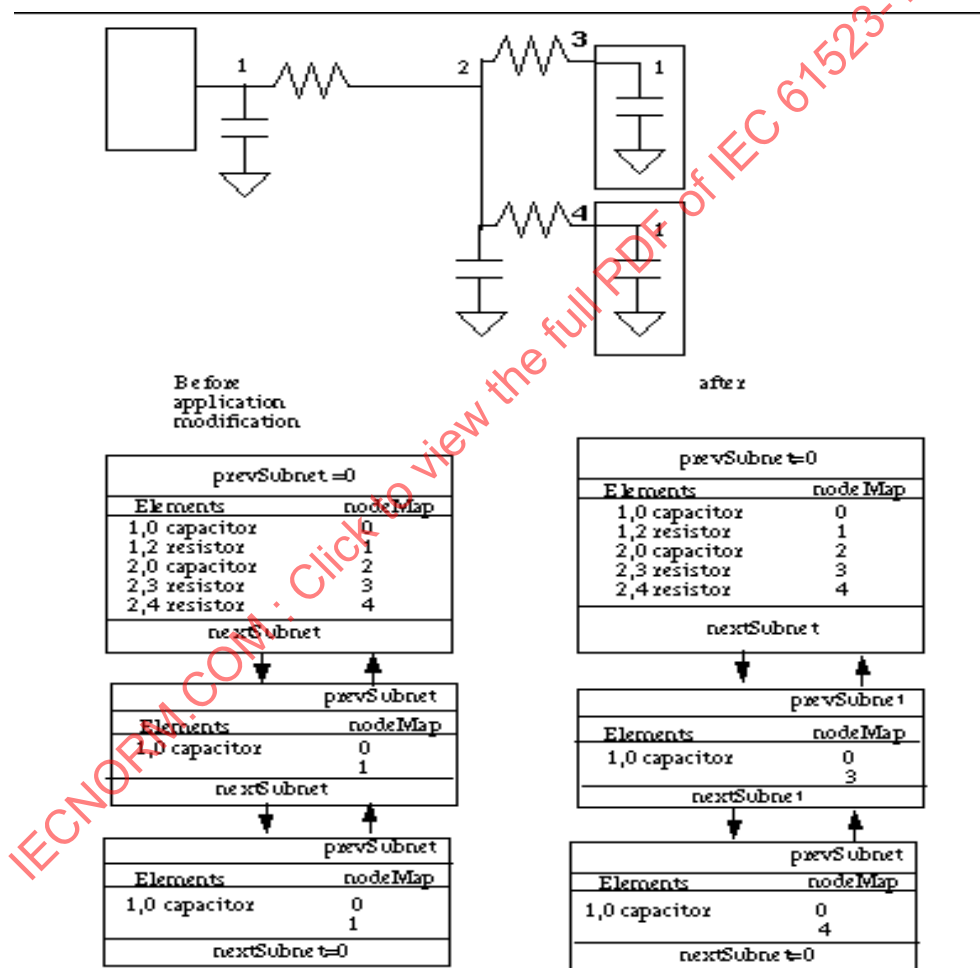


Figure 6—Subnet node mapping

NOTE—The application can determine from examining parasitic element coordinates within a library subnet whether or not additional parasitic elements are required. The application at its discretion may add these components to another subnet of its own creating using the node numbers corresponding to the reordered node numbers to specify where these parasitic elements should be attached.

To accommodate on-chip process variations, two parasitic networks can exist for each net, one containing element values for the minimum and the other having values for the maximum of the on-chip process uncertainty (see 10.23.8.3.1). These two networks shall be identical in all other respects, including the order in which their elements and nodes appear. If on-chip process variation is not to be modeled, then only one network shall be used.

When the library returns a subnet to the application, the application shall claim that subnet to prevent it and any of its members from being deleted. When the application is finished with any subnet, it shall unlink it from the list to which it was attached, setting the subnet's *nextSubnet* and *prevSubnet* pointers to zero (0). Only then shall the application disclaim that subnet.

The application shall neither claim nor disclaim the arrays within a subnet. The library shall disclaim these arrays when the subnet of which they are members is disclaimed.

10.21.6.11 Parasitic subnet structure construction

The application shall call the library for the construction of a new subnet so the library can manage the memory used. When the application calls *dpcmCreateSubnetStructure*, the library shall return a newly created subnet.

10.21.6.12 dpcmCreateSubnetStructure

Table 224 provides information on *dpcmCreateSubnetStructure*.

Table 224—*dpcmCreateSubnetStructure*

Function name	<i>dpcmCreateSubnetStructure</i>
Arguments	Number of parasitic elements
Result	Parasitic subnet
Standard Structure fields	None
DCL syntax	<pre>EXPOSE(dpcmCreateSubnetStructure): passed(int: numberOfParasiticElements) result(var parasiticSubnet: parasiticData);</pre>
C syntax	<pre>typedef struct { DCM_ParasiticSubnet *parasiticData; } T_ParasiticSubnet; int dpcmCreateSubnetStructure (const DCM_STD_STRUCT *std_struct, T_ParasiticSubnet *rtn, DCM_INTEGER numberOfParasiticElements);</pre>

This allocates a new subnet structure, creates an element array, and attaches that array to the structure. The element array shall be sufficiently large to hold the number of parasitic elements specified in the argument *numberOfParasiticElements*. The library shall set the *portMap*, *nodeMap*, *nodeTypeList*, *nextSubnet*, *prevSubnet*, and *ownerPrivate* fields of the new subnet to zero (0). The changed field shall also be set to zero (0), and the *techFamily* pointer shall be set to point to the technology family referenced in the Standard Structure.

10.21.6.13 Example array

Below is an example of the DCL code used to create a new subnet.

```
/******
```

Entry point the application calls to get a blank subnet for a specific number of elements

```
*****/
```

```

EXPOSE(dpcmCreateSubnetStructure):
    passed(integer: numberOfParasiticElements)
    local(
        // create the subnet structure
        var parasiticSubnet: pe = new(var parasiticSubnet)
    &
        // set the changed switch to zero
        pe.changed = 0,

        // create an parasiticElementArray of the specified size
        pe.parasiticElementArray =
            new(var parasiticElement var [numberOfParasiticElements]),
    &
        // for every parasitic element
        for((integer: loopCtr = 0),
            (loopCtr < numberOfParasiticElements),
            (loopCtr = loopCtr+1))
            // create a parasitic element structure
            local(
                pe.parasiticElementArray[loopCtr] =
                    new(var parasiticElement)
            &
                pe.parasiticElementArray[loopCtr].ownerPrivate = nil)
        &
            // set port- and node-map, node-type-list pointers to zero
            pe.portMap = nil,
            pe.nodeMap = nil,
            pe.nodeTypeList = nil
        &
            // set the linked list pointer to zero
            pe.nextSubnet = nil,
            pe.prevSubnet = nil
        &
            // set tech family
            pe.techFamily = current_tech_type().TT
        &
            pe.ownerPrivate = nil
    )
    result(var parasiticSubnet: parasiticData=pe);

```

10.21.6.14 dpcmGetDefaultInterconnectTechnology

Table 225 provides information on dpcmGetDefaultInterconnectTechnology.

Table 225—dpcmGetDefaultInterconnectTechnology

Function name	dpcmGetDefaultInterconnectTechnology
Arguments	None
Result	Default interconnect technology
Standard Structure fields	None
DCL syntax	EXPOSE(dpcmGetDefaultInterconnectTechnology): result(TECH TYPE: techFamily);
C syntax	<pre>typedef struct { const DCM_TechFamilyNugget *techFamily; } T_Technology; int dpcmGetDefaultInterconnectTechnology (const DCM_STD_STRUCT *std_struct, T_Technology *rtn);</pre>

This returns a technology the application can use when it cannot otherwise determine which technology owns a particular interconnect subnet back to the application. It is only valid in the *DEFAULT* technology; the application shall only call this function using a *Standard Structure* that is set to the *DEFAULT* technology. The technology returned shall not be used for subnets obtained from the library other than those created via *dpcmCreateSubnetStructure* (see 10.21.6.12).

10.21.6.15 dpcmScaleParasitics

Table 226 provides information on dpcmScaleParasitics.

Table 226—dpcmScaleParasitics

Function name	dpcmScaleParasitics
Arguments	Subnet to scale, Extraction operating point, Extraction voltages, Extraction process point, Extraction temperature
Result	Scaled parasitic subnets
Standard Structure fields	calcMode
DCL syntax	EXPOSE(dpcmScaleParasitics): passed(parasiticSubnet: subnet; int: extractionOpPointIndex; double: positiveExtractionVoltage, negativeExtractionVoltage, extractionTemperature, extractionProcessPoint) result(var parasiticSubnet: minSubnet, maxSubnet);
C syntax	<pre>typedef struct { DCM_ParasiticSubnet *minSubnet, *maxSubnet; } T_ScaledSubnets; int dpcmScaleParasitics (const DCM_STD_STRUCT *std_struct, T_ScaledSubnets *rtn, const DCM_ParasiticSubnet *subnet, DCM_INTEGER extractionOpPointIndex, DCM_DOUBLE positiveExtractionVoltages, DCM_DOUBLE negativeExtractionVoltages, DCM_DOUBLE extractionTemperature, DCM_DOUBLE extractionProcessPoint);</pre>

This scales the parasitics within a subnet to compensate for changes in the operating point, process point, voltages, and temperature. The library shall return subnets containing elements whose values are properly scaled for the differences between any extraction conditions and the current analysis conditions back to the application.

This function also provides modeling of on-chip process variation by returning pointers to two scaled subnets, one with element values for the minimum and another with values for the maximum of the on-chip process uncertainty (see 10.23.8.3.1). These two subnets shall be identical in all other respects, including the order in which their elements and nodes appear. If on-chip process variation is not supported by the library, the pointers returned shall be identical, pointing to the same subnet.

The application shall call this function to scale all subnets not contained within a cell. The application shall not call this function to scale subnets within a cell. The application shall help to ensure a subnet due to be scaled is completely owned by the technology called on to scale it. The library shall then set the *techFamily* members in the subnets it returns to point to that technology. If the technology does not support this call, the subnet values shall not be scaled and on-chip process variation shall not be modeled for that subnet.

dpcmScaleParasitics takes several passed arguments that enable it to perform the scaling operation, as detailed in the next subclauses.

10.21.6.16 extractionOpPointIndex

extractionOpPointIndex is the index into the operating point array (see 10.23.10.1), which identifies the extraction operating point. The valid range for *extractionOpPointIndex* is the integers from zero (0) through the index of the last operating point in the array. If the application chooses to supply the individual components of a process point, such as extraction temperature, then the value of *extractionOpPointIndex* shall be –1.

10.21.6.17 positiveExtractionVoltage

positiveExtractionVoltage represents the greatest positive voltage the driving cell applied to the subnet during the extraction process. *positiveExtractionVoltage* is only valid when the value of *extractionOpPointIndex* is –1.

10.21.6.18 negativeExtractionVoltage

negativeExtractionVoltage represents the most negative voltage the driving cell applied to the subnet during the extraction process. *negativeExtractionVoltage* is only valid when the value of *extractionOpPointIndex* is –1.

10.21.6.19 extractionTemperature

extractionTemperature represents the extraction temperature in that portion of the chip where the subnet is located. *extractionTemperature* is only valid when the value of *extractionOpPointIndex* is –1.

10.21.6.20 extractionProcessPoint

extractionProcessPoint represents the process point used during the extraction process. *extractionProcessPoint* is only valid when the value of *extractionOpPointIndex* is –1 (see 10.23.11.1).

10.21.7 Pin parasitics

The application shall provide the library with a complete interconnect network with which the library shall perform its calculations. The application shall ask the library for pin parasitics subnets to be included in this network.

Each pin-parasitics subnet returned by the library shall be a complete *parasiticSubnet* structure containing *port-map* (if needed), *node-map*, and *node-type* arrays. The entries in the *node-map* array shall be initialized to their index values, entry 0 being set to 0, entry 1 to 1, and so on. The library shall set the changed, *nextSubnet* and *prevSubnet* fields of the new subnet to zero (0). The *techFamily* pointer shall be

set to point to the technology family to which the pin's cell belongs (referenced in the Standard Structure). The library shall set the *ownerPrivate* pointer as it determines appropriate.

A pin's parasitic network can be different depending on whether it is driving the net, in a high impedance state, or receiving a logical signal. For example, a pin that is bidirectional can act as a receiver or a driver, whereas a tristate output pin can be in an active or high impedance state, depending on whether or not it is driving the attached net.

A pin also has separate parasitic subnets containing different element values for rising and falling signals. For example, pin capacitance can depend on signal direction. If the library models on-chip process variation, then a pin can have different subnets with different element values representing the minimum and maximum of the on-chip process uncertainty (see 10.23.11). All of these subnets shall be identical in all other respects, including the order in which their elements and nodes appear. The application is responsible for stitching the correct subnets into the overall interconnect work for the calculation desired.

There are two separate calls for obtaining pin parasitics, one for sink (receiver or high impedance) and one for source (driver) pin roles. For both calls, the library shall return up to four subnets (via pointers), representing all possible combinations of rising and falling signals and minimum and maximum on-chip process variation. If, for a given pin, the same parasitic subnet is used for both rising and falling signals, then the pointers returned for rising signals shall be identical to those for falling signals. If the library does not support on-chip process variation, then the pointers returned for the minimum of the on-chip process uncertainty shall be identical to those for its maximum.

Source-pin parasitics shall include only those elements needed to represent the constant portion of an output pin's parasitics. Elements representing the active part of a driver pin such as output resistance or admittance shall not be included. Instead, the library can include driver models for an output pin in the XWF structures (see 10.23.8), for example, associated with that pin.

The library shall return pointers to subnets appropriate for the current analysis conditions. Different conditions, with regard to operating point, for example, can yield subnets that are topologically as well as numerically different.

For a net that has multiple, independent drivers directly connected to it (not counting coupled aggressor drivers), the application shall create a different set of parasitic networks for use when each one of those drivers drives the net. For a particular driver, each network in such a set shall include source-pin parasitics for that driver and sink-pin parasitics for the other drivers. These networks shall then be used when load and interconnect models (see 10.21.9) are created for that driver and for the net as driven by the driver.

When multiple, parallel drivers act in unison, source-pin parasitics shall be included for each one of them in every parasitic network constructed for use when they drive a net. This same set of networks shall then be used when creating load models for each parallel driver and interconnect models for the net when driven by these drivers.

Under all circumstances, source-pin parasitics shall be used for all aggressor drivers.

Although a bidirectional pin can act as an input or an output, as observed at the pin, it shall not do both simultaneously. Consequently, when a parasitic network for a net is created for use when a bidirectional pin drives that net, the application shall include source parasitics only for that pin. Sink parasitics for the pin shall not be included under such circumstances. Conversely, while creating a parasitic network for use when a different pin drives the net, only the sink parasitics for that bidirectional pin shall be included.

10.21.7.1 dpcmGetSinkPinParasitics

Table 227 provides information on dpcmGetSinkPinParasitics.

Table 227—dpcmGetSinkPinParasitics

Function name	dpcmGetSinkPinParasitics
Arguments	Sink pin pointer
Result	Parasitic subnets
Standard Structure fields	CellName, cellData (timing), pathData (timing-pin-specific), calcMode
DCL syntax	<pre>EXPOSE (dpcmGetSinkPinParasitics) : passed (pin: sinkPin) result (var parasiticSubnet: risingMinSubnet, fallingMinSubnet, risingMaxSubnet, fallingMaxSubnet);</pre>
C syntax	<pre>typedef struct { DCM_ParasiticSubnet *risingMinSubnet, *fallingMinSubnet, *risingMaxSubnet, *fallingMaxSubnet; } T_PinParasitics; int dpcmGetSinkPinParasitics (const DCM_STD_STRUCT *std_struct, T_PinParasitics *rtn, const DCM_PIN sinkPin);</pre>

This returns pointers to parasitic subnets for a sink pin. These subnets shall be constructed from memory allocated by the library that is suitable for modification by the application. If any of the subnets within the scope of a single call are identical, the same pointer shall be returned for them. It shall be a severe error to call this function for a pin that cannot act as a sink.

10.21.7.2 dpcmGetSourcePinParasitics

Table 228 provides information on dpcmGetSourcePinParasitics.

Table 228Table 228—dpcmGetSourcePinParasitics

Function name	dpcmGetSourcePinParasitics
Arguments	Source pin pointer
Result	Parasitic subnets
Standard Structure fields	CellName, cellData (timing), pathData (timing-pin-specific), calcMode
DCL syntax	<pre>EXPOSE (dpcmGetSourcePinParasitics) : passed (pin: sourcePin) result (var parasiticSubnet: risingMinSubnet, fallingMinSubnet, risingMaxSubnet, fallingMaxSubnet);</pre>
C syntax	<pre>typedef struct { DCM_ParasiticSubnet *risingMinSubnet, *fallingMinSubnet, *risingMaxSubnet, *fallingMaxSubnet; } T_PinParasitics; int dpcmGetSourcePinParasitics (const DCM_STD_STRUCT *std_struct, T_PinParasitics *rtn, const DCM_PIN sourcePin);</pre>

This returns pointers to parasitic subnets for a source pin back to the application. These parasitic subnets shall be constructed from memory allocated by the library that is suitable for modification by the application. If any of the subnets within the scope of a single call are identical, the same pointer shall be returned for them. It shall be an error to call this function for a pin that cannot act as a source.

10.21.7.3 dpcmGetPortNames

Table 229 provides information on dpcmGetPortNames.

Table 229—dpcmGetPortNames

Function name	dpcmGetPortNames
Arguments	Pin pointer
Result	String array of port names
Standard Structure fields	CellName, cellData (timing), pathData(timing-pin specific)
DCL syntax	<pre>EXPOSE(dpcmGetPortNames): passed(pin: pinPointer) result(string[*]: portNames);</pre>
C syntax	<pre>typedef struct { DCM_STRING_ARRAY *portNames; } T_PortNames; int dpcmGetPortNames (const DCM_STD_STRUCT *std_struct, T_PortNames *rtn, const DCM_PIN pinPointer);</pre>

In many library cells, a single logical pin can have several physical ports where an interconnect network can be attached. These different attachment points affect the analysis of that interconnect. Ports are named within in a cell and are associated with a logical pin.

Both the application and library need quick references to these ports while performing an analysis. To enable this rapid access, the port names for each pin in a cell shall form an ordered list, indexed by the port numbers for that pin. *dpcmGetPortNames* returns this list of the port names associated with a pin.

dpcmGetPortNames has one argument, *pinPointer*, which identifies to the library the pin on a cell for which a port name list is requested. It returns one result, *portNames*, an array of the port names for that pin. The index of each name in this array is the number of the corresponding port.

Other calls referencing a port associated with this pin shall do so using the index of that port's name in this array. It shall be an error to use a port index that is not within the bounds of the port name array for the associated pin.

10.21.8 Modeling internal nodes

In situations where the internal network of a cell is complicated enough to warrant separate analysis for individual internal nodes, the library shall use *IMPORT* or *EXPORT* clauses to associate these internal nodes with the appropriate pins. The application shall treat each identified imported or exported node as a valid measurement point associated with the input or output pin present in the corresponding clause. Each such node shall be connected to one or more timing arcs which begin or end at the node. In such a situation, the library can also use the input or output pin as a measurement point.

The application shall detect the presence of an *IMPORT* or *EXPORT* clause in a model through a library call to *newNetSinkPropagateSegments()* or *newNetSourcePropagateSegments()*, respectively. However, the pin involved can be either an input or an output. The following rules shall govern how the application creates interconnect timing arcs in response to these calls:

- If an import pin is an input, the application shall create interconnect arcs from each driver pin on the connected net to the internal sink node (the node that is the target of the import). These arcs shall be created in lieu of those that would otherwise be directed from the driver pins to the import pin.

- If an import pin is an output, the application shall create interconnect arcs from each driver pin on the connected net, excluding the import pin, to the internal sink node. These arcs shall be created in addition to those directed from the import pin to the sink pins connected to the net.
- If an export pin is an input, the application shall create interconnect arcs from the internal source node (the node that is the target of the export) to each sink pin on the connected net, excluding the export pin. These arcs shall be created in addition to those directed from the driver pins connected to the net to the export pin.
- If an export pin is an output, the application shall create interconnect arcs from the internal source node to each sink pin on the connected net. These arcs shall be created in lieu of those that would otherwise be directed from the export pin to the sink pins.

For a bidirectional pin or tristate output, an imported or exported internal node shall be associated with the source or sink parasitic subnet for that pin, but not with both.

Because a bidirectional pin can act as either an input or an output, the application shall use the pin handle provided in each call to determine to which of these roles the call applies. For this to be possible, the application shall provide different, distinguishable handles for this pin in the input pin and output pin lists supplied in the *Standard Structure* for the *modelSearch* function call made for the cell involved.

The application shall continue to call for the typical common pin-related data, such as pin capacitance or pin parasitics, using the associated pin, not an internal node. The parasitic subnet for a pin shall contain all the elements that make up the network between the pin and all the internal sink or source nodes.

A “logical” pin shall be used to obtain the pin parasitics since the pin’s subnet is a network where every node can be reached from every other node. To eliminate the ambiguity of which of these nodes owns this subnet, the logical pin shall always have that role. The library shall identify the location of each internal node in this subnet.

The *pathData* pointer included in the *newTimingPin* call (used to define an internal node) is required when the application requests the library map this node to a node in a parasitic subnet.

10.21.8.1 Mapping parasitic subnet nodes to model nodes

When the application stitches a pin’s parasitic subnet (supplied by the library) to the interconnect network it created, it shall map the ports and internal nodes in the cell’s timing model that are associated with that pin to the nodes in this parasitic subnet, as shown in Table 230. For ports, this is done using the *portMap* in the subnet. To do this for internal nodes, the application shall call *dpcmIdentifyInternalNode* for each internal node associated with the pin. It shall be illegal for an internal node to be represented in more than one subnet in a cell’s model.

10.21.8.2 dpcmIdentifyInternalNode

Table 230 provides information on application timing arcs.

Table 230—Application timing arcs

Function name	dpcmIdentifyInternalNode
Arguments	None
Result	Parasitic subnet node number
Standard Structure fields	CellName, cellData (timing), pathData(timing-pin-specific)
DCL syntax	EXPOSE(dpcmIdentifyInternalNode) : result(int: nodeNumber);

C syntax	<pre>typedef struct { DCM_INTEGER nodeNumber; } T_NodeNumber; int dpcmIdentifyInternalNode (const DCM STD STRUCT *std_struct, T_NodeNumber *rtn);</pre>

This returns the *nodeMap* index of the node within the appropriate subnet that corresponds to the internal node identified by the *pathData* pointer in the *Standard Structure* back to the application.

10.21.9 Load and interconnect models

The application, after building its own parasitic networks and appending pin subnets to them, shall request that the library build load models for the networks' drivers and interconnect models for the paths from the drivers to the sinks on those networks. The library shall represent each of these interconnect networks using a model of a form that is proprietary to the library and shall return both these network models and the requested load and interconnect models to the application.

Each load model shall contain sufficient information so that the library can later identify the associated driver node in the corresponding network using that model. For a network with multiple, parallel drivers, the load models for the drivers shall contain sufficient information so that all of the driver nodes can be identified when these models are used together.

Each interconnect model and the load model for the associated driver pin shall together contain sufficient information so that the library can later identify the driver and sink nodes in the corresponding network using those models. For a network with multiple, parallel drivers, an interconnect model and the load models for all of these drivers shall contain sufficient information so that all the drivers can be identified when these models are used together.

To accommodate on-chip process variations, the application can pass pointers to two parasitic networks for each net to the library, one with element values for the minimum and another with values for the maximum of the on-chip process uncertainty (see 10.23.11). These two networks shall be identical in all other respects, including the order in which their elements and nodes appear. The library shall return two corresponding load or interconnect models and two corresponding network models to the application, all via pointers of type *DCM_STRUCT**.

If on-chip process variation is not modeled, these networks shall be completely identical, as shall the corresponding library net models. The parasitic-network pointers passed to the library shall then be the same, as shall the load- or interconnect-model pointers and the network-model pointers returned to the application.

During the first request for load or interconnect models for a net, the application shall supply null pointers for the library's network models. In subsequent requests to the library, the application shall supply the network model pointers returned in response to the previous request.

The application shall set (to a value of one (1)) the change field of the first subnet in a parasitic network that has changed since a load or interconnect model for that network was last requested. This shall be done whether the change was to an existing subnet, the insertion of a new subnet, the deletion of a subnet, or a change in the order the subnets were presented. The library can, thus, omit the reconstruction of what would otherwise be duplicate information.

The application shall claim each network model received from the library. When the application has determined that all the load and interconnect models derived from that network have been obtained, the application shall disclaim the network model. At this point, if no other reference to it exists, the memory used for the network model shall then be freed by the library.

The application shall claim the load and interconnect models returned to it if it needs to retain them beyond the time it makes the next call to any library function. When the application no longer needs a model that it has already claimed, it shall disclaim that model. Again, if no other reference to it exists, the memory used for the model shall then be freed by the library.

10.21.9.1 dpcmBuildLoadModels

Table 231 provides information on dpcmBuildLoadModels.

Table 231—dpcmBuildLoadModels

Function name	dpcmBuildLoadModels
Arguments	Parasitic networks, Library network models, Driving pin node number
Result	Library load models, Library network models
Standard Structure fields	CellName, cellData (timing), block, pathData (timing-pin-specific), toPoint
DCL syntax	<pre>EXPOSE(dpcmBuildLoadModels): passed(parasiticSubnet: minParasitics, maxParasitics; void: minNetModel, maxNetModel; int: minDrivingPinNodeNumber, maxDrivingPinNodeNumber) result(void: minLoadModel, maxLoadModel, newMinNetModel, newMaxNetModel);</pre>
C syntax	<pre>typedef struct { const DCM_STRUCT *minLoadModel, *maxLoadModel; const DCM_STRUCT *newMinNetModel, *newMaxNetModel; } T_BuildLoadModels; int dpcmBuildLoadModels(const DCM_STD_STRUCT *std_struct, T_BuildLoadModels *rtn, const DCM_ParasiticSubnet *minParasitics, const DCM_ParasiticSubnet *maxParasitics, const DCM_STRUCT *minNetModel, const DCM_STRUCT *maxNetModel, DCM_INTEGER minDrivingPinNodeNumber, DCM_INTEGER maxDrivingPinNodeNumber);</pre>

This returns pointers to the load models for a driver of an interconnect network, along with updated versions of the net models for the that network. The arguments passed are pointers to linked lists of parasitic subnets that represent the network and the attached pins, pointers to the net models and the node numbers of the driving pin for which the load models are to be created. The *toPoint* field in the *Standard Structure* shall be set to the driving pin.

If the parasitic networks passed to this function are identical, then *minParasitics* and *maxParasitics* shall have the same value. Similarly, if the net models passed are identical, then *minNetModel* and *maxNetModel* shall have the same value.

If the load models are identical, then the load-model pointers returned shall have the same value. If the new net models are identical, then the net-model pointers returned shall have the same value.

10.21.9.2 dpcmBuildInterconnectModels

Table 232 provides information on dpcmBuildLoadModels.

Table 232—dpcmBuildInterconnectModels

Function name	dpcmBuildInterconnectModels
Arguments	Parasitic networks, Library network models, Driving pin node number, Sink pin node number
Result	Library interconnect models, Library network models
Standard Structure fields	CellName, cellData (timing), block, pathData (timing-pin specific), fromPoint, toPoint
DCL syntax	<pre>EXPOSE (dpcmBuildInterconnectModels): passed (parasiticSubnet: minParasitics, maxParasitics; void: minNetModel, maxNetModel; int: minDrivingPinNodeNumber, minSinkPinNodeNumber, maxDrivingPinNodeNumber, maxSinkPinNodeNumber) result (void: minInterconnectModel, maxInterconnectModel, newMinNetModel, newMaxNetModel);</pre>
C syntax	<pre>typedef struct { const DCM_STRUCT *minInterconnectModel, *maxInterconnectModel; const DCM_STRUCT *newMinNetModel, *newMaxNetModel; } T_BuildInterconnectModels; int dpcmBuildInterconnectModels(const DCM_STD_STRUCT *std_struct, T_BuildInterconnectModels *rtn, const DCM_ParasiticSubnet *minParasitics, const DCM_ParasiticSubnet *maxParasitics, const DCM_STRUCT *minNetModel, const DCM_STRUCT *maxNetModel, DCM_INTEGER minDrivingPinNodeNumber, DCM_INTEGER minSinkPinNodeNumber, DCM_INTEGER maxDrivingPinNodeNumber, DCM_INTEGER maxSinkPinNodeNumber);</pre>

This returns pointers to models for the path from a driver of an interconnect network to one of the sinks on that network, along with updated versions of the net models for the that network. The passed arguments are pointers to linked lists of parasitic subnets that represent the interconnect network and the attached pins, pointers to the library's models for that network, and the node numbers of the driving and sink pins for which the interconnect models are to be created. The *fromPoint* field in the *Standard Structure* shall be set to the driving pin, whereas the *toPoint* field shall be set to the sink pin.

If the parasitic networks passed to this function are identical, then *minParasitics* and *maxParasitics* shall have the same value. Similarly, if the net models passed to this function are identical, then *minNetModel* and *maxNetModel* shall have the same value.

If the interconnect models are identical, then the interconnect-model pointers returned shall have the same value. If the new net models are identical, then the net-model pointers shall have the same value.

10.21.9.3 appGetInterconnectModels

Table 233 provides information on appGetInterconnectModels.

Table 233—appGetInterconnectModels

Function name	appGetInterconnectModels
Arguments	None
Result	Library interconnect models
Standard Structure fields	CellName, block, cellData (timing), pathData(timing-pin-specific), fromPoint, toPoint, sourceEdge, sinkEdge, calcMode

DCL syntax	EXTERNAL (appGetInterconnectModels) : result(void: minInterconnectModel, maxInterconnectModel);
C syntax	typedef struct { const DCM_STRUCT *minInterconnectModel, *maxInterconnectModel; } T_InterconnectModels; int appGetInterconnectModels (const DCM_STD_STRUCT *std_struct, T_InterconnectModels *rtn);

In the process of calculating a delay for a net or slew degradation across that net, performance can be greatly improved if the library can get access to interconnect models created previously. When interconnect models are created, the application shall associate them with the network being analyzed. When it is time to use these models for delay and slew calculation, the library shall ask the application for them by using *appGetInterconnectModels*.

appGetInterconnectModels returns the interconnect models created by the library for the path between the pins specified in the *Standard Structure*. If these interconnect models are identical, then the interconnect model pointers returned shall have the same value.

10.21.9.4 appGetLoadModels

Table 234 provides information on appGetLoadModels.

Table 234—appGetLoadModels

Function name	appGetLoadModels
Arguments	Pin pointer, Pin edge
Result	Library load models
Standard Structure fields	CellName, block, cellData (timing), calcMode
DCL syntax	EXTERNAL (appGetLoadModels) : passed(pin: pinPointer ; int: edge) result(void: minLoadModel, maxLoadModel);
C syntax	typedef struct { const DCM_STRUCT *minLoadModel, *maxLoadModel; } T_LoadModels; int appGetLoadModels(const DCM_STD_STRUCT *std_struct, T_LoadModels *rtn, DCM_PIN pinPointer, DCM EdgeTypes edge);

During the processing of cell delay and slew calculations, the driving cells need rapid access to precalculated load models. When load models are created, the application shall associate them with the driver pins being analyzed. When it is time to use these models for delay and slew calculation, the library shall ask the application for them by using *appGetLoadModels*.

appGetLoadModels returns the load models created by the library for the pin and edge passed to this function. If the load models are identical, then the load model pointers returned shall have the same value.

The legal pin edges for this call are as follows:

- DCM_RisingEdge
- DCM_FallingEdge

- DCM_OneToZ
- DCM_ZtoOne
- DCM_ZeroToZ
- DCM_ZtoZero

10.21.10 Obtaining parasitic networks

To accommodate on-chip process variations, the application *can* return two parasitic networks for each net to the library via pointers, one with element values for the minimum and another with values for the maximum of the on-chip process uncertainty (see 10.23.11). These two networks shall be identical in all other respects, including the order in which their elements and nodes appear. If on-chip process variation is not modeled, then these networks shall be completely identical, and the parasitic network pointers returned by the application shall be the same.

The following functions are defined for the library to use in obtaining parasitic networks.

10.21.10.1 appGetParasiticNetworksByPin

Table 235 provides information on appGetParasiticNetworksByPin.

Table 235—appGetParasiticNetworksByPin

Function name	appGetParasiticNetworksByPin
Arguments	Pin pointer
Result	Parasitic networks
Standard Structure fields	CellName, cellData (timing), pathData (timing-pin-specific), sinkEdge, calcMode
DCL syntax	EXTERNAL (appGetParasiticNetworksByPin) : passed(pin: pinPointer) result (parasiticSubnet: minParasitics, maxParasitics);
C syntax	typedef struct { const DCM_ParasiticSubnet *minParasitics, *maxParasitics; } T_ParasiticNetworks; int appGetParasiticNetworksByPin(const DCM_STD_STRUCT *std_struct, T_ParasiticNetworks *rtn, const DCM_PIN pinPointer);

This requests the application gather up the parasitic elements for a net connected to a pin. The application shall return the parasitic-element structures that represent the nonreduced parasitic networks for that net back to the library.

A bad return code from this call shall not be interpreted by the library as indicating parasitic information for the net is not available. In such cases, the library shall then request the application provide the load and interconnect models needed to perform the analysis associated with the net.

10.21.10.2 appGetParasiticNetworksByName

Table 236 provides information on appGetParasiticNetworksByName.

Table 236—appGetParasiticNetworksByName

Function name	appGetParasiticNetworksByName
Arguments	Pin name
Result	Parasitic networks
Standard Structure fields	CellName, block, cellData (timing), sinkEdge, calcMode
DCL syntax	EXTERNAL (appGetParasiticNetworksByName): passed(string: pinName) result(parasiticSubnet: minParasitics, maxParasitics);
C syntax	typedef struct { const DCM_ParasiticSubnet *minParasitics, *maxParasitics; } T_ParasiticNetworks; int appGetParasiticNetworksByName(const DCM_STD_STRUCT *std_struct, T_ParasiticNetworks *rtn, const DCM_STRING pinName);

This allows the library to gather parasitic information about pins that are not on the arc being analyzed (e.g., this call can be used to get parasitics for unbuffered outputs or arcs that pass through a cell unbuffered).

An *unbuffered output* is a configuration where a signal present at the input of a cell drives more than one output and the load on each such output affects the delay to the other outputs. This forces the calculation of the delay from the input to an output to take into account all the loads on the other unbuffered output pins, as well as the output of interest.

appGetParasiticNetworksByName returns the parasitic networks for the net connected to the named pin on the block supplied in the *Standard Structure*. The application shall return the parasitic-element structures that represent the nonreduced parasitic networks for the net connected to that pin back to the library. It shall be an error for the library to request the parasitic networks for a named pin that does not exist on the cell.

A bad return code from this call shall not be interpreted by the library as indicating parasitic information for the net is not available. In such cases, the library shall then request the application provide the load and interconnect models needed to perform the analysis associated with the net.

10.21.11 Persistent storage of load and interconnect models

Interconnect and load models contain data proprietary to the library. The application can have a need to save these models to disk and then restore them within a session or from one session to another. Because the details of the data in these models are proprietary to the library, the models cannot be saved in a standard format, such as the reduced models included in SPEF. SPEF only supports poles and zeros for interconnect and PI models for loading.

10.21.11.1 Application save and restore

The most efficient method for rapid access is to have the application store model information for the library, so these data are correlated with the design information. To do this, the library shall convert its proprietary data structures to a form that can be streamed to disk (at the request of the application). In situations where the application has previously requested the library passivate the load or interconnect models, the application shall request the library retrieve these models before they are used in calculation sequences.

To accommodate on-chip process variations, the application *can* pass two load or interconnect models to the library using pointers, one for the minimum and another for the maximum of the on-chip process uncertainty (see 10.23.11). The library shall passivate both models in each pair, storing them in one

collection of bytes. During subsequent retrieval, these models shall be restored by the library and returned to the application as a pair.

If on-chip process variation is not modeled, the models in each pair shall be identical. The load- or interconnect-model pointers for each pair shall then have the same value.

10.21.11.1.1 dpcmPassivateLoadModels

Table 237 provides information on dpcmPassivateLoadModels.

Table 237—dpcmPassivateLoadModels

Function name	dpcmPassivateLoadModels
Arguments	Load models
Result	Passivated load models, Size
Standard Structure fields	CellName, block, cellData (timing), pathData (timing-pin-specific), toPoint
DCL syntax	<pre>EXPOSE(dpcmPassivateLoadModels): passed(void: minLoadModel, maxLoadModel) result(void: passivatedLoadModels; int: size);</pre>
C syntax	<pre>typedef struct { DCM_STRUCT *passivatedLoadModels; DCM_INTEGER size; } T_PassivatedLoadModels; int dpcmPassivateLoadModels(const DCM_STD_STRUCT *std_struct, T_PassivatedLoadModels *rtn, const DCM_STRUCT *minLoadModel, const DCM_STRUCT *maxLoadModel);</pre>

This converts a pair of library load models to a contiguous collection of bytes that can be persistently stored. *dpcmPassivateLoadModel* returns a pointer to the collection of bytes and an integer value indicating the number of bytes back to the application.

10.21.11.1.2 dpcmPassivateInterconnectModels

Table 238 provides information on dpcmPassivateInterconnectModels.

Table 238—dpcmPassivateInterconnectModels

Function name	dpcmPassivateInterconnectModels
Arguments	Interconnect models
Result	Passivated interconnect models, Size
Standard Structure fields	CellName, block, cellData (timing), pathData (timing-pin-specific), fromPoint, toPoint
DCL syntax	<pre>EXPOSE(dpcmPassivateInterconnectModels): passed(void: interconnectModels) result(void: passivatedInterconnectModels; int: size);</pre>

C syntax	<pre>typedef struct { DCM_STRUCT *passivatedInterconnectModels; int size; } T_PassivatedInterconnectModels; int dpcmPassivateInterconnectModels(const DCM_STD_STRUCT *std_struct, T_PassivatedInterconnectModels *rtn, const DCM_STRUCT *minInterconnectModel, const DCM_STRUCT *maxInterconnectModel);</pre>

This converts a pair of library interconnect models to a contiguous collection of bytes that can be persistently stored. *dpcmPassivateInterconnectModel* returns a pointer to the first byte in the collection and an integer value indicating the number of bytes back to the application.

10.21.11.1.3 dpcmRestoreLoadModels

Table 239 provides information on dpcmRestoreLoadModels.

Table 239—dpcmRestoreLoadModels

Function name	dpcmRestoreLoadModels
Arguments	Library passivated load models, Size of passivated load models
Result	Library load models
Standard Structure fields	CellName, block, cellData (timing), pathData (timing-pin-specific), toPoint
DCL syntax	<pre>EXPOSE(dpcmRestoreLoadModels): passed(void: passivatedLoadModels ; int: size) result(void: minLoadModel, maxLoadModel);</pre>
C syntax	<pre>typedef struct { const DCM_STRUCT *minLoadModel, *maxLoadModel; } T_LoadModels; int dpcmRestoreLoadModel(const DCM_STD_STRUCT *std_struct, T_LoadModels *rtn, void *passivatedLoadModels, DCM_INTEGER size);</pre>

This converts a pair of library passivated load models to the *DCM_STRUCT* format used by the library during calculations made using the models. This call takes as arguments a pointer to the first byte in the previously passivated load models and an integer value indicating the number of bytes those models occupy. The application shall manage the memory containing the passivated load models.

10.21.11.1.4 dpcmRestoreInterconnectModels

Table 240 provides information on dpcmRestoreInterconnectModels.

Table 240—dpcmRestoreInterconnectModels

Function name	dpcmRestoreInterconnectModels
Arguments	Passivated library interconnect models, Size of passivated models
Result	Library interconnect models
Standard Structure fields	CellName, block, cellData (timing), pathData (timing-pin-specific), fromPoint, toPoint
DCL syntax	<pre>EXPOSE(dpcmRestoreInterconnectModels): passed(void: passivatedInterconnectModels; int: size) result(void: interconnectModels);</pre>

C syntax	<pre>typedef struct { const DCM_STRUCT *minInterconnectModel, *maxInterconnectModel; } T_InterconnectModels; int dpcmRestoredInterconnectModels(const DCM_STD_STRUCT *std_struct, T_InterconnectModels *rtn, void *passivatedInterconnectModels, DCM_INTEGER size);</pre>

This converts a pair of library passivated interconnect models to the *DCM_STRUCT* format used by the library during calculations made using the models. This call takes as arguments a pointer to the first byte in the previously passivated interconnect models and an integer value indicating the number of bytes those models occupy. The application shall manage the memory containing the passivated interconnect models.

10.21.12 Calculating effective capacitances and driving resistances

The following functions are defined for calculation of the effective loading capacitances on the output of a cell using a PI model for the loading network. The following functions shall be used to perform such calculations using arbitrary load models.

10.21.12.1 appGetCeff

Table 241 provides information on appGetCeff.

Table 241—appGetCeff

Function name	appGetCeff
Arguments	Delay calculation function pointer, Slew calculation function pointer, Min load model, Max load model
Result	Late Ceffective, Early Ceffective
Standard Structure fields	CellName, calcMode, block, sourceEdge, sinkEdge, pathData(timing-arc-specific), cellData(timing), fromPoint, toPoint, slew->early, slew->late, processVariation
DCL syntax	<pre>forward calc(stdCeffDelaySlewEq): passed(int: propagationMode; double: loadCap) result(double: value); EXTERNAL(appGetCeff): passed(stdCeffDelaySlewEq(): delayEq, slewEq; void: minLoadModel, maxLoadModel) result(double: lateCeffective, earlyCeffective);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE lateCeffective, earlyCeffective; } T_Ceffective; int appGetCeff(const DCM_STD_STRUCT *std_struct, T_Ceffective *rtn, DCM_GeneralFunction delayEq, DCM_GeneralFunction slewEq, const DCM_STRUCT *minLoadModel, const DCM_STRUCT *maxLoadModel);</pre>

This returns the effective capacitances seen by the passed driver pin (*toPoint*) back to the library.

If on-chip process variation for the driven net is not modeled, the pointers to the minimum and maximum load models shall be the same.

To obtain effective capacitances, the application shall call *dpcmCalcCeff*, passing the arguments declared above to *dpcmCalcCeff* without altering them.

10.21.12.2 dpcmCalcCeff

Table 242 provides information on dpcmCalcCeff.

Table 242—dpcmCalcCeff

Function name	dpcmCalcCeff
Arguments	Delay calculation function pointer, Slew calculation function pointer, Min load model, Max load model
Result	Late Ceffective, Early Ceffective
Standard Structure fields	CellName, calcMode, block, sourceEdge, sinkEdge, pathData(timing-arc-specific), cellData (timing), fromPoint, toPoint, slew->early, slew->late, processVariation
DCL syntax	<pre>forward calc(stdCeffDelaySlewEq): passed(int: propagationMode; double: loadCap) result(double: value); EXPOSE(dpcmCalcCeff): passed(stdCeffDelaySlewEq(): delayEq, slewEq; void: minLoadModel, maxLoadModel) result(double: lateCeffective, earlyCeffective);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE lateCeffective, earlyCeffective; } T_Ceffective; int dpcmCalcCeff (const DCM_STD_STRUCT *std_struct, T_Ceffective *rtn, DCM_GeneralFunction delayEq, DCM_GeneralFunction slewEq, const DCM_STRUCT *minLoadModel, const DCM_STRUCT *maxLoadModel);</pre>

This returns the effective capacitances seen by the passed driver pin (*toPoint*) back to the application.

The library shall use the *processVariation* field in the Standard Structure to determine how to use the load models in computing the early and late effective capacitances. If on-chip process variation for the driven net is not modeled, then the pointers to the minimum and maximum load models shall be the same.

10.21.12.3 dpcmCalcSteadyStateResistanceRange

Table 243 provides information on dpcmCalcSteadyStateResistanceRange.

Table 243—dpcmCalcSteadyStateResistanceRange

Function name	dpcmCalcSteadyStateResistanceRange
Arguments	Pin pointer
Result	Maximum resistance, Minimum resistance
Standard Structure fields	CellName, pathData(timing-pin-specific), cellData (timing)
DCL syntax	<pre>typedef (resistanceRange): result(double var: maxResistance, minResistance); EXPOSE(dpcmCalcSteadyStateResistanceRange): passed(pin: outputPin) result(resistanceRange: range);</pre>

C syntax	<pre>typedef struct { DCM_DOUBLE maxResistance, minResistance; } DCM_ResistanceRange; typedef struct { DCM_ResistanceRange range; } T_ResistanceRange; int dpcmCalcSteadyStateResistanceRange (const DCM_STD_STRUCT *std_struct, T_ResistanceRange *rtn, DCM_PIN outputPin);</pre>

In some situations, the application needs to assemble a parasitic network for the computation of delay involving both aggressors and victims. Some aggressors can be in an unknown but steady state, or a logical one, logical zero, or high-impedance state. For each of these possible states, the output of the cell has an equivalent resistance to either the power rail or *ground*. For the purposes of this call, a resistance to either the power rail or *ground* is considered an equivalent resistance to *ground*. For the calculation of delay to proceed and to account for the effects of drivers in an unknown steady state, the library needs to supply the maximum and minimum possible resistances the cell is capable of presenting to the output. Typically, the maximum resistance is used in the calculation of early mode delay and the minimum resistance is used to calculate the late mode delay.

dpcmCalcSteadyStateResistanceRange returns the maximum and minimum driving resistances possible for a cell that is not transitioning, regardless of the states where those resistances can occur. The pin handle for the pin of the requested resistances is the arguments to *dpcmCalcSteadyStateResistanceRange*.

10.21.12.4 dpcmCalcTristateResistanceRange

Table 244 provides information on *dpcmCalcTristateResistanceRange*.

Table 244—*dpcmCalcTristateResistanceRange*

Function name	<i>dpcmCalcTristateResistanceRange</i>
Arguments	The output pin
Result	Resistance range
Standard Structure fields	CellName, cellData (noise-specific), pathData (noise-pin-specific)
DCL syntax	<pre>typedef (resistanceRange): result (double var: maxResistance, minResistance); EXPOSE (dpcmCalcTristateResistanceRange): passed (pin: outputPin) result (resistanceRange: range);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE maxResistance, minResistance; } DCM_ResistanceRange; typedef struct { DCM_ResistanceRange range; } T_ResistanceRange; int dpcmCalcTristateResistanceRange (const DCM_STD_STRUCT *std_struct, T_ResistanceRange *rtn, DCM_PIN outputPin);</pre>

This function returns the maximum and minimum driving resistances possible for a cell that is in the high-impedance state. For the purposes of this call, a resistance to either the power rail or ground is considered an equivalent resistance to ground.

10.21.12.5 appSetCeff

Table 245 provides information on appSetCeff.

Table 245—appSetCeff

Function name	appSetCeff
Arguments	Early Ceffctive, Late Ceffctive
Result	None
Standard Structure fields	CellName, calcMode, block, sinkEdge, toPoint
DCL syntax	EXTERNAL(appSetCeff) : passed(double: earlyCeff, lateCeff) result(int: ignore);
C syntax	typedef struct { DCM_INTEGER ignore; } T_Ignore; int appSetCeff (const DCM_STD_STRUCT *std_struct, T_Ignore *rtn, DCM_DOUBLE earlyCeff, DCM_DOUBLE lateCeff);

This function sends to the application the effective load capacitances seen by the driver pin and for the edge identified by the *toPoint* and *sinkEdge* fields, respectively, in the *Standard Structure*.

The library shall call *appSetCeff* during a cell-arc delay computation initiated by the application during which these data are calculated. This call shall be made before the library function that calculates these delays exits.

10.21.13 Parasitic estimation

An interconnect network typically consists of a system of passive parasitic elements. This network is evaluated to determine the signal's delay and transition rate degradation as it reaches a terminal on the network. Accurate parasitic element evaluation is required for accurate delay and slew prediction. The system shall, however, accommodate those application situations where the complete parasitic information is not yet fully understood.

At the earliest stages of design, where no placement or wiring information exists, the typical approach is to use a statistical model, such as the wire load model. This gives crude approximations when there is no other information available. When simple statistical models, such as wire load models, fail to produce the accuracy needed, the library needs to be capable of more advanced calculations, which can vary from simple approximations of the global parasitic elements to more detailed evaluations of each parasitic element in the network. To accomplish this goal for a wide variety of situations, the library can perform calculations based on a wire load model or use models of greater detail based on discrete parasitic elements when more accuracy is necessary.

The application can choose between the two methods and can also choose the amount of precision contained in each element to be evaluated. It can, for example, estimate the routing length and the amount of adjacent wire. It can ask the library to calculate the capacitance and resistance on these estimated elements. It can also determine the individual resistance and capacitances for each individual wire segment producing a much more accurate result.

A collection of functions is needed so the library and application can interact on the calculation of parasitic elements. One function calculates the capacitive elements, one calculates the resistive elements, and others allow for the exchange of technology layer information.

Typically, semiconductor devices are constructed as a sandwich of many different types of layers. Each layer has a general purpose that shall be described to the application so clear and concise communication can take place between the library and application. The following is a brief description of this collection of calls and their basic functions:

- `dpcmGetLayerArray` transmits process layer names and types to the application. This data is transmitted to the application in a pair of synchronous arrays, one containing the name of each layers and the other containing their respective types.
- `dpcmCalcCouplingCapacitance` computes the capacitance between two conductors, even if these conductors are on different layers.
- `dpcmCalcSegmentResistance` calculates the resistance of an interconnect segment.

10.21.13.1 Shapes

To utilize the library in the process of capacitance calculation, the application needs to be able to describe the shapes of the two objects (between which the application needs the capacitance). These shapes shall be flexible enough to allow the application to make simplifying assumptions, when necessary, or to supply full detail. The only difference in the quality of the capacitance value calculated is the quality of the arguments supplied to the call. The shape description is designed to allow the transfer of a varying amounts of detail from simple rectangles to full polygons.

The shape information is contained in an array of coordinates. Coordinates are paired values where even elements in the array represent the *x* coordinates and odd elements represent the *y* coordinates. The coordinates shall be ordered in a clockwise or counterclockwise sequence of points, which describe the enclosed polygon. The coordinate points shall be an absolute distance from some reference point, such as a corner of the chip or the corner cell, or relative to the first pair of coordinates on the aggressor shape.

Special simplifying shapes can be recognized by the number of elements in the array. For example, simple rectangles and circles do not have to supply all the points of the shape.

Rectangles can be represented as complete polygons or simplified figures. The simplified rectangle shall have four elements. The first pair of elements (0 and 1) represent the coordinates of one corner. The second pair of elements (2 and 3) represent the corner diagonally opposite from the first coordinate pair.

Simplified rectangles can only be used when the coordinates of the rectangles are either parallel or perpendicular to the coordinates of the chip.

10.21.13.1.1 `dpcmCalcCouplingCapacitance`

Table 246 provides information on `dpcmCalcCouplingCapacitance`.

Table 246—`dpcmCalcCouplingCapacitance`

Function name	<code>dpcmCalcCouplingCapacitance</code>
Arguments	Aggressor XY coordinates, Aggressor layer index, Victim XY coordinates, Victim layer index
Result	Coupling capacitance
Standard Structure fields	<code>calcMode</code>
DCL syntax	<pre>EXPOSE(dpcmCalcCouplingCapacitance): passed(double[*]: aggressorCoordinates, victimCoordinates; int: aggressorLayerIndex, victimLayerIndex) result(double: couplingCapacitance);</pre>

C syntax	<pre>typedef struct { DCM_DOUBLE couplingCapacitance; } T_CouplingCapacitance; int dpcmCalcCouplingCapacitance(const DCM_STD_STRUCT *std_struct, T_CouplingCapacitance *rtn, DCM_DOUBLE_ARRAY *aggressorCoordinates, DCM_DOUBLE_ARRAY *victimCoordinates, DCM_INTEGER aggressorLayerIndex, DCM_INTEGER victimLayerIndex);</pre>

This takes the aggressor's shape, the aggressor's layer, the victim's shape, and the victim's layer as arguments. *dpcmCalcCouplingCapacitance* calculates the capacitance between these two conductors. The application needs to help ensure there are no conductive obstructions between the aggressor and the victim. This call is not intended to calculate shielded capacitances.

10.21.13.1.2 dpcmCalcSubstrateCapacitance

Table 247 provides information on *dpcmCalcSubstrateCapacitance*.

Table 247—dpcmCalcSubstrateCapacitance

Function name	dpcmCalcSubstrateCapacitance
Arguments	Aggressor XY coordinates, Aggressor layer index
Result	Substrate capacitance
Standard Structure fields	calcMode
DCL syntax	<pre>EXPOSE (dpcmCalcSubstrateCapacitance) : passed(double[*]: aggressorCoordinates; int: aggressorLayerIndex) result(double: substrateCapacitance);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE substrateCapacitance; } T_SubstrateCapacitance; int dpcmCalcSubstrateCapacitance(const DCM_STD_STRUCT *std_struct, T_SubstrateCapacitance *rtn, DCM_DOUBLE_ARRAY *aggressorCoordinates, DCM_INTEGER aggressorLayerIndex);</pre>

This computes the plate and fringe capacitances between a conductor on any given layer and the substrate. *dpcmCalcSubstrateCapacitance* computes both plate and fringe capacitances as a single lumped value. The application shall only call *dpcmCalcSubstrateCapacitance* for those portions of the conductor with a clear “view” of the substrate. *dpcmCalcSubstrateCapacitance* takes as arguments the aggressor's shape and the aggressor's layer index.

10.21.13.1.3 dpcmCalcSegmentResistance

Table 248 provides information on *dpcmCalcSegmentResistance*.

Table 248—dpcmCalcSegmentResistance

Function name	dpcmCalcSegmentResistance
Arguments	Metal shape, Layer index, Entrance face, Exit face
Result	Resistance
Standard Structure fields	calcMode
DCL syntax	<pre>EXPOSE(dpcmCalcSegmentResistance): passed(double[*]: shape; int: layerIndex, injectingFace, exitFace) result(double: resistance);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE resistance; } T_SegmentResistance; int dpcmCalcSegmentResistance(const DCM_STD_STRUCT *std_struct, T_SegmentResistance *rtn, DCM_DOUBLE_ARRAY *shape, DCM_INTEGER layerIndex, DCM_INTEGER injectingFace, DCM_INTEGER exitFace);</pre>

This takes the shape of the wire, the layer where the wire is routed, the side of wire where the current is injected, and the side where the current exits and calculates the resistance. The shape follows the same semantics as defined for *dpcmCouplingCapacitance*. The *layerIndex* is the index of the layer where the wire or via is. *injectingFace* and *exitFace* represent where the edge current enters and exits the shape, respectively.

- For fully elaborated polygons, these indicate the first set of coordinates in the shapes matrix representing the start of the edge of the polygon where the current enters and exits. The values of *injectingFace* and *exitFace* shall be even integers in these cases.
- For rectangles, *injectingFace* indicates whether the current enters the rectangle along the *x* or *y* coordinate. *exitFace* indicates whether the current exits the rectangle on the *x* or *y* coordinate. A value of zero (0) for *injectingFace* or *exitFace* indicates the *x* axis; a value of one (1) indicates the *y* axis.
- For the part of the via within insulation layers, the current is assumed to enter the top or bottom and exit the opposite face of the via. The library knows this implicitly because of the layer index, and shall ignore the values of *injectingFace* and *exitFace* in these situations.

10.21.13.2 Layer definitions

Manufacturing layer types describe a layer's purpose: the bulk wafer, the interconnecting circuitry within a cell, insulating layers and the interconnect between cells. Manufacturing layer types are defined in Table 249.

Table 249—Manufacturing layer type values

Layer type	Value	Definition
POLY	3	Polysilicon interconnect layer.
DEVICE	2	The layers used to develop the transistor within a cell.
INSULATION	1	Separation layer where vias can penetrate.
INTERCONNECT	0	Metal interconnect layers

10.21.13.2.1 dpcmGetLayerArray

Table 250 provides information on dpcmGetLayerArray.

Table 250—dpcmGetLayerArray

Function name	dpcmGetLayerArray
Arguments	Number of manufacturing layers
Result	Array of layer names, Number of manufacturing layers
Standard Structure fields	None
DCL syntax	<pre>EXPOSE(dpcmGetLayerArray): passed(integer: numberOfMetalLayers) result(string[*]: layerNames; int[*]: layerTypes);</pre>
C syntax	<pre>typedef struct { DCM_STRING_ARRAY *layernames; DCM_INTEGER_ARRAY *layertypes; } T_LevelArray; int dpcmGetLayerArray(const DCM_STD_STRUCT *std_struct, T_LevelArray *rtn, DCM_INTEGER numberOfMetalLayers);</pre>

This returns an ordered array of layer names and layer types back to the application. The order begins with the zeroth element, which is the first layer the library chooses for calculating capacitance values, starting from the layer closest to the device layer and building up. The layer type array is synchronized to the layer names array, where the *i*th element of the layer type array contains the value for the *i*th element of the *layerNames* array. The library shall store the value of the argument *numberOfMetalLayers* for future reference.

NOTE—For other calls the application makes that use the layer index, such as *dpcmCalcCouplingCapacitance*, the index is relative to the last call the application made to this function.

10.21.13.2.2 dpcmGetRuleUnitToMeters

Table 251 provides information on dpcmGetRuleUnitToMeters.

Table 251—dpcmGetRuleUnitToMeters

Function name	dpcmGetRuleUnitToMeters
Arguments	None
Result	Scale factor exponent
Standard Structure fields	None
DCL syntax	<pre>EXPOSE(dpcmGetRuleUnitToMeters): result(int: scaleFactorPower);</pre>
C syntax	<pre>typedef struct { INTEGER scaleFactorPower; } T_RuleUnitToMeters; int dpcmGetRuleUnitToMeters (const DCM_STD_STRUCT *std_struct, T_RuleUnitToMeters *rtn);</pre>

This returns the basic distance units the library assumes, expressed as an integer power of 10, back to the application. The value 10 *scaleFactorPower* when multiplied by a distance value, changes the distance value's units to meters.

Example

The following example shows how a DPCM indicates the distance unit in nanometers:

```
EXPOSE calc(dpcmGetRuleUnitToMeters): result(integer: -9);
```

10.21.13.2.3 dpcmGetRuleUnitToAmps

Table 252 provides information on dpcmGetRuleUnitToAmps.

Table 252—dpcmGetRuleUnitToAmps

Function name	dpcmGetRuleUnitToAmps
Arguments	None
Result	Scale factor power
Standard Structure fields	None
DCL syntax	EXPOSE(dpcmGetRuleUnitToAmps): result(int: scaleFactorPower);
C syntax	<pre>typedef struct { INTEGER scaleFactorPower; } T_RuleUnitToAmps; int dpcmGetRuleUnitToAmps (const DCM_STD_STRUCT *std_struct, T_RuleUnitToAmps *rtn);</pre>

This returns the basic current units the library assumes, expressed as an integer power of 10, back to the application. The value 10 *scaleFactorPower*, when multiplied by a current value, changes the resistance value's units to amps.

10.21.14 Threshold voltages

This subclause includes two items related to threshold voltages.

10.21.14.1 appGetDriverThresholds

Table 253 provides information on appGetDriverThresholds.

Table 253—appGetDriverThresholds

Function name	appGetDriverThresholds
Arguments	inputPin
Result	driver thresholds
Standard Structure fields	CellName, calcMode, block, cellData (timing), pathData(timing-pin specific)
DCL syntax	<pre>typedef(driverThresholdStruct): passed(pin: inputPin) result(double: vol, voh, lowerTransitionThreshold, upperTransitionThreshold, riseSwitchLevel, fallSwitchLevel); EXTERNAL(appGetDriverThresholds): result(driverThresholdStruct [*]: driverThresholds);</pre>

C syntax	<pre> typedef struct { DCM_DOUBLE vol; DCM_DOUBLE voh; DCM_DOUBLE lowerTransitionThreshold; DCM_DOUBLE upperTransitionThreshold; DCM_DOUBLE riseSwitchLevel; DCM_DOUBLE fallSwitchLevel; } DCM_DriverThresholds; typedef DCM_DriverThresholds *DCM_DriverThresholds_ARRAY; typedef struct { DCM_DriverThresholds_ARRAY *driverThresholds; } T_DriverThresholds; int appGetDriverThresholds(const DCM_STD_STRUCT *std_struct, T_DriverThresholds *rtn, const DCM_PIN inputPin); </pre>

This function returns the voltage limits and transition and switching thresholds for all of the drivers of the net connected to the pin identified by the *inputPin* argument. The application shall call the function *dpcmGetThresholds* to obtain this information.

- All values returned by this function shall be voltages referenced to ground.
- If the pin for which thresholds are requested is bidirectional, the *pathData* field shall be used to determine whether thresholds for the input or the output part of the pin are returned. If the *pathData* field is set to zero (0), then thresholds suitable for use for both parts of this pin shall be returned.

The order of the drivers in the returned array shall be considered to be arbitrary. No assumptions regarding this order shall be made by either the application or by the library.

If any driver of a pin is in a different technology than the cell to which that pin belongs, this function shall not be used for the pin. If the application detects such a call, then it shall return an error to the library.

10.21.15 Obtaining aggressor window overlaps

This subclause contains specifications for a set of functions used by the library to obtain aggressor timing window overlaps from the application.

In calculating overlap-window data, the application shall use interaction windows provided previously by the library. These windows shall indicate intervals of time in which activity at aggressor drivers are of interest to the library. Each interaction window shall be defined by early and late times that shall be relative to an edge time at the pin identified by the *fromPoint* and *sourceEdge* fields in the *Standard Structure*. There shall be separate interaction windows relative to the early and late edge times at the *fromPoint* pin.

10.21.15.1 appGetAggressorOverlapWindows

Table 254 provides information on *appGetAggressorOverlapWindows*.

Table 254—appGetAggressorOverlapWindows

Function name	<i>appGetAggressorOverlapWindows</i>
Arguments	early and late overlap windows arrays
Result	None
Standard Structure fields	CellName, calcMode, fromPoint, toPoint, block, sourceEdge, sinkEdge, processVariation

DCL syntax	<pre> typedef (overlapWindow): result(int var: overlapPresent double var: earlyTime, lateTime, earlySlew, lateSlew, earlyDriverResistance, lateDriverResistance; void var: earlyXWF, lateXWF); EXTERNAL(appGetAggressorOverlapWindows): passed(var overlapWindow transient[*]: earlyOverlapArray, lateOverlapArray) result(int: ignore); </pre>
C syntax	<pre> typedef struct { DCM_INTEGER overlapPresent; DCM_DOUBLE earlyTime, lateTime, earlySlew, lateSlew; DCM_DOUBLE earlyDriverResistance, lateDriverResistance; const DCM_STRUCT *earlyXWF, *lateXWF; } DCT_OverlapWindow; typedef DCT_OverlapWindow *DCM_OverlapWindow_ARRAY; typedef struct { DCM_INTEGER ignore; } T_Ignore; int appGetAggressorOverlapWindows (const DCM_STD_STRUCT *std_struct, T_Ignore *rtn, DCM_OverlapWindow_ARRAY *earlyOverlapArray, *lateOverlapArray); </pre>

This function can be called by the library to obtain arrays of arrival-window information about the aggressor drivers coupled to a net for which propagation-arc calculation (delay, slew, etc.) is being done. The propagation arc shall be defined by the *fromPoint*, *toPoint*, *sourceEdge*, and *sinkEdge* fields in the *Standard Structure*. The function can also be used during propagation-arc calculation for a cell that drives such a net.

There shall be one *overlap window* in each array for each aggressor driver. The order of these windows shall be the same as the order in which the corresponding aggressor drivers first appear in the parasitic network for the net as supplied by the application. In calculating these windows, the application shall use interaction windows provided previously by the library via the function *appSetAggressorInteractWindows* (see 10.21.15.2).

All data associated with this function shall be in the units of the library technology that calls the function.

To avoid memory-management overhead for individual overlap windows, the library shall allocate arrays of transient overlap-window structures (see 7.4.5.1.1), passing them to the application via *earlyOverlapArray* and *lateOverlapArray*. The application shall fill in the contents of these structures but shall neither claim nor disclaim the arrays. Because the structures are transient, the application shall not attempt to either claim or disclaim them.

The overlap windows returned by the application shall define intervals of activity at each aggressor driver occurring within each corresponding interaction window. Each overlap window shall be defined by an early and a late time, which shall again be relative to an edge time at the *fromPoint* pin. Separate overlap windows for the early and late edge times at that pin shall be returned for each aggressor driver.

Each overlap window shall also contain an early and a late slew that define the range of possible slew values for aggressor transitions that can occur within the window. If available to the application, pointers to XWF structures corresponding to these slews shall also be supplied. If no such XWF data are available or these data were provided by a library technology other than that calling this function, these pointers shall be set to zero (0).

If and only if XWF structures are not supplied, an overlap window shall also contain two output resistances for the associated aggressor driver, one corresponding to each slew provided. If there is no aggressor activity within an interaction window, these values shall be set to the quiet output resistances of the driver. The application shall obtain these resistances by calling *dpcmCalcSteadyStateResistanceRange* (see 10.21.12.3). The application shall set the *earlyDriverResistance* field in the overlap-window structure to the maximum resistance returned by that function and the *lateDriverResistance* field to the minimum resistance returned.

If the aggressor is active but it is in a different library technology, output resistances obtained via *dpcmCalcOutputResistances* (see 10.21.15.6) shall be used. Whenever XWF structures are supplied, these resistance values shall be set to zero and shall be ignored by the library.

If aggressor activity can occur within an interaction window, the *overlapPresent* field in the corresponding overlap-window structure shall be set to one (1). Otherwise, this field shall be set to zero (0) and the remaining fields, other than the output resistance, shall not be used by the library.

Each early overlap window shall represent an interval in which an edge can occur at the associated aggressor that has the same polarity as the edge identified by the *sinkEdge* field in the *Standard Structure*.

Each late overlap window shall represent an interval in which an edge can occur at the associated aggressor that has the opposite polarity from the edge identified by the *sinkEdge* field in the *Standard Structure*.

If no interaction-window data have been provided by the library for the propagation arc for which *appGetAggressorOverlapWindows* is called, the application shall return default overlap windows of its own choosing. The contents of these default windows are dependent on the algorithm used by the application to account for aggressor activity at a design level. If the library provides a zero-valued pointer for an individual early or late interaction window, the application shall respond as appropriate for the algorithm. That algorithm shall not be mandated in this standard.

If the library provides a zero-valued pointer for either an early or a late overlap window, the application shall omit the calculation of data for that window. If the pointer to either the early or the late overlap-window array has a value of zero (0), the application shall omit all early or all late overlap-window calculation, respectively, for the arc.

10.21.15.2 appSetAggressorInteractWindows

Table 255 provides information on *appSetAggressorInteractWindows*.

Table 255—appSetAggressorInteractWindows

Function name	<i>appSetAggressorInteractWindows</i>
Arguments	Early and late interaction-window arrays
Result	None
Standard Structure fields	CellName, calcMode, processVariation, fromPoint, toPoint, block, sourceEdge, sinkEdge
DCL syntax	<pre>typedef (timeRange) : result(double var: earlyTime, lateTime); EXTERNAL (appSetAggressorInteractWindows) : passed(timeRange[*]: earlyInteractArray, lateInteractArray) result(int: ignore);</pre>

C syntax	<pre> typedef struct { DCM_DOUBLE earlyTime, lateTime; } DCM_TimeRange; typedef DCM_TimeRange *DCM_TimeRange_ARRAY; typedef struct { DCM_INTEGER ignore; } T_Ignore; int appSetAggressorInteractWindows(const DCM_STD_STRUCT *std_struct, T_Ignore *rtn, const DCM_TimeRange_ARRAY *earlyInteractArray, const DCM_TimeRange_ARRAY *lateInteractArray); </pre>

The library shall use this function to store arrays of interaction windows, one window in each array for each aggressor driver, in the application. This action shall occur only if overlap windows are needed during subsequent calculations by the library.

The interaction-window data shall be in the units of the library technology that calls this function.

The library shall call this function only during delay computations initiated by the application. Two arrays of interaction windows, one corresponding to the early delay and one to the late delay calculated during a delay computation, shall be set. The order of the windows within each array shall be the same as the order in which the corresponding aggressor drivers first appear in the parasitic network for the net supplied by the application.

This function shall set interaction windows for the propagation arc defined by the *fromPoint*, *toPoint*, *sourceEdge*, and *sinkEdge* fields in the *Standard Structure*. The application shall record the passed arrays for subsequent use in calculating overlap windows at the request of the library (see 10.21.15.1). These windows shall indicate intervals of time in which activity at the aggressor drivers are of interest to the library. If the library provides a zero-valued pointer for an individual early or late interaction window, the application shall assume that no relevant activity can occur for the corresponding aggressor.

The library shall call this function during computation of delay values before the function that calculates these delays (generated from a DCL *DELAY* statement) exits. The *Standard Structure* shall remain unmodified during this operation.

10.21.15.3 Modeling the effect of propagated noise on delay

An aggressor or dc-coupled driver may have no full transitions that occur within an interaction window of interest to the library but may still propagate noise during that window. An application can choose to model the effect of propagated noise on delay, passing information about propagated noise to the library through *appGetOverlapNWFs*. This requires support for both the timing domain and the noise domain, and the application may need to switch between the two domains to iteratively resolve the interactions between them. If the application chooses not to model these effects, it is free to leave *appGetOverlapNWFs* and *appSetDriverInteractWindows* unimplemented.

10.21.15.4 *appGetOverlapNWFs*

Table 256 provides information on *appGetOverlapNWFs*.

Table 256—appGetOverlapNWFs

Function name	appGetOverlapNWFs
Arguments	Noise waveforms, Driver noise waveforms
Results	None
Standard StructureStandard Structure fields	CellName, calcMode, fromPoint, toPoint, block, sourceEdge, sinkEdge, processVariation
DCL syntax	<pre>typedef (overlapNWFs) : result(NWF[*] var: NWFs; PWFdriverModel var[*]: PWFs); EXTERNAL (appGetOverlapNwf) : passed (var overlapNWFs transient[*]: driverOverlapNWFs, aggressorOverlapNWFs) result (int: ignore);</pre>
C syntax	<pre>typedef struct { DCM_NWF_ARRAY *NWFs; DCM_PWFdriverModel_ARRAY *PWFs; } DCT_overlapNWFs; typedef DCT_overlapNWFs *DCT_overlapNWFs_ARRAY; int appGetOverlapNWFs(DCM_STD_STRUCT *std_struct, T_Ignore *rtn, DCT_overlapNWFs_ARRAY *driverOverlapNWFs, DCT_overlapNWFs_ARRAY *aggressorOverlapNWFs);</pre>

This function can be called by the library to obtain arrays of noise waveforms for dc-connected drivers and aggressor drivers coupled to a net for which propagation-arc calculation (delay, slew, etc.) is being done. The propagation arc shall be defined by the *fromPoint*, *toPoint*, *sourceEdge*, and *sinkEdge* fields in the *Standard Structure*. The function can also be used during propagation-arc calculation for a cell that drives such a net.

There shall be one overlap NWF in the driverOverlapNWFs array for each dc-connected driver. The order of these overlap NWFs shall be the same as the order in which the corresponding dc-connected drivers first appear in the parasitic network for the net as supplied by the application. In calculating these overlap NWFs, the application shall use interaction windows provided previously by the library via the function *appSetDriverInteractWindows* (see 10.21.15.5).

There shall be one overlap NWF in the aggressorOverlapNWFs array for each aggressor driver. The order of these overlap NWFs shall be the same as the order in which the corresponding aggressor drivers first appear in the parasitic network for the net as supplied by the application. In calculating these overlap NWFs, the application shall use interaction windows provided previously by the library via the function *appSetAggressorInteractWindows* (see 10.21.15.2).

All data associated with this function shall be in the units of the library technology that calls the function.

To avoid the overhead of memory allocation and deallocation, which involves claiming and disclaiming for individual overlap NWFs, the library shall provide the addresses of arrays of pointers to overlap NWF structures in its own temporary memory. These pointer arrays shall be *neither* claimed *nor* disclaimed by the application. The application can, however, call functions such as *dcm_getNumElements* to determine the characteristics of the arrays. The application shall place the requested overlap-NWF data into the structures so addressed. To indicate the memory used for these structures is of this temporary or transient sort, the name of the C struct type for an overlap NWF shall begin with the prefix *DCT_* (not *DCM_*).

The overlap NWFs returned by the application shall define intervals of activity at each dc-connected and aggressor driver occurring within each corresponding interaction window. Each overlap NWF *shall contain* offsets for the earliest and latest possible start times for the waveform, which shall be relative to an edge time at the *fromPoint* pin.

For drivers that are in the same technology as the *fromPoint* pin, a pointer to an NWF shall be returned in the overlap NWFs array, and the corresponding PWF *driver-model* pointer shall be set to zero (0).

For drivers that are in a different technology than the *fromPoint* pin, a pointer to a *PWF driver model* shall be returned in the overlap NWFs array, and the corresponding NWF pointer shall be set to zero (0). The *PWF driver model shall contain* both a piece-wise linear representation of the noise waveform and an output resistance for each point in the PWF (see Table 264).

If no interaction-window data have been provided by the library for the propagation arc for which *appGetOverlapNWFs* is called, the application shall return default overlap NWFs of its own choosing. The contents of these default overlap NWFs are dependent on the algorithm used by the application to account for aggressor *noise* activity at a design level. If the library provides a zero-valued pointer for an individual early or late interaction window, the application shall respond as appropriate for the algorithm. That algorithm shall not be mandated in this standard.

If the library provides a zero-valued pointer for an overlap NWF, then the application shall omit the calculation of data for that window.

10.21.15.5 appSetDriverInteractWindows

Table 257 provides information on *appSetDriverInteractWindows*.

Table 257—appSetDriverInteractWindows

Function name	<i>appSetDriverInteractWindows</i>
Arguments	Early and late interaction-window arrays
Result	None
Standard Structure fields	CellName, calcMode, processVariation, fromPoint, toPoint, block, sourceEdge, sinkEdge
DCL syntax	<pre>typedef (timeRange) : result (double var: earlyTime, lateTime); EXTERNAL (appSetDriverInteractWindows) : passed (timeRange[*]: earlyInteractArray, lateInteractArray) result (int: ignore);</pre>

C syntax	<pre> typedef struct { DCM_DOUBLE earlyTime, lateTime; } DCM_TimeRange; typedef DCM_TimeRange *DCM_TimeRange_ARRAY; typedef struct { DCM_INTEGER ignore; } T_Ignore; int appSetDriverInteractWindows (const DCM_STD_STRUCT *std_struct, T_Ignore *result, const DCM_TimeRange_ARRAY *earlyInteractArray, const DCM_TimeRange_ARRAY *lateInteractArray); </pre>

The library shall use this function to store arrays of *interaction windows*, one window in each array for each dc-connected driver, in the application. This action shall occur only if overlapNWFs are needed during subsequent calculations by the library.

The interaction-window data shall be in the units of the library technology that calls this function.

The library shall call this function only during delay computations initiated by the application. Two arrays of interaction windows, one corresponding to the early delay and one to the late delay calculated during a delay computation, shall be set. The order of the windows within each array shall be the same as the order in which the corresponding dc-connected drivers first appear in the parasitic network for the net supplied by the application.

This function shall set interaction windows for the propagation arc defined by the *fromPoint*, *toPoint*, *sourceEdge*, and *sinkEdge* fields in the *Standard Structure*. The application shall record the passed arrays for subsequent use in calculating overlapNWFs at the request of the library (see 10.21.15.4). These windows shall indicate intervals of time in which activity at the dc-connected drivers are of interest to the library. If the library provides a zero-valued pointer for an individual early or late interaction window, then the application shall assume that no relevant activity can occur for the corresponding dc-connected driver.

The library shall call this function during computation of delay values before the function that calculates these delays (generated from a *.DCL DELAY* statement) exits. The *Standard Structure* shall remain unmodified during this operation.

10.21.15.6 dpcmCalcOutputResistances

Table 258 provides information on dpcmCalcOutputResistances.

Table 258—dpcmCalcOutputResistances

Function name	dpcmCalcOutputResistances
Arguments	Output-pin pointer, Early and late slews, Early and late XWF data structures
Result	Early and late output resistances
Standard Structure fields	calcMode, processVariation, block, pathData (timing-pin-specific), cellData (timing)
DCL syntax	<pre> EXPOSE(dpcmCalcOutputResistances): passed(pin: outputPin; double: earlySlew, lateSlew; void: earlyXWF, lateXWF) result(double: earlyRout, lateRout); </pre>

C syntax	<pre>typedef struct { DCM_DOUBLE earlyRout; DCM_DOUBLE lateRout; } T_Rout; int dpcmCalcOutputResistances(const DCM_STD_STRUCT *std_struct, T_Rout *rtn, DCM_PIN outputPin, DCM_DOUBLE earlySlew, lateSlew, const DCM_STRUCT *earlyXWF, *lateXWF);</pre>

This function returns output resistances for the passed pin corresponding to the early and late slews and XWF structures provided. If the pin is not an output nor bidirectional, then an error shall occur.

The application shall call this function to obtain output resistances in circumstances for which the XWFs cannot be used directly. An example of this is when, during crosstalk analysis, overlap-window data for an aggressor driver in a technology different from that of a victim is requested by the library. The application can, at its discretion, record these values for use after an immediate need is met.

10.22 Noise analysis

The noise domain includes models and calculations for static noise analysis, including noise propagation and noise violation checking. For static noise analysis, test vectors are not required. Instead, worst-case noise values are calculated and propagated through a noise network of cones and pins. Cell models for use in a noise network are supplied by the library during model elaboration through the function *appNewNoiseCone*.

Noise may be present in a circuit through a variety of physical mechanisms, including parasitic coupling from aggressor nets to victim nets, power supply variations, charge sharing, and noise from external circuitry (provided by the application). Several predefined types of noise are included explicitly in the noise domain. In addition, the noise domain contains interface functions that allow the library to define additional types of noise and exchange information with the application for calculations and violations involving those noise types.

Although a library may consider multiple types of noise while doing a particular calculation, noise propagation is done using a composite NWF that represents the worst-case combination of all types of noise at a given pin. There are two primary functions for calculating how NWFs propagate. For cells, *dpcmCalcOutputNoise* uses the conditions on a cell's input pins (including NWFs, transitions, and constant values) as well as noise occurring within the cell (such as that due to charge sharing and leakage) to create sets of NWFs on the related output pins of the cell. Noise activity can also occur on other pins related to a particular output, such as power pins or other outputs connected through unbuffered output-to-output relationships; in which case, those pins shall also be considered in the related set for that output. For interconnect, *dpcmCalcInputNoise* uses conditions on dc-connect driver pins and coupled aggressor-driver pins to create a set of NWFs on a receiver input pin.

There are two interactions between the timing domain and the noise domain. First, static timing analysis can determine the lower and upper bounds (early and late values) within which all transitions on an aggressor net occur. This information from the timing domain can reduce pessimism in the noise domain. Due to on-chip process variation and other types of uncertainty, noise calculations shall be done assuming that within those bounds, transitions can occur at all possible times and in all cycles of operation. Thus, an NWF has a similar range of uncertainty, represented by bounds on the earliest and latest times at which the waveform can start. Library calculations involving multiple NWFs shall be done considering the worst possible alignment of these waveforms given those bounds. For asynchronous relationships between aggressors and victims, aggressor NWFs shall be considered to occur at all possible times.

In the second relationship, crosstalk noise occurring while a transition is propagating across a victim net can affect the propagation delay of that transition. This noise-induced delay shall be considered in the timing domain rather than the noise domain because the most significant crosstalk effects are caused by full transitions on aggressor nets. However, it is possible for an aggressor to be quiescent during all possible times that a victim transition can propagate. In that case, there might still be some noise propagating through the aggressor that may affect the victim transition. To accurately model this case, an application can use the results of calculations in the noise domain for noise-induced delay calculations in the timing domain. Iteration at the application level may be necessary to achieve convergence between the two domains.

The noise domain supports two types of violation checks. A library may enforce its own noise limits to help ensure that circuit elements function properly and reliably. In addition, an application may specify noise budgets to define a more conservative margin for some designs or to allow for reduction of specific types of noise to have more margin for other types. Both types of checks are performed by the library during *dpcmCalcInputNoise* and *dpcmCalcOutputNoise*, and notification of violations is provided to the application via the *appSetNoiseViolation* function.

10.22.1 Types of noise

This subclause contains definitions for several predefined types of noise as well as an interface through which the library can define additional noise types.

10.22.1.1 noiseType

noiseType is an enumerated type that identifies the type of noise involved in a calculation, as shown in Table 259.

Table 259—DCM_NoiseTypes

Enumerated name	Value	Definition
DCM_TOTAL_NOISE	0	A composite value representing the sum of all of the types of noise at a given pin.
DCM_PROPAGATED_NOISE	1	Noise propagated from a previous stage.
DCM_CHARGE_SHARING_NOISE	2	Noise due to charge redistribution between parts of a circuit.
DCM_POWER_NOISE	3	Noise originating on the power or ground rails of a cell.
DCM_CROSS_TALK_NOISE	4	Noise induced on a victim net by activity on an aggressor net through capacitive or inductive coupling.
DCM_LIBRARY_DEFINED_NOISE	5	A type of noise defined by the library.

10.22.1.1.1 dpcmGetLibraryNoiseTypesArray

Table 260 provides information on *dpcmGetLibraryNoiseTypesArray*.

Table 260—dpcmGetLibraryNoiseTypesArray

Function name	dpcmGetLibraryNoiseTypesArray
Arguments	None
Result	Array of noise types
Standard Structure fields	None
DCL syntax	EXPOSE(dpcmGetLibraryNoiseTypesArray): result(string[*]: libraryNoiseTypes);
C syntax	<pre>enum DCM_NoiseTypes { DCM_TOTAL_NOISE, DCM_PROPAGATED_NOISE, DCM_CHARGE_SHARING_NOISE, DCM_POWER_NOISE, DCM_CROSS_TALK_NOISE, DCM_LIBRARY_DEFINED_NOISE }; typedef struct { DCM_STRING_ARRAY *libraryNoiseTypes; } T_LibraryNoiseTypesArray; int dpcmGetLibraryNoiseTypesArray(const DCM_STD_STRUCT *std_struct, T_LibraryNoiseTypesArray *rtn);</pre>

This returns an ordered array of strings identifying the custom noise types defined by the library. Each technology can define a single set of custom noise types that are to be used for all of its cells. If a technology has no custom noise types, then this function shall return an empty array.

10.22.2 Noise models

A library *that supports noise analysis* shall define a model in the noise domain for each of its cells. Within the noise domain, there are no arcs. Instead, noise models shall consist of information about relationships between the output(s) and the other related pins on a cell.

These data shall be in the form of cones of influence known as noise cones, one per output pin, specifying the related pins from which noise can be propagated to that output pin. Noise can be propagated to an output pin from any other pin, including an input, another output, a power pin, or an internal node. An internal node can also assume the role of an output pin for noise-propagation purposes.

To obtain a noise model for a cell, the application shall call the *modelSearch* function with the *modelDomain* field in the standard structureset to the string “noise”. In response, the library shall call the function *appNewNoiseCone* for each output pin or internal node to which noise can be propagated.

Because a bidirectional pin can act as either an input or an output, it can appear as both the output pin of a noise cone and as a related pin in the cone for another output pin. Because of this, the application shall use the pin handle provided in each call to determine to which of these roles the call applies. For this to be possible, the application shall provide different, distinguishable handles for a bidirectional pin in the input pin and output pinlists in the Standard Structure passed to *modelSearch*.

When a cell is modeled, applications that choose to include noise contributions associated with power-supply pins shall include the cell’s supply pins in the list of the input pins contained in the *Standard Structure*. The application shall identify the supply pins by calling 10.23.22.2. The application shall not include supply pins in the list of output pins.

Libraries that model noise contributions associated with power pins shall include those pins in the related-pins array passed to the application by *appNewNoiseCone*.

While the `modelSearch` function is executing, the library can call the function `newTimingPin` (see 10.21.8) to define an internal node within a cell. The library can also call `newNetSinkPropagateSegments` and `newNetSourcePropagateSegments` to convey `pathData` pointers for input and output pins, respectively, to the application (see 10.21.8).

DCL can be used to define a cell model in the noise domain. A `MODELPROC` or `SUBMODEL` construct associated with the noise domain can contain `INPUT` and `OUTPUT` statements describing the cell's noise configuration to the application. The `INPUT` statements describe the data to be used by the library during noise calculations performed at input pins. The `OUTPUT` statements describe the cones of influence passed to the application and cache noise-related data to be use by the library during noise calculations at output pins.

An `OUTPUT` statement contained within a `MODELPROC` in the noise domain shall not contain a propagate sequence but shall have a post code sequence containing a call to `appNewNoiseCone`. The `OUTPUT` statement(s) may contain `STORE` and `METHODS` clauses for caching and retrieving library data.

An `INPUT` statement contained within a `MODELPROC` in the noise domain shall not contain a propagate sequence. `INPUT` statements can contain `STORE` and `METHODS` clauses for caching and retrieving library data.

The following example demonstrates the use of a DCL `OUTPUT` statement in the noise domain:

```
tabledef(noiseCones):
    passed(string: cellName, domain, outputPin)
    qualifiers(cellName, domain, outputPin)
    data(string[*]: relatedPinNames);

import calc(locateRelatedPins):
    passed(string[*]: relatedPinNames)
    result(pin[*]: relatedPins);

external(appNewNoiseCone):
    passed(pin: outputPin; pin[*]: relatedPins)
    result(integer: ignore);

model(n):
    defines(INV1.*.noise);

modelproc(n):
    output(Z):
        store(noiseCones(cell, model_domain, 'Z'))

appNewNoiseCone(
    from_point,
    locateRelatedPins(
        noiseCones(cell, model_domain, 'Z'
        ).relatedPinNames
    ).relatedPins
);
end;

table(noiseCones):
    INV1.noise.Z: ['A'];
end;
```

10.22.2.1 appNewNoiseCone

Table 261 provides information on appNewNoiseCone.

Table 261—appNewNoiseCone

Function name	appNewNoiseCone
Arguments	Output pin, Array of related pins
Results	None
Standard Structure fields	None
DCL syntax	<pre>external(appNewNoiseCone): passed(pin: outputPin; pin[*]: relatedPins) result(int: ignore);</pre>
C syntax	<pre>int appNewNoiseCone(DCM_STD_STRUCT *std_struct, T_Ignore *rtn, DCM_PIN outputPin, DCM_PIN_ARRAY *relatedPins);</pre>

This function describes the set of pins that are related to (that can have noise affecting) an output pin on a cell. The order of the *related* pins in the array provided is significant. The application shall use this order in subsequent calls to *dpcmCalcOutputNoise* (see 10.22.6.1.1). The related pins may include inputs, power pins, internal nodes, and other output pins.

In response to a call to *modelSearch* made by the application for the noise domain, the library shall call this function for each output pin to which noise can be propagated.

10.22.2.2 Interconnect noise cones

For each input pin on a net, the application shall create an interconnect noise cone. All dc-connected drivers and all aggressor drivers that are present in the parasitic network supplied to the library for that net (see 10.22.4) shall be included in this cone. The order of these drivers shall be the same as the order in which they first appear in that parasitic network. This same driver order shall be used when the application requests that the library calculate noise for the input pin (see 10.22.5).

A bidirectional pin can be included in multiple interconnect noise cones for a net. It can appear as the target pin of an interconnect cone (reflecting its role as an input) and as a related pin in other interconnect cones (reflecting its role as an output). A bidirectional pin shall not appear as both the target pin and as a related pin in the same interconnect noise cone.

If instructed by the library, the application shall substitute an internal node for any given pin in an interconnect noise cone (see 10.21.8).

10.22.2.3 Modeling internal nodes

For situations in which a cell is complicated enough to warrant separate analysis for individual internal nodes, the library can use IMPORT or EXPORT clauses to associate these internal nodes with the appropriate pins. The application shall treat each identified imported or exported node as a valid measurement point associated with the input or output pin present in the corresponding clause. Each such node shall be connected to one or more noise cones that begin or end at the node.

The application shall detect the presence of an IMPORT or EXPORT clause in a model through a library call to *newNetSinkPropagateSegments* or *newNetSourcePropagateSegments*, respectively. However, the pin involved can be either an input or an output.

The following rules shall govern how the application creates interconnect noise cones in response to these calls:

- If an import pin is an input, the application shall create a noise cone from the driver pins on the connected net to the internal sink node (the node that is the target of the import). This cone shall be created in lieu of that which would otherwise be directed from those drivers to the import pin.
- If an import pin is an output, the application shall create a noise cone from the driver pins on the connected net, excluding the import pin, to the internal sink node. This cone shall be created in addition to those directed from the import pin and the other drivers to the sink pins connected to the net.
- If an export pin is an input, the application shall include the internal source node (the node that is the target of the export) together with the driver pins of the connected net in the noise cone directed to each sink pin on that net, excluding the export pin. These cones shall be created in addition to that directed from the drivers to the export pin.
- If an export pin is an output, the application shall include the internal source node together with the other driver pins of the connected net in the noise cone directed to each sink pin on the net. The export pin shall not be included in these noise cones.

For a bidirectional pin or tristate output, an imported or exported internal node shall be associated with the source or sink parasitic subnet for that pin, but not with both.

10.22.3 Noise waveforms

Noise waveforms typically originate from a transition on an aggressor net, although they can also be created in other ways. Like transitions, noise waveforms do not have a precise start time. Instead, each noise waveform is defined to start within a window of time bounded by early and late values. The characteristics of each noise waveform are defined with respect to their own internal reference point in time, which has the value 0.

The start time of a noise waveform shall be considered to be an offset added to each time point defined within that waveform. For a noise waveform having a synchronous relationship with the temporal context in which it is used, the earliest and latest possible offsets shall be associated with that waveform. Such a waveform shall be considered to be a synchronous waveform in that context. A noise waveform having an asynchronous relationship with a temporal context shall be considered to be an asynchronous waveform in that context. The earliest and latest offsets in an asynchronous waveform shall be considered to be undefined, and the library shall assume that the waveform may start at any time relative to its context.

There are two representations of noise waveforms. For noise calculations involving a single technology, the NWF representation, which associates the early and late offsets with arbitrary, library-private data, shall be used. The PWF representation shall be used when noise calculations involve more than one technology or when noise waveforms are supplied by the application. Each PWF associates early and late offsets with an initial voltage and a set of subsequent time-voltage pairs.

The application shall determine whether a noise calculation involves more than one technology. For each noise waveform originating in a different technology than the one responsible for the calculation, the application shall call `dpcmGetPWFArray` to convert the noise waveform from an NWF representation to a PWF representation. When the calculation is in the same synchronous frame of reference (temporal context), the offsets of the NWF and the resultant PWF shall be the same.

The application shall also determine whether a noise calculation involves waveforms that are with respect to multiple temporal frames of references, and if so convert all of the waveforms into a single consistent frame of reference for the calculation. To do so, an application shall call `dpcmCopyNWFArray` or `dpcmCopyPWFArray` for each waveform array, then set the offsets for each entry in the new array to reflect this other frame of reference.

When the calculation is in a different frame of reference that has a synchronous relationship to the original frame of reference for a waveform, the offsets in the copy of the waveform shall be adjusted to reflect the difference between the two frames of reference. When the calculation is in a frame of reference that has an asynchronous relationship to the original frame of reference for a waveform, the waveform type shall be set to DCM_ASYNCCHRONOUS in the copy.

For noise calculations on a victim net that include the effects of noise waveforms at dc-connected driver pins or coupled aggressor-driver pins, where the receiver pin on the victim net is in a different technology than a given driver pin, output resistance shall be used in a simplified driver model for that driver pin. A PWFdriverModelstructure contains a PWF and the corresponding output resistances for the voltages in the PWF. These voltages and output resistances shall together form a set of Thevenin equivalent models for the driver, one model for each point in the PWF.

The application shall determine when a PWFdriverModelarray is needed for a driver (rather than the corresponding NWFs) and shall then call dpcmGetPWFdriverModelArrayto obtain that array (Table 262).

Table 262—NWF type

DCL syntax
<pre>typedef (NWF) : result(int var: waveformType; double var: earlyOffset, lateOffset; void var: NWFinfo);</pre>
C syntax
<pre>enum DCM_WaveformTypes { DCM_ASYNCCHRONOUS, DCM_SYNCHRONOUS }; typedef struct { DCM_WaveformTypes waveformType; DCM_DOUBLE earlyOffset, lateOffset; const DCM_STRUCT *NWFinfo; } DCM_NWF; typedef DCM_NWF *DCM_NWF_ARRAY;</pre>

An NWF shall contain early and late offsets and arbitrary, library-private data that together describe a noise waveform. A field indicating the type of the waveform (synchronous or asynchronous) shall also be included. The early offset shall represent the earliest possible start time for the NWF, whereas the late offset shall represent its latest possible start time. The offsets in an asynchronous NWF shall be considered irrelevant and shall be ignored.

The NWFinfo field is a pointer to a library-private representation of the noise waveform. The memory for this representation shall be managed by the library. Applications shall not modify this field or attempt to use it or the data to which it points in any way.

When multiple NWFs are stored in an NWF array, they shall represent a collection or set of noise waveforms at a particular pin. The order of the entries in the array shall not be considered to be significant, except perhaps to the library technology that has created it (Table 263).

Table 263—PWF type

DCL syntax
<pre>typedef(PWFpoint): result(double var: time, voltage); typedef(PWF): result(int var: waveformType; double var: earlyOffset, lateOffset; var PWFpoint var[*] var: points);</pre>
C syntax
<pre>typedef struct { DCM_DOUBLE time; DCM_DOUBLE voltage; } DCM_PWFpoint; typedef DCM_PWFpoint *DCM_PWFpoint_ARRAY; typedef struct { DCM_WaveformTypes waveformType; DCM_DOUBLE earlyOffset, lateOffset; DCM_PWFpoint_ARRAY *points; } DCM_PWF; typedef DCM_PWF *DCM_PWF_ARRAY;</pre>

A PWF structure explicitly represents the shape of a noise waveform as a set of points, where each point consists of a time and a voltage value. The first time value shall always be 0, which shall be used as a reference time for the subsequent points. The subsequent points shall have monotonically increasing time values.

A field indicating the type of the waveform (synchronous or asynchronous) is also included. The early offset represents the earliest possible start time for the PWF, whereas the late offset represents its latest possible start time. The offsets in an asynchronous PWF are irrelevant and shall be ignored.

Libraries and applications shall use piece-wise linear interpolation when interpreting a PWF to determine voltage values between points. Between each pair of points, a linear segment is implicitly defined, and for any time value between those two points, the corresponding voltage shall be calculated to fall on that linear segment. For time values less than zero (0), the voltage shall be defined to be the same as the voltage at time zero (0). For time values greater than the time of the last point, the voltage value shall be defined to be the same as the voltage at that point.

When the area under the curve of a noise waveform is calculated, the baseline voltage shall be defined as a linear segment between the voltage at time zero (0) and the voltage at the last point. Although this voltage might not be a constant, the noise waveform itself shall be described completely between the first and last points.

The PWF for a noise waveform that represents a constant dc voltage shall contain a single point whose time is zero and whose voltage is that constant. The area associated with a constant noise waveform shall be defined to be zero (0).

When multiple PWFs are stored in a PWF array, they shall represent a collection or set of noise waveforms. The order of the entries in this array shall not be considered to be significant (Table 264).

Table 264—PWFdriverModel

DCL syntax
<pre>typedef (PWFdriverModel) : result(var PWF: pwf ; double var[*] var: resistances);</pre>
C syntax
<pre>typedef structure { const DCM_PWF *pwf; DCM_DOUBLE_ARRAY *resistances; } DCM_PWFdriverModel; typedef DCM_PWFdriverModel *DCM_PWFdriverModel_ARRAY;</pre>

A *PWFdriverModel* contains both a noise waveform (as a PWF) and, for each point in the PWF, an associated output resistance. Each output resistance, together with the voltage in the corresponding PWF point, shall constitute a Thevenin-equivalent model for that point on the waveform. The size of the resistance array shall be the same as the number of points in the PWF. The order of the points and the entries in the resistances array shall be the same.

10.22.3.1 dpcmGetPWFarray

Table 265 provides information on dpcmGetPWFarray.

Table 265—dpcmGetPWFarray

Function name	dpcmGetPWFarray
Arguments	Noise waveform array
Results	Pulse waveform array
Standard Structure fields	None
DCL syntax	<pre>EXPOSE(dpcmGetPWFarray) : passed(NWF[*]: NWFarray) result(PWF[*]: PWFarray);</pre>
C syntax	<pre>typedef struct { DCM_PWF_ARRAY *PWFarray; } T_PWFarray; int dpcmGetPWFarray(const DCM_STD_STRUCT *std_struct, T_PWFarray *rtn, DCM_NWF_ARRAY *NWFarray);</pre>

dpcmGetPWFarray converts an array of library-proprietary NWFs into an array of PWFs, which are suitable for interpretation by an application or by a different technology than that which created the NWF. Applications shall call this function to convert noise waveforms for propagation from one technology to another.

10.22.3.2 dpcmCreatePWF

Table 266 provides information on dpcmCreatePWF.

Table 266—dpcmCreatePWF

Function name	dpcmCreatePWF
Arguments	Number of points
Results	Empty pulse waveform
Standard Structure fields	None
DCL syntax	<pre>EXPOSE(dpcmCreatePWF): passed(integer numPoints) result(PWF: emptyPWF);</pre>
C syntax	<pre>typedef struct { DCM_PWF *emptyPWF; } T_PWF; int dpcmCreatePWF(DCM_STD_STRUCT *std_struct, T_PWF *rtn, DCM_INTEGER numPoints);</pre>

dpcmCreatePWF creates a PWF with *numPoints* points in which the waveform type, the offsets, and the time and voltage values for each point are all uninitialized. The application shall fill in these fields with the appropriate values for a specific waveform. The resulting PWF is then suitable for subsequent use by the library, for example, in calls to *dpcmCalcInputNoise* or *dpcmCalcOutputNoise*.

Applications shall call this function to create waveforms for noise sources that they define, such as for noise injected on a primary input. As each PWF is created, the application shall lock it, then unlock it when it is no longer needed.

To create an array of PWFs, the application shall call *dcm_new_DCM_ARRAY* and then set each pointer in that array to point to individual, locked PWFs created using *dpcmCreatePWF*. When it is no longer needed, the application shall unlock this array as a unit, which shall result in its PWF entries being unlocked automatically.

If an individual PWF is included in more than one PWF array, it shall be locked once for each of those inclusions.

10.22.3.3 dpcmCopyNWFarray

Table 267 provides information on *dpcmCopyNWFarray*.

Table 267—dpcmCopyNWFarray

Function name	dpcmCopyNWFarray
Arguments	NWF array
Results	NWF array copy
Standard Structure fields	None
DCL syntax	<pre>EXPOSE(dpcmCopyNWFarray): passed(NWF[*]: sourceNWFarray) result(NWF[*]: NWFarray);</pre>
C syntax	<pre>typedef struct { DCM_NWF_ARRAY *NWFarray; } T_NWFarray; int dpcmCopyNWFarray(const DCM_STD_STRUCT *std_struct, T_NWFarray *rtn, DCM_NWF_ARRAY *sourceNWFarray);</pre>

dpcmCopyNWFFarray allocates space for a new array of NWFs and copies the contents of the source NWF array passed to it into this new array. Each entry in the new array shall have the same values for its *waveformType*, *earlyOffset*, *lateOffset*, and *NWFinfo* fields as the corresponding entry in the source array. For each entry, the structure to which *NWFinfo* points shall be locked an additional time to account for its use in the new array.

10.22.3.4 dpcmCopyPWFarray

Table 268 provides information on *dpcmCopyPWFarray*.

Table 268—dpcmCopyPWF

Function name	dpcmCopyPWF
Arguments	PWF array
Results	PWF array copy
Standard Structure fields	None
DCL syntax	EXPOSE (dpcmCopyPWFarray) : passed(PWF[*]: sourcePWFarray) result(PWF[*]: PWFarray);
C syntax	int dpcmCopyPwf(const DCM_STD_STRUCT *std_struct, T_PWFarray *rtn, DCM_PWF_ARRAY *sourcePWFarray);

dpcmCopyPWFarray allocates space for a new PWF array and then copies the contents of the source PWF array passed to it into this new array. Each entry in the new array shall have the same values for its *waveformType*, *earlyOffset*, *lateOffset*, and *points* fields as the corresponding entry in the source array. For each entry, the array to which *points* points shall be locked an additional time to account for its use in the new array.

10.22.3.5 dpcmCreatePWFdriverModel

Table 269 provides information on *dpcmCreatePWFdriverModel*.

Table 269—dpcmCreatePWFdriverModel

Function name	dpcmCreatePWFdriverModel
Arguments	Pulse waveform
Results	PWF driver model
Standard Structure fields	None
DCL syntax	EXPOSE (dpcmCreatePWFdriverModel) : passed(PWF: pwf) result(PWFdriverModel: emptyPWFdriverModel);
C syntax	typedef struct { DCM_PWFdriverModel *emptyPWFdriverModel; } T_PWFdriverModel; int dpcmCreatePWFdriverModel(const DCM_STD_STRUCT *std_struct, T_PWFdriverModel *rtn, DCM_PWF *PWF);

dpcmCreatePWFdriverModel creates a *PWFdriverModel* that incorporates the PWF passed to it and includes a resistance array whose size is the same as the number of points in the PWF. The library shall lock the PWF to reflect its use in the new *PWFdriverModel*. The resistance values shall be uninitialized and shall be set by the application to the appropriate values for a specific driver model. The resulting *PWFdriverModel* is then suitable for subsequent use by the library, for example in calls to *dpcmCalcInputNoise*.

Applications shall call this function when defining noise models for output pins in a circuit and for external drivers of primary inputs. As each *PWFdriverModel* is created, the application shall lock it, then unlock it when it is no longer needed.

To create a *PWFdriverModel* array, the application shall call *dcm_new_DCM_ARRAY* and then set each pointer in that array to an individual, locked *PWFdriverModel* created using *dpcmCreatePWFdriverModel*. When it is no longer needed, the application shall unlock the array as a unit, which shall result in its *PWFdriverModel* entries being unlocked automatically.

If an individual *PWFdriverModel* is included in more than one *PWFdriverModel* array, it shall be locked once for each of those inclusions.

10.22.3.6 dpcmGetPWFdriverModelArray

Table 270 provides information on *dpcmGetPWFdriverModelArray*.

Table 270—*dpcmGetPWFdriverModelArray*

Function name	<i>dpcmGetPWFdriverModelArray</i>
Arguments	Driver pin, Array of noise waveforms
Results	Array of PWF driver models
Standard Structure fields	CellName, block, calcMode, CellData (noise-specific), pathData (noise-pin-specific), processVariation
DCL syntax	<pre>EXPOSE(dpcmGetPWFdriverModelArray): passed(pin: driverPin; NWF[*]: NWFarray) result(PWFdriverModel[*]: PWFdriverModelArray);</pre>
C syntax	<pre>typedef struct { DCM_PWFdriverModel_ARRAY *PWFdriverModelArray; } T_PWFdriverModelArray; int dpcmGetPWFdriverModelArray(DCM_STD_STRUCT *std_struct, T_PWFdriverModelArray *rtn, DCM_PIN driverPin, DCM_NWF_ARRAY *NWFarray);</pre>

dpcmGetPWFdriverModelArray creates an array of PWF driver models using an array of library-proprietary NWFs and output resistances for the driver pin passed to it. These NWFs shall be assumed to occur at that driver pin, and the output resistances shall correspond to the voltages in the PWFs contained within the resulting driver models.

10.22.4 Noise network models

For noise propagation across an interconnect network, a large amount of information is required. This information shall include the interconnect parasitics associated with the network, coupling parasitics to any aggressor nets, and the interconnect parasitics for those aggressors, and the pin-parasitic subnetworks for each pin that is a port on the overall network. Similarly, for noise propagation across a cell, a full parasitic network for the output net, including any coupled aggressor nets, is needed.

The representation of parasitics is described in more detail in 10.21.6. The application shall build a full parasitic network for a net and any aggressor nets coupled to it, including the pin-parasitic subnetworks, as described in 10.21.6.10.

When preparing a parasitic network for noise calculations, pin parasitics shall be handled differently than for timing calculation. The application shall call *dpcmSinkPinNoiseParasitics* and *dpcmSourcePinNoiseParasitics* to obtain appropriate pin-parasitic subnetworks for use during noise calculations.

When creating a parasitic network for use during interconnect-noise propagation, all driver pins shall be modeled as sources, including both dc-connected drivers and aggressor drivers. When creating a parasitic network for cell-noise propagation, the driver pin that is the destination of the propagated noise shall be modeled as a source, aggressor drivers shall also be modeled as sources, but any other dc-connected drivers on the driven net shall be modeled as sinks.

When the noise propagated across a given net is calculated multiple times for an input pin, performance can be significantly improved by first creating *noise interconnect* models for this pin that are specifically designed for use by the library's internal algorithms and then reusing the models during those calculations. The application shall call *dpcmCreateNoiseInterconnectModels* to allow the library to create these models. The library can use a parasitic network as a noise interconnect model if it chooses.

Similarly, when the noise propagated across a cell to an output pin is calculated multiple times, performance can be improved by creating and reusing *noise load* models for this pin. The application shall call *dpcmBuildNoiseLoadModels* to allow the library to create these models. The library can use a parasitic network as a noise load model if it so chooses.

Each noise load model shall contain sufficient information so that the library can later identify the associated driver node in the corresponding network using that model. For a network having multiple, parallel drivers, the load models for the drivers shall contain sufficient information so that all of the driver nodes can be identified when these models are used together.

Each noise interconnect model and the noise load models for the associated driver pins shall together contain sufficient information so that the library can later identify the driver and sink nodes in the corresponding network using those models. For a network having multiple, parallel drivers, an interconnect model and the load models for all of these drivers shall contain sufficient information so that all of the drivers can be identified when these models are used together.

When a library is called several times to create noise interconnect models for the same parasitic network, it can be more efficient for the library to create an intermediate model of the network once and then convert that intermediate model into the noise interconnect noise models for the sink pins on the network. This intermediate representation of the network shall be referred to as a *noise network* model.

Multiple calls to create noise load models for the same net shall be handled in a similar manner. However, since parasitic networks used to create noise load models contain different pin-parasitic subnetworks, the noise network models produced by the library can be different as well.

If the library chooses not to use noise network models and simply creates noise interconnect and load models directly from the parasitic networks passed to it, the library shall return null pointer values for the noise network models.

To accommodate on-chip process variations, the application *can* pass pointers to two parasitic networks for each net to the library, one with *element values* for the minimum and another *with values* for the maximum of the on-chip process uncertainty (see 10.23.11). These two networks shall be identical in all other respects, including the order in which their elements and nodes appear. The library shall return two corresponding noise load or interconnect models and, optionally, two corresponding noise network models to the application, all via pointers of type DCM_STRUCT *.

If on-chip process variation is not modeled, these networks shall be *completely* identical, as shall the corresponding library network models *and the load or interconnect models*. The parasitic-network pointers passed to the library shall then be the same, as shall the load- or interconnect-model pointers and the network-model pointers returned to the application.

To support libraries that use noise network models, during the first request for noise interconnect models for a net, the application shall supply null pointers for the library's noise network models. In subsequent

requests to the library, the application shall supply the noise network model pointers returned in response to the previous request.

The application shall set (to a value of one (1)) the change field of the first subnet in a parasitic network that has changed since a noise load or interconnect model for that network was last requested. This shall be done whether the change was to an existing subnet, the insertion of a new subnet, the deletion of a subnet, or a change in the order the subnets were presented. The library can thus omit the reconstruction of what would otherwise be duplicate information.

The application shall lock each noise network model received from the library. When the application has determined that all the noise load or interconnect models derived from that network model have been obtained, the application shall unlock that model. At this point, if no other reference to it exists, the memory used for that model shall then be freed by the library.

The application shall lock the load and interconnect noise models returned to it if it needs to retain them beyond the time it makes the next call to any library function. When the application no longer needs a noise model that it has already locked, it shall unlock that model. Again, if no other reference to it exists, the memory used for the model shall then be freed by the library.

10.22.4.1 Noise pin parasitics

A pin's parasitic network can be different depending on whether it is driving the connected net, in a high impedance state, or receiving a logical signal. For example, a pin that is bidirectional can act as a receiver or a driver, while a tristate output pin can be in an active or a high-impedance state, depending on whether or not it is driving the attached net.

In the noise domain, a pin shall not have different parasitic subnets for rising and falling signals. If the library models on-chip process variation, a pin *can* have different subnets with different *element* values representing the minimum and maximum of the on-chip process uncertainty (see 10.23.11). These two subnets shall be identical in all other respects, including the order in which their elements and nodes appear. The application shall be responsible for stitching the appropriate subnets into overall interconnect networks for calculations to be performed.

There shall be two separate functions for obtaining pin parasitics, one for sink (receiver or high impedance) and one for source (driver) pin roles. For both of these functions, the library shall return pointers to *either* one or two subnets, *with two subnets* representing the minimum and maximum on-chip process variation. If the library does not support on-chip process variation, then the pointers returned for the minimum of the on-chip process uncertainty shall be identical to those for its maximum.

Source-pin parasitics shall include only those elements needed to represent the constant portion of an output pin's parasitics. Elements representing the active part of a driver pin such as output resistance or admittance shall not be included. Instead, the library can include driver models for an output pin in the NWF structures (see 10.22.3), for example, associated with that pin.

The library shall return pointers to subnets appropriate for the current analysis conditions. Different conditions, with regard to operating point, for example, can yield subnets that are topologically as well as numerically different.

Although a bidirectional pin can act as an input or an output, as observed at the pin, it shall not do both simultaneously. Consequently, when a parasitic network for a net is created for use when a bidirectional pin drives that net, the application shall include source parasitics only for that pin. Sink parasitics for the pin shall not be included under such circumstances. Conversely, while creating a parasitic network for use when a different pin drives the net, only the sink parasitics for that bidirectional pin shall be included.

10.22.4.1.1 dpcmGetSinkPinNoiseParasitics

Table 271 provides information on dpcmGetSinkPinNoiseParasitics.

Table 271—dpcmGetSinkPinNoiseParasitics

Function name	dpcmGetSinkPinNoiseParasitics
Arguments	Sink-pin pointer
Result	Parasitic subnets
Standard Structure fields	CellName, cellData (noise), pathData (noise-pin-specific), calcMode
DCL syntax	<pre>EXPOSE(dpcmGetSinkPinNoiseParasitics): passed(pin: sinkPin) result(var parasiticSubnet: minSubnet, maxSubnet);</pre>
C syntax	<pre>typedef struct { DCM_ParasiticSubnet *minSubnet, *maxSubnet; } T_PinNoiseParasitics; int dpcmGetSinkPinNoiseParasitics(const DCM_STD_STRUCT *std_struct, T_PinNoiseParasitics rtn, const DCM_PIN sinkPin);</pre>

This returns two pointers to parasitic subnets for a sink pin, which contain element values for minimum and maximum on-chip process variation. These subnets shall be constructed from memory allocated by the library that is suitable for modification by the application. If the two subnets are *completely* identical, the same pointer shall be returned for both of them. It shall be a severe error to call this function for a pin that cannot act as a sink.

10.22.4.1.2 dpcmGetSourcePinNoiseParasitics

Table 272 provides information on dpcmGetSourcePinNoiseParasitics.

Table 272—dpcmGetSourcePinNoiseParasitics

Function name	dpcmGetSourcePinNoiseParasitics
Arguments	Source-pin pointer
Result	Parasitic subnets
Standard Structure fields	CellName, cellData (noise), pathData (noise-pin-specific), calcMode
DCL syntax	<pre>EXPOSE(dpcmGetSourcePinNoiseParasitics): passed(pin: sourcePin) result(var parasiticSubnet: minSubnet, maxSubnet);</pre>
C syntax	<pre>typedef struct { DCM_ParasiticSubnet *minSubnet, *maxSubnet; } T_PinNoiseParasitics; int dpcmGetSourcePinNoiseParasitics (const DCM_STD_STRUCT *std_struct, T_PinNoiseParasitics rtn, const DCM_PIN sourcePin);</pre>

This returns two pointers to parasitic subnets for a source pin, which contain element values for minimum and maximum on-chip process variation. These subnets shall be constructed from memory allocated by the library that is suitable for modification by the application. If the two subnets are *completely* identical, the same pointer shall be returned for both of them. It shall be an error to call this function for a pin that cannot act as a source.

10.22.4.1.3 dpcmBuildNoiseInterconnectModels

Table 273 provides information on `dpcmBuildNoiseInterconnectModels`.

Table 273—`dpcmBuildNoiseInterconnectModels`

Function name	<code>dpcmBuildNoiseInterconnectModels</code>
Arguments	Parasitic networks, Noise-network, models, Sink pin, Sink-pin node numbers
Results	Noise-interconnect, models, Noise-network models
Standard Structure fields	<code>cellData</code> (noise), <code>cellName</code> , <code>block</code> , <code>pathData</code> , (noise-pin-specific)
DCL syntax	<pre>EXPOSE(dpcmBuildNoiseInterconnectModels): passed(pin: sinkPin; int: minSinkPinNodeNumber, maxSinkPinNodeNumber; parasiticSubnet: minParasitics, maxParasitics; void: minNoiseNetworkModel, maxNoiseNetworkModel) result(void: minNoiseInterconnectModel, maxNoiseInterconnectModel, newMinNoiseNetworkModel, newMaxNoiseNetworkModel);</pre>
C syntax	<pre>typedef struct { const DCM_STRUCT *minNoiseInterconnectModel; const DCM_STRUCT *maxNoiseInterconnectModel; const DCM_STRUCT *newMinNoiseNetworkModel; const DCM_STRUCT *newMaxNoiseNetworkModel } T_NoiseInterconnectModels; int dpcmBuildNoiseInterconnectModels(DCM_STD_STRUCT *std_struct, T_NoiseInterconnectModels *rtn, DCM_PIN sinkPin, DCM_INTEGER minSinkPinNodeNumber, DCM_INTEGER maxSinkPinNodeNumber, const DCM_parasiticSubNet *minParasitics, const DCM_parasiticSubNet *maxParasitics, const DCM_STRUCT *minNoiseNetworkModel, const DCM_STRUCT *maxNoiseNetworkModel);</pre>

This returns pointers to the noise interconnect models for a sink pin (*sinkPin*) on an interconnect network, along with updated versions of the library's intermediate noise models for that network.

The *minParasitics* and *maxParasitics* arguments are pointers to linked lists of minimum and maximum parasitic subnets, respectively, that represent the net, any aggressor nets and the coupling to those nets, and the pins attached to all of the nets. The *minNoiseNetworkModel* and *maxNoiseNetworkModel* arguments are pointers to the library's intermediate models for the network. The *minSinkPinNodeNumber* and *maxSinkPinNodeNumber* arguments are the node numbers for the sink pin in the minimum and maximum networks, respectively.

If the parasitic networks passed to this function are identical, *minParasitics* and *maxParasitics* shall have the same value, as shall *minSinkPinNodeNumber* and *maxSinkPinNodeNumber*. Similarly, if the noise network models passed are identical, *minNoiseNetworkModel* and *maxNoiseNetworkModel* shall have the same value.

If the noise interconnect models are identical, the *minNoiseInterconnectModel* and *maxNoiseInterconnectModel* pointers returned shall have the same value. If the new noise network models are identical, the *minNoiseNetworkModel* and *maxNoiseNetworkModel* pointers returned shall have the same value.

10.22.4.1.4 dpcmBuildNoiseLoadModels

Table 274 provides information on dpcmBuildNoiseLoadModels.

Table 274—dpcmBuildNoiseLoadModels

Function name	dpcmBuildNoiseLoadModels
Arguments	Parasitic networks, Noise network models, Driving pin, Driving pin node numbers
Results	Noise load models, Noise network models
Standard Structure fields	cellData (noise), cellName, block, pathData (noise-pin-specific)
DCL syntax	<pre>EXPOSE(dpcmBuildNoiseLoadModels): passed(pin: drivingPin; int: minDrivingPinNodeNumber, maxDrivingPinNodeNumber; parasiticSubnet: minParasitics, maxParasitics; void: minNoiseNetworkModel, maxNoiseNetworkModel) result(void: minNoiseLoadModel, maxNoiseLoadModel, newMinNoiseNetworkModel, newMaxNoiseNetworkModel);</pre>
C syntax	<pre>typedef struct { const DCM_STRUCT *minNoiseLoadModel; const DCM_STRUCT *maxNoiseLoadModel; const DCM_STRUCT *newMinNoiseNetworkModel; const DCM_STRUCT *newMaxNoiseNetworkModel; } T_noiseModels; int dpcmBuildNoiseLoadModels(DCM_STD_STRUCT *std_struct, T_noiseModels *rtn; DCM_PIN drivingPin, DCM_INTEGER minDrivingPinNodeNumber, DCM_INTEGER maxDrivingPinNodeNumber, const DCM_parasiticSubNet *minParasitics, const DCM_parasiticSubNet *maxParasitics, const DCM_STRUCT *minNoiseNetworkModel, const DCM_STRUCT *maxNoiseNetworkModel);</pre>

This returns pointers to the noise load models for a driver (*drivingPin*) of an interconnect network, along with updated versions of the library's intermediate noise models for that network.

The *minParasitics* and *maxParasitics* arguments are pointers to linked lists of minimum and maximum parasitic subnets, respectively, that represent the driven net, any aggressor nets and the coupling to those nets, and the pins attached to all of the nets. The *minNoiseNetworkModel* and *maxNoiseNetworkModel* arguments are pointers to the library's intermediate models for the network. The *minDrivingPinNodeNumber* and *maxDrivingPinNodeNumber* arguments are the node numbers for the driver pin in the minimum and maximum networks, respectively.

If the parasitic networks passed to this function are identical, *minParasitics* and *maxParasitics* shall have the same value. Similarly, if the noise network models passed to this function are identical, *minNoiseNetworkModel* and *maxNoiseNetworkModel* shall have the same value.

If the noise load models are identical, the *minNoiseLoadModel* and *maxNoiseLoadModel* pointers returned shall have the same value. If the new noise network models are identical, the *minNoiseNetworkModel* and *maxNoiseNetworkModel* pointers returned shall have the same value.

10.22.5 Calculating composite noise at cell inputs

When calculating noise propagated across an interconnect network, the library can consider a variety of different factors. The activity on the dc-connected drivers may be important, as well as on the coupled aggressor drivers.

For each driver pin, the application shall provide this information to the library in a *driverPinNoise* structure. An array of *driverPinNoise* structures with one entry for each dc-connected driver and one entry for each aggressor driver shall be passed to the function *dpcmCalcInputNoise*, which performs noise calculation for a sink pin on the network.

A tristate bus can have several drivers that are active at different times. When a given driver is enabled and actively driving the bus, noise can be propagated from that driver onto the bus. However, when the driver is disabled and its output is in a high impedance state, the output stage of the driver is assumed to be perfectly isolated from the rest of the owning cell, and no noise shall be propagated from that driver onto the bus. The library may perform case analysis considering both the enabled and disabled states for each driver.

10.22.5.1 driverPinNoise

Table 275 provides information on driverPinNoise.

Table 275—driverPinNoise

DCL syntax
<pre>typedef(resistanceRange): result(double var: maxResistance, minResistance); typedef(driverTransitionWindow): result(double: earlyTime, lateTime, earlySlew, lateSlew, earlyResistance, lateResistance; void: earlyXWF, lateXWF); typedef(driverPinNoise): result(int: level; resistanceRange: holdingResistance, tristateResistance; driverTransitionWindow transient [*]: riseTransitions, fallTransitions; NWF[*]: NWFarray; PWFdriverModel[*]: PWFdriverModelArray);</pre>

C syntax

```
typedef struct {  
    DCM_DOUBLE maxResistance, minResistance;  
} DCM_ResistanceRange;  
  
typedef struct {  
    DCM_DOUBLE earlyTime, lateTime, earlySlew, lateSlew;  
    DCM_DOUBLE earlyResistance, lateResistance;  
    const DCM_STRUCT *earlyXWF, *lateXWF;  
} DCT_DriverTransitionWindow;  
  
typedef DCT_DriverTransitionWindow *DCM_DriverTransitionWindow_ARRAY;  
  
typedef struct {  
    DCM_LogicLevel level;  
    DCM_ResistanceRange holdingResistance, tristateResistance;  
    const DCM_DriverTransitionWindow_ARRAY *riseTransitions,  
    *fallTransitions;  
    const DCM_NWF_ARRAY *NWFarray;  
    const DCM_PWFdriverModel_ARRAY *PWFdriverModelArray;  
} DCT_DriverPinNoise;  
  
typedef DCT_DriverPinNoise *DCM_DriverPinNoise_ARRAY
```

A *driverPinNoise* structure represents all activity at a given driver pin that might be relevant for noise propagation across a net that the pin drives directly or for noise propagation onto a victim net that is coupled to the net that the pin drives. Although each structure is transient, it shall be contained within a managed array having an entry for each driver on the net and its aggressor nets.

Each *driverPinNoise* structure can include the following three types of data: constant values, full transitions, and propagated noise. The *level* field indicates whether a driver pin is held at a constant value (see 10.23.21.3). If *level* is *DCM_LogicUnknown* (indicating a nonconstant value), then the *riseTransitions* and *fallTransitions* fields shall point to arrays representing sets of windows during which full transitions (rise and fall, respectively) can occur. If only a single rise or a single fall transition can occur (but not both), either *riseTransitions* or *fallTransitions* shall be set to point to the transition data. The other transition-window array pointer shall then be set to zero (0), and it shall be assumed that no transitions of that other polarity can occur.

Each transition window shall be represented by a *driverTransitionWindow* structure in which the *earlyTime* and *lateTime* fields specify the window's time bounds. The waveform of the transition that occurs between those times (inclusive) shall be represented by slews and either XWFs or driver resistances. The *earlyResistance* and *lateResistance* fields represent the active resistance of the driver as it propagates transitions with slews of *earlySlew* and *lateSlew*, respectively. If either the *earlyXWF* or the *lateXWF* field has a value of zero (0), the library shall use only the slews and driver resistances. If either *earlyXWF* or *lateXWF* field is nonzero, the library shall ignore the driver resistances.

If the *level* field in a *driverPinNoise* structure is *DCM_LogicZero* or *DCM_LogicOne* (indicating that the driver pin is held at the corresponding constant value), *riseTransitions* and *fallTransitions* shall be set to zero (0). If no transition information is available, *level* shall be set to *DCM_LogicUnknown* and both *riseTransitions* and *fallTransitions* shall be set to zero (0). In this case, it shall be assumed that a rise or fall transition could occur at any time.

The *holdingResistance* field represents the range of quiescent resistance for periods of time during which there are no transitions or noise waveforms propagating through the driver. The application shall obtain these values by calling *dpcmCalcSteadyStateResistanceRange* (see 10.21.12.3). The *tristateResistance* field represents the resistance range of a tristate driver when the driver is disabled and in a high-impedance state. The application shall obtain these values by calling *dpcmCalcTristateResistanceRange* (see 10.21.12.4).

The *NWFarray* and *PWFdriverModelArray* fields are both pointers to arrays of noise waveforms present at the driver pin. Only one of these two fields shall be set to a nonzero value. The contents of the corresponding array shall be used during noise calculation whether the pin is held at a constant value or not.

When preparing a *driverPinNoise* structure for use by the library, the application shall determine whether the sink pin for which noise is to be calculated is in a different technology than the driver pin represented by that structure. If this is so, the application shall set the *earlySlew*, *lateSlew*, *earlyResistance*, and *lateResistance* fields in each *driverTransitionWindow* structure contained within the *driverPinNoise* structure, and it shall set *PWFdriverModelArray* to point to an array representing noise at that driver. For this case, the *earlyXWF* and *lateXWF* fields in each *driverTransitionWindow* structure and the *NWFarray* pointer shall all be set to zero (0).

The application can also use this approach for driver-pin data it supplies directly and that is not calculated by the library.

If the sink and driver pins are in the same technology (and the application is not supplying the driver-pin data), the application shall instead set the *NWFarray* field to point to an array representing noise propagated to the driver, and the *PWFdriverModelArray* pointer shall be set to zero (0). In this case, the application shall represent driver transitions using slews and either XWFs (if available) or the corresponding driver resistances in each *driverTransitionWindow* structure.

10.22.5.1.1 dpcmCalcInputNoise

Table 276 provides information on *dpcmCalcInputNoise*.

Table 276—dpcmCalcInputNoise

Function name	<i>dpcmCalcInputNoise</i>
Arguments	Input pin, Noise interconnect models, Driver noise array
Results	Noise waveforms
Standard Structure fields	block, CellName, cellData (noise-specific), pathData (noise-pin-specific), calcMode, processVariation
DCL syntax	<pre>EXPOSE(dpcmCalcInputNoise): passed(pin, inputPin; void: minInterconnectModel, maxInterconnectModel; driverPinNoise transient[*]: noises) result(NWF[*]: noiseWaveforms);</pre>
C syntax	<pre>typedef struct { DCM_NWF_ARRAY *noiseWaveforms; } T_NoiseWaveforms; int dpcmCalcInputNoise(const DCM_STD_STRUCT *std_struct, T_NoiseWaveforms *rtn, DCM_PIN inputPin, const DCM_NoiseInterconnectModel *minInterconnectModel, const DCM_NoiseInterconnectModel *maxInterconnectModel, const DCM_DriverPinNoise_ARRAY *noises);</pre>

This function calculates a set of composite noise pulses at the input of a cell, given the activity on the dc-connected drivers and the aggressor drivers on coupled nets. The function can also be called to calculate noise propagated to a bidirectional pin that is acting as an input.

An unbuffered output pin can act as a related pin (see 10.22.2.1) for other output pins on a cell, propagating noise into the cell from the net attached to the unbuffered pin. This function can be called for an unbuffered output pin to calculate noise that is then propagated to the other outputs.

There shall be one driverPinNoise entry in the noises array for each dc-connected driver and each aggressor driver. The application shall create this array and the transient structures within it (see 7.4.5.1.1). The order of these entries shall be the same as the order in which the corresponding dc-connected drivers and aggressor drivers first appeared in the minimum and maximum parasitic networks supplied by the application when dpcmBuildNoiseInterconnectModels (see 10.22.4.1.3) was called for the input pin.

If the noise interconnect models passed to this function are identical, the minInterconnectModel and maxInterconnectModel pointers shall have the same value.

When the input pin is connected to a tristate bus, the noises parameter shall be used to describe noise activity on each driver on the bus, including all tristate drivers. When performing case analysis considering both the enabled and disabled states for each driver, the library shall only consider the noise waveforms for a given tristate driver when *treating that driver as enabled*. When the library is treating a tristate driver *as disabled*, it shall use the tristate resistance range from the noises entry for the driver.

When the bus is properly implemented, only one driver is enabled at a given time, and the noise waveforms for different drivers do not overlap in time. If the noise waveforms for several tristate drivers overlap in time, this indicates a possibility of bus contention, which the library shall consider during its analysis.

If the library detects a noise violation while performing these calculations, it shall call *appSetNoiseViolation* to communicate information describing the violation to the application.

10.22.6 Calculating composite noise at cell outputs

When calculating noise propagated through a cell, the library can consider several different factors. The activity on the cell's input pins may be important, as well as the internal characteristics of the cell. Activity on other pins, such as power pins and related unbuffered outputs, can also be relevant.

A related pin can have full transitions and noise propagated to it from across an interconnect network, or it could be held at a constant logic level. For each related pin, the application shall provide this information to the library in the form of *relatedPinNoise* structures.

10.22.6.1 relatedPinNoise

Table 277 provides information on relatedPinNoise.

Table 277—relatedPinNoise

DCL syntax
<pre>typedef(transitionWindow): result(double: earlyTime, lateTime, earlySlew, lateSlew; void: earlyXWF, lateXWF); typedef(relatedPinNoise): result(int: level; transitionWindow transient[*]: riseTransitions, fallTransitions; NWF[*]: NWFarray; PWF[*]: PWFarray);</pre>

C syntax

```
typedef struct {
    DCM_DOUBLE earlyTime, lateTime, earlySlew, lateSlew;
    const DCM_STRUCT *earlyXWF, *lateXWF;
} DCT_TransitionWindow;

typedef DCT_TransitionWindow *DCM_TransitionWindow_ARRAY;

typedef struct {
    DCM_LogicLevel level;
    DCM_TransitionWindow_ARRAY *riseTransitions, *fallTransitions;
    const DCM_NWF_ARRAY *NWFarray;
    const DCM_PWF_ARRAY *PWFarray;
} DCT_RelatedPinNoise;

typedef DCT_RelatedPinNoise *DCM_RelatedPinNoise_ARRAY;
```

A *relatedPinNoise* structure represents all activity at a given pin on a cell that might be relevant for noise propagation through the cell to a related output pin. *While each structure is transient, it shall be contained within a managed array having an entry for each pin related to the output.*

Each *relatedPinNoise* structure can include the following three types of data: constant values, full transitions, and propagated noise. The *level* field indicates whether a related pin is held at a constant value (see 10.23.21.3). If *level* is *DCM_LogicUnknown* (indicating a nonconstant value), the *riseTransitions* and *fallTransitions* fields shall point to arrays representing sets of windows during which full transitions (rise and fall, respectively) can occur. If only a single rise or a single fall transition can occur (but not both), either *riseTransitions* or *fallTransitions* shall be set to point to the transition data. The other transition-window array pointer shall then be set to zero (0), and it shall be assumed that no transitions of that other polarity can occur.

Each transition window shall be represented by a *transitionWindow* structure in which the *earlyTime* and *lateTime* fields specify the window's time bounds. The waveform of the transition that occurs between those times (inclusive) shall be represented by slews and XWFs. If either the *earlyXWF* or the *lateXWF* field has a value of zero (0), the library shall use only the slews.

If the *level* field in a *relatedPinNoise* structure is *DCM_LogicZero* or *DCM_LogicOne* (indicating that the related pin is held at the corresponding constant value), *riseTransitions* and *fallTransitions* shall be set to zero (0). If no transition information is available, *level* shall be set to *DCM_LogicUnknown* and both *riseTransitions* and *fallTransitions* shall be set to zero (0). In this case, it shall be assumed that a rise or fall transition could occur at any time.

The *level* field in a structure associated with a power pin shall also be set to *DCM_LogicUnknown*.

The *NWFarray* and *PWFarray* fields are both pointers to arrays of noise waveforms present at the related pin. Only one of these two fields shall be set to a nonzero value. The contents of the corresponding array shall be used during noise calculation whether the pin is held at a constant value or not.

When the application supplies related-pin data directly (or none are available from the library), it shall set the *earlySlew* and *lateSlew* fields in each *transitionWindow* structure contained within the *relatedPinNoise* structure, and it shall set *PWFarray* to point to an array representing noise at that pin. For this case, the *earlyXWF* and *lateXWF* fields in each *transitionWindow* structure and the *NWFarray* pointer shall all be set to zero (0).

If related-pin data are available from the library and the application is not supplying it, the application shall instead set the *NWFarray* field to point to an array representing noise propagated to the pin, and the *PWFarray* pointer shall be set to zero (0). In this case, the application shall represent related-pin *transitions using slews and XWFs (if available)* in each *transitionWindow* structure.

10.22.6.1.1 dpcmCalcOutputNoise

Table 278 provides information on relatedPinNoise.

Table 278—dpcmCalcOutputNoise

Function name	dpcmCalcOutputNoise
Arguments	The output pin, Min and max noise load models, Array of related pin noises, Array of tristate driver resistances
Results	Noise waveforms
Standard Structure fields	block, CellName, cellData (noise-specific), pathData (noise-pin-specific), calcMode, processVariation
DCL syntax	<pre>EXPOSE(dpcmCalcOutputNoise): passed(pin: outputPin; void: minLoadModel, maxLoadModel; relatedPinNoise transient[*]: noises; resistanceRange[*]: tristateResistances) result(NWF[*]: noiseWaveforms);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE maxResistance, minResistance; } DCM_ResistanceRange; typedef DCM_ResistanceRange *DCM_ResistanceRange_ARRAY; typedef struct { DCM_NWF_ARRAY *noiseWaveforms; } T_NoiseWaveforms; int dpcmCalcOutputNoise(const DCM_STD_STRUCT *std_struct, T_NoiseWaveforms *rtn, DCM_PIN outputPin, const DCM_STRUCT *minLoadModel, const DCM_STRUCT *maxLoadModel, const DCT_RelatedPinNoise_ARRAY *noises, const DCM_ResistanceRange_ARRAY *tristateResistances);</pre>

This function calculates a set of composite noise pulses at the output of a cell, given the activity on the related pins. The function can also be called to calculate noise propagated to a bidirectional pin that is acting as an output.

There shall be one relatedPinNoise entry in the noises array for each related pin. The application shall create this array and the transient structures within it (see 7.4.5.1.1). The order of these entries shall be the same as the order in which the related pins appeared in the relatedPins array passed via appNewNoiseCone (see 10.22.2.1) for the output pin during model elaboration for the noise domain.

If the noise load models passed to this function are identical, the minLoadModel and maxLoadModel pointers shall have the same value.

When the output pin is a driver of a tristate bus, the resistances of the other drivers on the bus can affect noise propagation to that output. The tristateResistances array shall contain one entry for each driver on the bus, including the output pin itself. The order of the entries in the this array shall be the same as the order in which the bus drivers appeared in the min and max parasitic networks passed to dpcmBuildNoiseLoadModels when the noise load models for the output pin were created.

For non-tristate driver entries in the tristateResistances array, the DCM_ResistanceRange pointer shall be set to zero (0). If there are no tristate drivers on the net driven by the output pin, including the output pin itself, the tristateResistances pointer shall be set to zero (0).

If the library detects a noise violation while performing these calculations, it shall call *appSetNoiseViolation* to communicate information describing the violation to the application

10.22.6.2 Handling parallel drivers

When *dpcmCalcOutputNoise* is called for an output pin on one of a set of parallel drivers the library can choose to do noise propagation simultaneously across all of the parallel drivers. In that case, the library shall call *appForEachNoiseParallelDriver* for that output pin (the initiating driver pin) to gather information about the noise activity on the related pins of all of the other parallel driver pins.

In response, the application shall pass this noise activity data to the library by calling *dpcmSetParallelRelatedNoise* for each of those other parallel driver pins. When that function is called, the library shall record the activity on the related pins and the load models for that parallel driver pin. After this has been done for all of the other parallel driver pins, the function *appForEachNoiseParallelDriver* shall return the count of the other parallel driver pins to the library.

The library shall then calculate the noise propagated to each parallel driver pin, including the initiating driver pin. The noise calculated for each of the other parallel driver pins can be passed back to the application using *appSetParallelOutputNoisebefore* *dpcmCalcOutputNoise* returns. This avoids redundant calculations for each of the other parallel driver pins.

10.22.6.2.1 appForEachNoiseParallelDriver

Table 279 provides information on *appForEachNoiseParallelDriver*.

Table 279—appForEachNoiseParallelDriver

Function name	<i>appForEachNoiseParallelDriver</i>
Arguments	Initiating output pin
Results	Parallel driver count
Standard Structure fields	CellName, cellData (noise-specific), pathData (noise-pin-specific)
DCL syntax	<pre>EXPOSE (appForEachNoiseParallelDriver) : passed (pin: outputPin) result (int: countOfParallelDrivers);</pre>
C syntax	<pre>typedef struct { DCM_INTEGER countOfParallelDrivers; } T_ParallelDriverCount; int appForEachNoiseParallelDriver (const DCM_STD_STRUCT *std_struct, T_ParallelDriverCount *rtn, DCM_PIN outputPin);</pre>

When this function is called, the application shall respond by calling *dpcmSetParallelRelatedNoise* once for each of the other parallel driver pins on the net to which the passed pin is connected.

10.22.6.2.2 dpcmSetParallelRelatedNoise

Table 280 provides information on *dpcmSetParallelRelatedNoise*.

Table 280—dpcmSetParallelRelatedNoise

Function name	dpcmSetParallelRelatedNoise
Arguments	The output pin, Min and max noise load models, Array of input pin info
Results	None
Standard Structure fields	block, CellName, cellData (noise-specific), pathData (noise-pin-specific), calcMode, processVariation
DCL syntax	<pre> expose(dpcmSetParallelRelatedNoise): passed(pin: outputPin, initiatingPin; void: minLoadModel, maxLoadModel ; relatedPinNoise transient[*]: noises) result(integer: ignored); </pre>
C syntax	<pre> int dpcmSetParallelRelatedNoise (const DCM_STD_STRUCT *std_struct, T_Ignore *rtn, DCM_PIN initiatingPin, DCM_PIN outputPin, const DCM_STRUCT *minLoadModel, const DCM_STRUCT *maxLoadModel, const DCM_RelatedPinNoise_ARRAY *noises); </pre>

This function shall be called once for each driver pin on the net to which initiatingPin is connected that is parallel to initiatingPin. For each call, minLoadModel, maxNoiseLoadModel, and noises shall be passed to enable the library to propagate noise simultaneously across all of the parallel drivers. If, as a side effect of the original propagation started by dpcmCalcOutputNoise for the initiatingPin, the library computes appropriate noise waveforms at each parallel outputPin, those waveforms can be passed back to the application using appSetParallelOutputNoise.

10.22.6.2.3 appSetParallelOutputNoise

Table 281 provides information on appSetParallelOutputNoise.

Table 281—appSetParallelOutputNoise

Function name	appSetParallelOutputNoise
Arguments	The output pin, Min and max noise load models, Array of input pin info
Results	Noise waveforms
Standard Structure fields	block, CellName, cellData (noise-specific), pathData (noise-pin-specific), calcMode, processVariation
DCL syntax	<pre> expose(appSetParallelOutputNoise): passed(pin: outputPin; NWF[*]: noiseWaveforms) result(int: ignore); </pre>
C syntax	<pre> int appSetParallelOutputNoise (const DCM_STD_STRUCT *std_struct, T_Ignore *rtn, DCM_PIN outputPin, DCM_NWF_ARRAY *noiseWaveforms); </pre>

This function can be called by the library when noise waveforms are computed for the output of a parallel driver as a side effect of the calculations done by dpcmCalcOutputNoise for an initiating driver. The application shall store the noise waveforms passed and use them for subsequent noise propagation rather than making a separate call to dpcmCalcOutputNoise for the parallel output.

10.22.7 Setting noise budgets

Libraries can impose limits on the magnitude of particular types of noise to help ensure that a technology functions correctly. Applications can also choose to set budgets for particular noise types supported by the library.

10.22.7.1 dpcmSetNoiseLimit

Table 282 provides information on dpcmSetNoiseLimit.

Table 282—dpcmSetNoiseLimit

Function name	dpcmSetNoiseLimit
Arguments	Noise type, Library noise type, Noise limit
Results	violations
Standard Structure fields	CellName, block
DCL syntax	<pre>external(dpcmSetNoiseLimit): passed(NoiseType: noiseType ; int: libraryNoiseType; double: noiseLimit) result(int: ignore);</pre>
C syntax	<pre>int dpcmSetNoiseLimit(const DCM_STD_STRUCT *std_struct, T_Ignore *rtn, DCM_NoiseTypes noiseType, DCM_INTEGER libraryNoiseType, DCM_DOUBLE noiseLimit);</pre>

This function specifies an application budget for the specified noise type. The *noiseLimit* argument is specified in volts and represents the maximum allowable peak magnitude for that type of noise. This noise limit applies to all calculations for the specified *CellName* and *block* until overridden by another call for the same noise type, *CellName*, and *block*.

10.22.8 Reporting noise violations

When the library calculates the noise propagated across an interconnect or through a cell, it may detect one or more noise budget violations related to a particular pin. These violations shall be reported to the application by calling *appSetNoiseViolation*, which uses *noiseViolationInfo* structures to provide a summary of these violations. The application can request additional information about the noise components that contribute to a particular violation by calling *dpcmGetNoiseViolationDetails*.

10.22.8.1 noiseViolationInfo

Table 283 provides information on noiseViolationInfo.

Table 283—noiseViolationInfo

DCL syntax
<pre>typedef(noiseViolationInfo): result(noiseType: noiseType; int: libraryNoiseType; double: noiseLimit ; int: noiseLimitOrigin; double: violationMagnitude, violationArea);</pre>

C syntax

```
typedef enum DCM_NoiseLimitOrigin {
    DCM_LibraryNoiseLimit,
    DCM_AppNoiseLimit
} DCM_NoiseLimitOrigin;

typedef struct {
    DCM_NoiseTypes noiseType;
    DCM_INTEGER libraryNoiseType;
    DCM_DOUBLE noiseLimit;
    DCM_NoiseLimitOrigin noiseLimitOrigin;
    DCM_DOUBLE violationMagnitude;
    DCM_DOUBLE violationArea;
} DCM_NoiseViolationInfo;

typedef DCM_NoiseViolationInfo *DCM_NoiseViolationInfo_ARRAY;
```

A *noiseViolationInfo* structure provides a summary of a noise-budget violation. The *noiseType* and *libraryNoiseType* fields identify the type of noise for which the violation occurred.

The *noiseLimit* field represents the noise limit measured in volts and constrains the peak amplitude of the violation. If this limit was determined by the library, the *noiseLimitOrigin* field shall be set to *DCM_LibraryNoiseLimit*. If the limit was supplied by the application using *dpcmSetNoiseLimit*, the *noiseLimitOrigin* field shall be set to *DCM_AppNoiseLimit*.

The *violationMagnitude* field contains the peak noise value (in volts), whereas the *violationArea* field is the total area of the noise waveform for which the noise value exceeded *noiseLimit*.

10.22.8.1.1 appSetNoiseViolation

Table 284 provides information on *appSetNoiseViolation*.

Table 284—*appSetNoiseViolation*

Function name	<i>appSetNoiseViolation</i>
Arguments	Pin
Results	violations
Standard Structure fields	CellName, block, pathData, cellData
DCL syntax	<pre>external (appSetNoiseViolation): passed (pin: violatingPin; noiseViolationInfo: violationInfo) result (integer: ignore);</pre>
C syntax	<pre>int appSetNoiseViolation(DCM_STD_STRUCT *std_struct, T_Ignore *rtn, DCM_PIN violatingPin, DCM_NoiseViolationInfo_ARRAY *violationInfoArray);</pre>

This function describes violations detected by the library during noise calculation. The *violatingPin* argument shall be set to the input pin for violations detected during input-noise calculation and to the output pin for those found during output-noise computation. Several different types of noise violations may be detected during the same calculation, and *violationInfoArray* contains entries representing each of these violations.

10.22.8.1.2 dpcmGetNoiseViolationDetails

Table 285 provides information on *dpcmGetNoiseViolationDetails*.

Table 285—dpcmGetNoiseViolationDetails

Function name	dpcmGetNoiseViolationDetails
Arguments	None
Results	violations
Standard Structure fields	CellName, block, pathData, cellData
DCL syntax	external (dpcmGetNoiseViolationDetails) : result (noiseViolationInfo[*][*]: relatedPinDetails);
C syntax	<pre> typedef struct { DCM_NoiseViolationInfo ARRAY *noiseViolationInfoArray; } DCM_RelatedPinNoiseViolationDetails; typedef DCM_RelatedPinNoiseViolationDetails *DCM_RelatedPinNoiseViolationDetails_ARRAY; typedef struct { DCM_RelatedPinNoiseViolationDetails_ARRAY *relatedPinDetails; } T_NoiseViolationDetails; int dpcmGetNoiseViolationDetails (const DCM_STD_STRUCT *std_struct, T_NoiseViolationDetails *rtn); </pre>

Applications can call this function to obtain detailed information about the noise of various types contributed by each related pin to a violation reported via *appSetNoiseViolation*. If the application calls this function for a given violation, it shall do so before the call to *appSetNoiseViolation* through which that violation was reported returns.

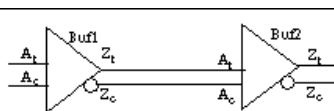
This function returns an array having one entry for each related pin, and the order of entries in this array shall be the same as the order of the related pins passed to *dpcmCalcInputNoise* or *dpcmCalcOutputNoise*. For each related pin, the array shall contain the values associated with that pin for each relevant noise type. The library can choose to provide either the original noise values as measured at each related pin or the noise values propagated from each related pin as measured at the violating pin.

10.23 Delay and slew calculations for differential circuits

In single-ended logic, a signal is measured at one point, with reference to *ground*. In contrast, a *differential signal* is composed of two complementary components, a “true” and a “complement” component, which are measured relative to each other.

10.23.1 Sample figures

Examples of differential signals and associated timing models are shown in Figure 7 through Figure 9.

**Figure 7—Differential buffer chain**

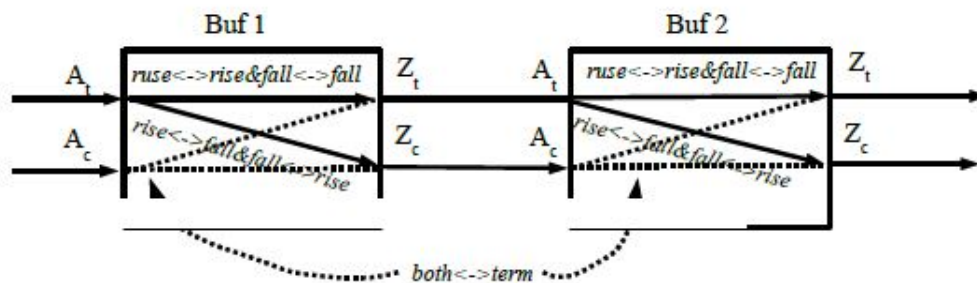


Figure 8—Timing models for a differential buffer chain

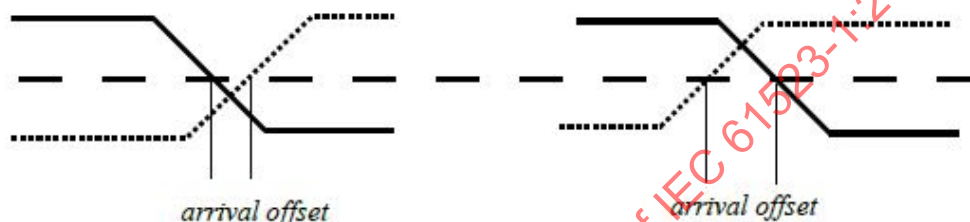


Figure 9—Arrival offset for differential signals

When a single-ended signal transitions, it is considered to have changed state when it passes through its threshold voltage. When one component of a differential signal transitions from a high to a low level and the other transitions in the opposite direction, the differential signal is considered to change state at the time these two transitions cross through the same voltage.

To compute the propagation delay of a differential signal transition accurately across a cell, the library shall compute the point in time at which its component signals cross each other. When viewed as individual signals, these components have arrivals at the times they cross through their respective threshold voltages. Using the offset distance between these arrival times together with the slews of these transitions (and the threshold and other associated voltages), the library can compute the time that the transitions intersect, and thus, the time at which the differential signal changes state. This is illustrated in Figure 9.

When responding to a call for delay or slew along a path from an input of a differential cell to one of the cell's outputs, the library can obtain arrival information for the corresponding complementary input from the application. The application shall return the arrival information for the transition at this complementary input that is nearest the transition for which the delay or slew was requested and that has the edge direction specified by the library.

The early and late arrival times returned by the application shall be relative to the early and late arrival times of the original input transition, respectively. If the application finds two arrival times of the complementary input to be equidistant from the early arrival time of the original input, it shall return the earlier of these two complementary input arrivals. Similarly, if two complementary input arrival times that are equidistant from the original input's late arrival are encountered, the application shall return the later of the two.

A negative offset represents the arrival time at the passed pin that precedes the reference arrival time. A positive offset represents an arrival time at the passed pin that follows the reference signal arrival time.

10.23.2 appGetArrivalOffsetsByName

Table 286 provides information on appGetArrivalOffsetsByName.

Table 286—appGetArrivalOffsetsByName

Function name	appGetArrivalOffsetsByName
Arguments	Offset pin name, Offset edge
Result	Early arrival time, Late arrival time, Early slew, Late slew
Standard Structure fields	block, CellName, fromPoint, sourceEdge, calcMode
DCL syntax	<pre>EXTERNAL (appGetArrivalOffsetsByName) : passed(string: offsetPinName; int: offsetEdge) result(double: earlyArrivalOffset, lateArrivalOffset, earlySlew, lateSlew);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE earlyArrivalOffset, lateArrivalOffset; DCM_DOUBLE earlySlew, lateSlew; } T_ArrivalOffsets; int appGetArrivalOffsetsByName (const DCM_STD_STRUCT *std_struct, T_ArrivalOffsets *rtn, const DCM_STRING offsetPinName, DCM_EdgeTypes offsetEdge);</pre>

This returns the arrival offsets and slews for the signal associated with the edge and pin name passed as the arguments back to the library.

One use of *appGetArrivalOffsetsByName* is to obtain the arrival offset for a differential input pair. However, the usage of this function is not restricted to differential input pairs. Any input pin from the same cell is a valid argument to this function.

10.23.2.1 appGetArrivalOffsetArraysByName

Table 287 provides information on appGetArrivalOffsetArraysByName.

Table 287—appGetArrivalOffsetArraysByName

Function name	appGetArrivalOffsetArraysByName
Arguments	Offset pin names, Offset edges
Result	Early arrival time, Late arrival time, Early slew, Late slew
Standard Structure fields	block, CellName, fromPoint, sourceEdge, calcMode
DCL syntax	<pre>EXTERNAL (appGetArrivalOffsetArraysByName) : passed(string[*]: offsetPinName; int[*]: offsetEdge) result(double[*]: earlyArrivalOffset, lateArrivalOffset, earlySlew, lateSlew);</pre>

C syntax	<pre>typedef struct { DCM_DOUBLE_ARRAY *earlyArrivalOffset, *lateArrivalOffset; DCM_DOUBLE_ARRAY *earlySlew, *lateSlew; } T_ArrivalOffsetArrays; int appGetArrivalOffsetArraysByName (const DCM_STD_STRUCT *std_struct, T_ArrivalOffsetArrays *rtn, const DCM_STRING_ARRAY *offsetPinNames, DCM_EdgeTypes *offsetEdges);</pre>

This returns arrays of arrival offsets and slews for the signal associated with the edge and pin name passed as the arguments back to the library.

One use of *appGetArrivalOffsetArraysByName* is to obtain the arrival offset for differential input pairs. However, the usage of this function is not restricted to differential input pairs. Any input pin from the same cell is a valid argument to this function.

10.23.3 API extensions for function modeling

The library conveys the structure of a static timing and power model to the application through the use of a graph containing the block, pin(s), and arc(s). The *block* is given as the container of the cell, each *pin* is a node in the graph, and the *arcs* are signal pathways. The boundary condition of the cell, including the external pins, are supplied by the application as a starting point for the library to begin its modeling process. This process continues and is expanded to cover function- and vector-specific requirements.

In general, each pin (external or internal) has a name. That name does not have to be unique, except under the following conditions: the names of the pins given as the inputs shall be unique within the input pin list; the names of the output pins shall be unique within the list of output pins. In fact, a bidirectional pin is a pin on both the output pin list and the input pin list which uses the same name. The internal pins generated by the library and given to the application are typically uniquely named, but this is not a requirement.

The paths between these pins are called “arcs” and have no name. These are generated by the application at the library’s request.

The library can send or cache additional information about an arc or a pin through the use of a pointer called the *pathData* pointer. Typically, all arcs generated by the library have a *pathData* pointer. This pointer shall be returned when the application requests a calculation from the library. The library typically uses these arc-specific path data pointers to cache arc-specific information for later use during a calculation.

The role of the nodes and arcs *have been expanded* to carry the function and vector information of a cell. Each function or vector shall be a graph of nodes connected by arcs. Each node represents a logical operation such as *AND*. Each arc contains information on function modeling, data types, and stranding. To accomplish this, the additional information is accessed through the *pathData* pointer. This results in internal nodes being created for function and vector graphs that have nonzero *pathData* pointer values associated with them.

10.23.3.1 Standard Structure extensions

The subsequent subclauses describe the extensions to the *Standard Structure* definition provided in IEEE Std 1481-1999.

10.23.3.2 Node representation

A function node is represented through the callback function *newTimingPin()*. This function instructs the application to create a new internal point or *node* in the current cell.

A *pathData* pointer present in the *Standard Structure* when *newTimingPin()* is called is associated with each node. The application, as part of the required operation for elaborating a function model, shall save this pointer for later use by the library. The *pathData* pointer shall point to a *PathDataBlock* structure with private (reserved for compiler use) and public (for use by an application) components. A zero-valued *pathData* pointer shall not be used in the behavior domain.

10.23.3.3 Path or arc representation

A function path or arc is represented through a callback API to the application. This function is *newPropagateSegment()* and instructs the application to create a path with the two pins or nodes supplied by the function. In contrast to its use in other domains, such as timing, the delay matrix handle supplied with this function shall be zero (0) when the latter is used to create a function arc. As for a node, a non-zero *pathData* pointer contained within the *Standard Structure* shall be associated with each path.

10.23.3.4 PathDataBlock data structure

Seven fields of the *PathDataBlock* data structure are utilized by compliant applications. They are all unsigned short integers, as follows: *cycle_adj* (the number of valid bytes after this field at the end of the *PathDataBlock*), *corrind* (the enumerated value for a node *PRIMITIVE* or arc *DATA_TYPE*), *modifiers* (the enumerated value for a node or arc *MODIFIER*), *msbStrandSource*, *lsbStrandSource*, *msbStrandSink*, and *lsbStrandSink* (these last four are for a strand sequence represented by an arc). For both arc- and node-based statements, such as *PATH*, *BUS*, *DO*, *INPUT* and *OUTPUT*, the first two fields (*corrind* and *modifiers*) shall be present if the *PRIMITIVE*, *DATA_TYPE* or *MODIFIERS* clauses are present in the *MODELPROC*. When none of the fields associated with the keywords are present, the *cycle_adj* field shall have a value of zero (0). When any one or more fields are missing, a default value shall be supplied for its representative field. The default values for the *DATA_TYPE*, *PRIMITIVE* and *MODIFIERS* values shall be zero (0). The default value for the *STRAND* fields shall be 0xFFFF. The contents of the *PathDataBlock* are as follows:

```

/*****
** PathDataBlock.
*****/
struct DCM_PathDataBlock {
/*****
** This information is intended for application use.
*****/
DCM_STRING path; /* path name under calculation */
/* this variable is set by the */
/* calculator during model build */
/* and can be ignored by other */
/* programs */
void **reserved1 /* reserved for the run-time linker */
/*****
** -> path constants data block.
** The PCDB contains both user and DCM control data that are constant
** for each PROPAGATE clause.
*****/
DCM_PCDB *pcdb;
/*****
** Items below this line are used by DCM for internal maintenance and
** consistency checking. DO NOT TOUCH!

```

```

*****/
int reserved2; /* reserved for the run-time linker */
short reserved3; /* reserved for the run-time linker */
/*****
** The first two are shared by TEST and functional statements.
*****/
short cycle_adj; /* TEST Cycle adjust, */
/* or functional byte count. */
short corrind; /* TEST Correlation index, */
/* or functional primitive. */
/*****
** The rest are for functional statements.
*****/
unsigned short modifiers; /* data type modifiers. */
unsigned short msbStrandSource; /* MSB src strand num for arc. */
unsigned short lsbStrandSource; /* LSB src strand num for arc. */
unsigned short msbStrandSink; /* MSB sink strand num for arc */
unsigned short lsbStrandSink; /* LSB sink strand num for arc */
};

```

10.23.3.5 Arc ordering

Some operations are very dependent on which operand appears on the left or right of the operator. Table 288 provides information on arc ordering. Table 289 describes the *modifiers* field in the *PathDataBlock* to distinguish the left operand from the right and provide a sequence number for user-defined types. This is only a requirement when arcs are propagated to nodes where the side of the operator where a signal appears is of concern (e.g., subtraction, division, or shifts). In the case of precedence operations, the *modifier* field shall represent the level of nesting to follow: zero (0) for the current level of nesting and one (1) for a new level of nesting.

Table 288—Arc ordering

Predefined data type modifiers	Enumeration
When the terminating node represents a dyadic operator	
DCM_MODIFIER_RIGHT_OPERAND	0x0000
DCM_MODIFIER_LEFT_OPERAND	0x0001
When the terminating node represents a mux, priority mux, or a priority storage node	
DCM_MODIFIER_CONTROL	0x4000 - 0x7FFF
DCM_MODIFIER_DATA	0x0000 - 0x3FFF
When the terminating node represents an @ operator	
DCM_MODIFIER_DATA	0x0000 - 0x3FFF

10.23.3.6 Priority operation

The *vector operator* is a single node that accepts two types of input arcs. Those input arcs that represent controlling signals have priority arcs with modifier values of 0x4000 - 0x7FFF and those that represent data have modifier values of 0x0000 - 0x3FFF. Both the priority and data are evaluated in the order of their respective *modifier* values, where the arc with the lowest modifier value is analyzed before the next higher value. The application shall start by analyzing the arc with the *modifier* value 0x4000 first, then 0x4001, and so on until the application encounters an arc that evaluates to *true*. The application next evaluates the data arc whose modifier value is equal to the offset from 0x4000 of the priority arc that evaluated *true*. This data arc is then set as the value of the vector node. If no priority arcs evaluate to *true*, the priority node shall remain at its previous value. The modifiers enumeration values are shown in Table 289.

Table 289—PathDataBlock->modifiers enumeration values for priority operation

Predefined data type modifiers	Enumeration
When the terminating node represents a mux, priority mux, or a priority storage node	
DCM_MODIFIER_CONTROL	0x4000 - 0x7FFF
DCM_MODIFIER_DATA	0x0000 - 0x3FFF
When the terminating node represents an @ operator	
DCM_MODIFIER_DATA	0x0000 - 0x3FFF

Selectors, priority mux, and priority storage element operators all behave in a similar manner. The difference is in the actual operation performed by the operator and the numerical relationships between the quantity of control arcs and data arcs, and whether or not the node containing the operator remembers its last state. This concept is illustrated in Figure 10.

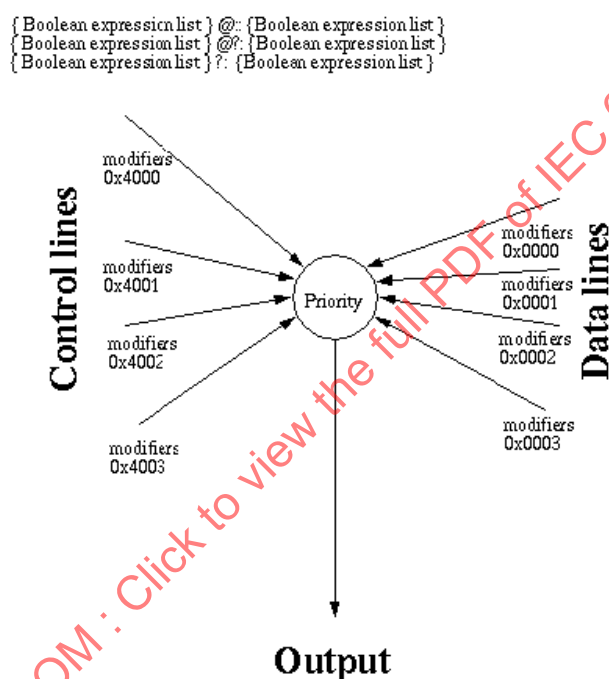


Figure 10—Priority operation

10.23.3.7 Precedence

Precedence is reserved for the control of large blocks where the priority node becomes too unwieldy. Precedence is a technique for representing scoped “if-then-else” chains, as shown in Figure 11.



The two types of Boolean assignments are *nonblocking*, which occur at the *FUNCTION* clause level of scope, and *blocking*, which occur within the scope of a precedence operator within the *FUNCTION* clause.

Nonblocking Boolean assignments in the *FUNCTION* clause scope shall be an arc from the source to the target. That arc shall have the *modifier* bit *0x8000*, indicating this is an *assignment arc*. This bit indicates to the application that this arc leads to a node whose value shall be saved as a temporary state until all the nonblocking assignments are completed for the cell.

Blocking Boolean assignments are identified by a node with an = primitive. These assignments are in control of the precedence operator where they are chained. The precedence operator has a conditional input that causes the assignment node where it is connected in this precedence operator to be evaluated in order or skips to the next *ELSE* in the else chain. The action list connected to each precedence node can contain a mixture of Boolean assignments or other precedence operators. Each element in the list is evaluated in the order encountered, as the list is traversed from the precedence node. When another precedence node is encountered, all its action nodes shall be evaluated before returning to process the next element on the action list.

10.23.3.8.3 Strand ranges

Multiwire arcs can be made up of one or more strands. Strands shall be numbered consecutively from the low strand number to the upper strand number. The least significant bit shall be represented by the *lsbStrand* member and the most significant bit shall be represented by the *msbStrand* member of the *PathDataBlock* structure. One set of strand members for the start of the arc (*lsbStrandSource* and *msbStrandSource*) and one set of strand members for the end of the arc (*lsbStrandSink* and *msbStrandSink*) exist on each *PathDataBlock* for paths that use the *sourceStrand* or *sinkStrand* words. For bit arrays, the most significant number shall be considered the sign bit for signed arithmetic. For floating point operations, the IEEE floating point arc shall have a bit pattern that conforms to the IEEE floating point standard for its width. (See ISO/IEC 9899:1990.)

The strand range at the start of an arc and the end of an arc can be different. However, the number of strands at the start of the arc shall match the number of strands at the end of the arc, as shown in Figure 12.

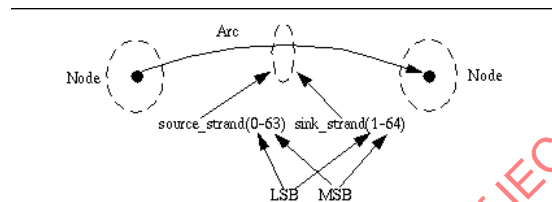


Figure 12—Strand ranges

10.23.3.8.4 Buses

The *bus range* (a continuous set of indices identifying the strands of the bus) at the start of an arc and the end of an arc can be different in strand assignments. However, the number of strands at the start of the arc shall match the number of strands at the end of the arc. Strand notation can be used to select a portion of a bus, switch the ordering of a bus, or rotate the bus. A strand notation of (*source_strands(0--31) sink_strands(31--0)*) reverses the bit ordering of the bus. The strand notation, combined with the *merge operator*node(><), can be used to swap bytes or portions of a bus.

Example

```
newBus = (source_strands<0 -- 15>) ><{
(source_strands(0 -- 7) sink_strands(8 -- 15)) data,
(source_strands(8 -- 15) sink_strands(0 -- 7)) data}
```

10.23.3.8.5 Fanout distributions

When a single point or group of points is routed to an entire bus, the source strands shall select the pattern bits and the sink strands shall determine the final bus configuration. For example, to save space, an integer constant of zero (0) can be represented from a single 0 bit that is fanned out to 32 bits (*source_strands(0--0) sink_strands(0--31)*). The source strands represent a bit pattern that is replicated until the sink strands bit width is satisfied.

Example

A two bit pattern of 0x2 can be fanned out to a width of 64 by using the strand notation:

```
(source_strands(0--1)sink_strands(0--63))
```

Results in 0xAAAAAAAAAAAAAAAA

To alter the final bit pattern to 0x5555555555555555, change the strand notation to the following:

```
(source_strands(1 -- 0) sink_strands(0 -- 63)) or
```

```
(source_strands(0 -- 1) sink_strands(63 -- 0))
```

10.23.3.8.6 Bundling

DCL treats each pin as an entity. The language represents a range of pins as a shorthand to the library developer that is expanded at run time. It is useful to group a range of pins and treat them as a single entity when using data flow operators. The semantics of the *dpcmGetCellIOlists()* have been extended to handle *bundling*. *dpcmGetCellIOlists()* returns three arrays of strings containing the valid pin names: one for the inputs, one for the outputs, and one for the bidirectional pins. The extension includes the bundle description for the *behavior*, *vectorTiming*, and *vectorPower* domains only. The bundle names are in a different name space than the actual pin names.

Example

A={A[0-128]} the valid pin names: one for the inputs, one for the outputs, and one for the bidirectional pins. The extension includes the bundle description for the *behavior*, *vectorTiming*, and *vectorPower*
B={B<63-0>,C<0-63>}

The *bundled pin name* is the identifier before the equal sign and the *bundle description* is contained within the {}. A single group of contiguously numbered pins shall be represented with pin range syntax. The range is delimited by [] or <>. The value on the left of the hyphen represents the LSB and the value on the right represents the MSB. When a bundle contains a comma-delimited list of pin names, the pin farthest to the left represents the LSB. All the pins to the right represent increasing strand bits until the ending delimiter is encountered. When an application communicates with the library, the only valid pin names are bitwise-expanded pin names of an unbundled list of the bundle name (those names to the left of the equal sign). It shall be considered an error if the application attempts to model a cell and supplies it with the individual strands of a bundle.

10.23.3.9 Extensions for retain modeling

An application determines if a particular arc models *retain* by looking at the following string field in the *Standard Structure*:

```
pathData->pcdb->objectType
```

This field shall return the string *'retain'* if, and only if, this path models *retain*. Any other value implies it is NOT *retain*. describes the method a library developer can use to indicate a particular timing arc is used for *retain*.

10.23.3.10 Extensions for skew testing

The *DIFFERENTIAL_SKEW* test mode operator specifies the edge identified as the source shall occur within a window of time before or after the edge identified as the reference. The bias sets the magnitude of the window. The *BIAS* value shall be positive and indicate the signal can vary in time before or after the reference by the amount of the bias. The use of this test mode operator with the *EARLY* test mode operator indicates the test is to be performed on the early edges of both the signal and reference. The use of this test mode operator with the *LATE* mode operator indicates the test is to be performed on the late edges of both the signal and reference. See Table 290 for information on DCM_TestTypes.

Table 290—DCM_TestTypes

Enumerator	Enumeration	Description
DCM_SetupTest	0	DCM Setup test.
DCM_HoldTest	1	DCM hold test.
DCM_ClockPulseWidthTest	2	DCM clock pulse width test.
DCM_ClockSeparationTest	3	DCM clock separation test.
DCM_DataPulseWidthTest	4	DCM data pulse width test.
DCM_DataSeparationTest	5	DCM data separation test.
DCM_ClockGatingPulseWidthTest	6	DCM clock gating pulse width test.
DCM_ClockGatingHoldTest	7	DCM clock gating hold test.
DCM_ClockGatingSetupTest	8	DCM clock gating setup test.
DCM_EndOfCycleTest	9	DCM end of cycle test.
DCM_DataHoldTest	10	DCM data hold test.
DCM_RecoveryTest	11	DCM recovery test.
DCM_RemovalTest	12	DCM removal test.
DCM_SkewTest	13	DCM skew test.
DCM_NoChangeTest	14	DCM no change test.
DCM_DifferentialSkewTest	15	DCM differential skew test.

10.23.4 Explicit APIs for user-defined primitives

The next subclauses specify additional explicit *EXPOSE* and *EXTERNAL* functions required for user-defined primitives.

10.23.4.1 dpcmPerformPrimitive

Table 291 provides information on dpcmPerformPrimitive.

Table 291—dpcmPerformPrimitive

Function name	dpcmPerformPrimitive
Arguments	None
Result	Array of bits for an arc
Standard Structure fields	block, CellName, cellData, pathData (behavior-node-specific)
DCL syntax	<i>EXPOSE</i> (dpcmPerformPrimitive): result(int[*]: outputPattern);
C syntax	<pre>typedef struct { DCM_INTEGER_ARRAY *outputPattern; } T_PerformPrimitive; int dpcmPerformPrimitive (const DCM_STD_STRUCT *std_struct, T_PerformPrimitive *rtn);</pre>

When an application encounters a user-defined node, the application shall call the library *EXPOSE* function *dpcmPerformPrimitive*, which passes a single dimensional array of integers, each representing a bit of the primitive's value, back to the application. Each integer array element represents an enumeration value corresponding to the pin state.

10.23.4.2 appGetArcStructure

Table 292 provides information on appGetArcStructure.

Table 292—appGetArcStructure

Function name	appGetArcStructure
Arguments	Sequence number
Result	Array of bits for an arc, Arc data type token, Arc ending strand range
Standard Structure fields	block, CellName, cellData, pathData
DCL syntax	<pre>EXTERNAL (appGetArcStructure) : passed(int: sequenceNumber) result(int[*]: bitPattern; int: arcDataType, endingStrandRangeLsb, endingStrandRangeMsb);</pre>
C syntax	<pre>typedef struct { DCM_INTEGER_ARRAY *bitPattern; DCM_INTEGER arcDataType; DCM_INTEGER endingStrandRangeLsb; DCM_INTEGER endingStrandRangeMsb; } T_ArcStructure; int appGetArcStructure(const DCM_STD_STRUCT *std_struct, T_ArcStructure *rtn, DCM_INTEGER sequenceNumber);</pre>

During the library's processing of the user-defined primitive, it can call back the application for information about the individual arcs propagating to the node under analysis. This includes the actual bit pattern present at the time of the call, as well as the data type and any strand information.

10.23.4.3 dpcmGetNodeSensitivity

Table 293 provides information on dpcmGetNodeSensitivity.

Table 293—dpcmGetNodeSensitivity

Function name	dpcmGetNodeSensitivity
Arguments	Sequence number
Result	Array of state strings
Standard Structure fields	block, CellName, cellData, pathData
DCL syntax	<pre>EXPOSE (dpcmGetNodeSensitivity) : passed(int: sequenceNumber) result(string[*]: stateArray);</pre>
C syntax	<pre>typedef struct { DCM_STRING_ARRAY *arcArray; } stateStructure; int dpcmGetNodeSensitivity (const DCM_STD_STRUCT *std_struct, struct stateStructure *rtn, DCM_INTEGER sequenceNumber);</pre>

To minimize the number of calls to *dpcmPerformPrimitive* (see 10.23.4.1), the application can make a call to *dpcmGetNodeSensitivity*. This returns a list of strand states that shall be monitored. A change to any of the listed states requires the application to call *dpcmPerformPrimitive*. To conform to the naming conventions, the strand numbers shall be immediately preceded by the character “s.”

10.23.5 APIs for hierarchy

Hierarchy can be implicitly implemented using *SUBMODEL* statements. However, with this type of implementation, the application does not have access to information regarding the hierarchical structure. Thus, hierarchy for this specification is defined as the act of an application calling the library for additional detail for the hierarchy of a cell.

10.23.5.1 Direct callback base hierarchy

Direct callback base hierarchy allows for full hierarchical nesting (i.e., when one level of hierarchy is expanded, it can contain other hierarchical nodes). The application can call the DPCM for an expansion of this hierarchical node by calling *dpcmModelMoreFunctionDetail*. The result of calling this function is a more detailed function graph replacing the lesser detailed subgraph at the higher level of hierarchy.

A hierarchical node is identified as a node containing a primitive value of *USER_DEFINED_MACRO*. The application shall use the list of arc sources feeding the hierarchical node and the list of arc sinks leaving the hierarchical node to populate a Standard Structure, where the list of arc sources represent the input list and the list of arc sinks represent the output list, respectively.

10.23.5.1.1 dpcmModelMoreFunctionDetail

Table 294 provides information on *dpcmModelMoreFunctionDetail*.

Table 294—dpcmModelMoreFunctionDetail

Function name	dpcmModelMoreFunctionDetail
Arguments	None
Result	Boolean indicating detail was returned
Standard Structure fields	block, CellName, cellData, pathData (behavior-node-specific), inputPins, outputPins, inputPinCount, outputPinCount
DCL syntax	EXPOSE(dpcmModelMoreFunctionDetail): result(int: detailProvided);
C syntax	typedef struct { DCM_INTEGER detailProvided; } T_DetailProvided; int dpcmModelMoreFunctionDetail (const DCM_STD_STRUCT *std_struct, T_DetailProvided *rtn);

This causes the library to model the behavior of the node identified by the *pathData* field in the Standard Structure with a greater level of detail (than previously requested by the application). The application shall create a new Standard Structure and supply it to *dpcmModelMoreFunctionDetail*.

A returned value of 1 for *detailProvided* indicates additional detail was provided by the library. Otherwise, a value of 0 shall be returned.

10.23.6 Built-in APIs for function modeling

Several new built-in functions have been added to ease the locating of input pins, output pins, and nodes.

10.23.6.1 LOCATE_INPUT

The *LOCATE_INPUT* built-in function takes a string expression as an argument and searches the input pin list supplied by the application for a matching pin. If the pin is found, the pin handle is returned through the default result. If not, *NULL* is returned.

10.23.6.2 LOCATE_OUTPUT

The *LOCATE_OUTPUT* built-in function takes a string expression as an argument and searches the output pin list supplied by the application for a matching pin. If the pin is found, the pin handle is returned through the default result. If not, *NULL* is returned.

10.23.6.3 LOCATE_NODE

The *LOCATE_NODE* built-in function takes a string expression as an argument and searches the node list created and maintained by the library for a matching node. If the node is found, the node's pin handle is returned through the default result. If not, *NULL* is returned.

10.23.7 API Extensions for VECTOR modeling

This subclause defines the additional APIs used to model vectors.

10.23.7.1 Vector domains

Libraries that utilize VECTOR clauses to model sequences of transitions shall do so in the “vectorTiming” or “vectorPower” domains.

10.23.7.2 Standard Structure fields for VECTOR clause in vectorTiming domain

When the *VECTOR* node is created, the Standard Structure shall also have the *fromPoint* and *toPoint* fields set. The application shall inspect the *PathDataBlock* associated with a node and, if this *pathData* is for a *VECTOR* node, shall save these fields. When the application calls for *delay*, *slew*, *check*, or other statements that require the *PathDataBlock*, the application shall update the *fromPoint* and *toPoint* members in the Standard Structure. The actual value of the member does not have to be identical, but rather the equivalent information shall be conveyed (i.e., the *fromPoint* and *toPoint* are not required to be persistent, but their content shall be reproduced).

10.23.7.3 VECTOR clause for delay and slew

During the phase when the application models and elaborates cells in the library, the library shall initiate callbacks to the application indicating the presence of *VECTOR* clauses for delay and slew of a given timing arc. This is done via calling *newNetSinkPropagateSegments()* and *newDelayMatrixRow()*, and getting the value of the *primitive* member in the *PathDataBlock*. Edge information is transferred to the application by *newDelayMatrixRow()* calls. To convey the pin name associated with the vector delay, the *newNetSinkPropagateSegment()* call shall have the vector node as the *importPin* argument. The *sinkPin* argument shall have a value of *NULL*.

The application shall determine whether the vector node has delays associated with it based on the value of the *primitive* member. The *primitive* value 0x0090 identifies this pin as having propagation delays associated with it. The application shall determine the edges of the vector being analyzed from the delay matrix row contents, the vector expression, and the function expression. To request a delay, the application shall call the standard delay function initializing the *Standard Structure*, where the *fromPoint* member and the *toPoint* member are the same as returned in the *Standard Structure* during the call to *newNetSinkPropagateSegments()*.

10.23.7.4 VECTOR clause for timing check

The library callbacks for vector timing checks are identical to those for the vector delay and slew. If the value of the *corrind* member is *0x0091*, this indicates a timing check value is specified in the library. The type of test (e.g., *setup*, *hold*, etc.) is indicated by the value of the *modifier's* member of the *PathDataBlock*. The type of test indicated in the *modifier's* member is the enumeration returned by the implicit callback function *newTestMatrixRow()*. The clock or reference pin shall be placed in the *fromPoint* field of the *Standard Structure* and the data or signal pin shall be placed in the *toPoint* field.

To determine a test time offset or bias, the application shall call the implicit check function. The *fromPoint* and *toPoint* fields shall be set to the values returned by *NetSinkPropagateSegments()*.

10.23.7.5 Vector target node generation

Vector target nodes are nodes that carry the information that allows the application to associate a vector expression graph with the actions required to query the dpcm for delay and check (*0x0090* and *0x0091*). The vector target node is created in a two-step process, as follows:

- The library issues a call to *newTimingPin* with three arguments: a *Standard Structure*, a string representing the node's name, and a pointer to where the application can put its newly created pin handle. The *pathData* pointer in the *Standard Structure* argument shall be zero (NULL). In response to this call, the application shall create a node with a name that is the same as the string argument.
- The library then calls *newNetSinkPropagateSegments* as described in 10.23.7.3 and 10.23.7.4. When this call is made to the application, the *pathData* field of the *Standard Structure* shall contain the vector target information and the application shall associate the *pathData* in the *Standard Structure* with the node created on the previous call.

10.23.8 APIs for XWF

Traditional slew measurements rely on measuring the delta time between two points on a waveform (e.g., 10% and 90% measuring points between ground and vdd), which results in a linear representation. To provide more accuracy, a library developer-specified definition of XWFs for the analysis of slew, delay, and so on may be used. XWF is a proprietary library vendor mechanism the library vendor uses to model slews. Thus, slews no longer need be only linear. Slews can be described using three points, six points, or even a mathematical representation, such as an equation (see Figure 13). XWFs are *DCM_STRUCTs* that occupy library allocated memory that correspond to the slew waveform representation chosen by the library developer.

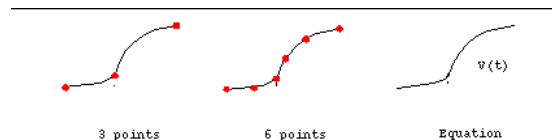


Figure 13—Various methods of using XWF to model waveforms for slew computation

The application is not privy to the contents of the library proprietary XWF data. However, the library vendor is free to publish the format and representation of the contents of an XWF.

An application simply manages handles or pointers to library-calculated output slews for cells and interconnect during EDA tool interaction with a library that contains XWF representation. The interaction proceeds as follows: The application requests the library compute output slew based on an input slew (for a timing arc within a cell). The library shall then request the handle or pointer to an XWF for the input slew of that cell.

If the application does not have the requested XWF handle or pointer (this occurs when this is a primary timing input or chip input, i.e., the beginning node of a timing path), the library shall translate the two-point ramp slew the application provides to an appropriate XWF. This two-point ramp slew is provided in the Standard Structure as *slew->early* and *slew->late*. This shall be done by allocating memory for the XWF representation and then passing a pointer to the resulting output XWF back to the application.

Once this operation has finished, the application has a handle to the output XWF for this cell. When the application requests the output slew (either interconnect or cell) for the next stage of this timing path, the application shall provide the library with the output XWF that was generated by the library (see Figure 14). During delay and slew calculation, the application shall implicitly be given a two point (piecewise) slew translation of the internal XWF representation. The slew values so obtained shall be used for determining slew limits and so on.

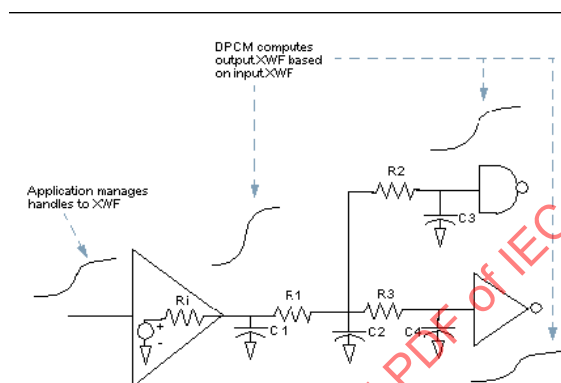


Figure 14—Propagation of XWF “handles” by application

The XWF definition contains in the two new APIs as shown in Table 296. These APIs provide the library developer with the flexibility of defining the XWF in any format. They are also external and are not accessible by the application.

Table 295—XWF APIs

New API	Description
<i>appGetXWF()</i>	Retrieving XWF from the application.
<i>appSetXWF()</i>	Setting the computed XWF for an arc in the application.

The application shall compute delay, slew and timing checks with the implicit *delayFunction()*, *slewFunction()* and *checkFunction()* calls. The library, during the calculation of the requested data, shall request the application save pointers to the XWF data structure associated with a particular computation.

The application shall continue to use the slew values computed by the library for comparison and reporting purposes.

The slews and the XWF data computed by the library for the output of a given propagation arc shall be correlated in their effects on subsequent timing arcs in a circuit path. If the application selects a representative slew value from the output slews of multiple arcs that converge at a pin, it shall continue to do so based on comparisons of the slew values themselves. Once this selection is made, the application shall return the XWF data from the same arc where it obtained this representative slew value (in response to requests from the library for the XWF at that pin).

10.23.8.1 XWF definition

The library developer shall designate the XWF definition; it shall contain any information the developer deems necessary. The application shall not attempt to understand the contents of the library-specific XWF data structure. To accomplish this, the XWF definition shall be passed and returned between the library and the application as a *DCM_STRUCT* pointer. The library vendor shall create the XWF definition using the *DCM_STRUCT* type. The library is also responsible for allocating the space needed for the structure and its components.

10.23.8.2 Freeing XWF memory allocation

The application is responsible for freeing all XWF data structure representations returned to it by the library. This can be accomplished in one of two ways:

- Do nothing at all. The next time the library is entered at level zero, the space is automatically released back to the memory pool.
- The application can first lock the XWF data structure representation, forcing it to remain until it is eventually unlocked. When the application no longer has any use for that instance of the XWF data structure representation, it unlocks it. The mechanism for this is described in 10.25.2.

This simple mechanism minimizes the need for the application to understand the internal organization of the XWF data structure representation, reduces the need for unwanted copying, and removes the additional callbacks to free the space.

10.23.8.3 XWF API definitions

The XWF APIs are defined in the next subclauses.

10.23.8.3.1 appSetXWF

Table 297 provides information on appSetXWF.

Table 296—appSetXWF

Function name	appSetXWF
Arguments	Early and late XWF data structures
Result	None
Standard Structure fields	CellName, toPoint, calcMode, block, sinkEdge, pathData (timing-arc- or pin-specific), cellData (timing)
DCL syntax	EXTERNAL (appSetXWF) : passed(void: earlyXWF, lateXWF) result(int: ignore);
C syntax	typedef void DCM_STRUCT; typedef struct { DCM_INTEGER ignore; } T_Ignore; int appSetXWF(const DCM_STD_STRUCT *std_struct, T_Ignore *rtn, DCM_STRUCT *earlyXWF, DCM_STRUCT *lateXWF);

The library calls this to store an XWF value in the application. This action shall occur only if the library developer has implemented the library with an XWF data structure representation.

The library shall call *appSetXWF* during slew computations initiated by the application. This API informs the application of the XWF data structure for the specific slew computation requested by the application.

This function sets computed XWF data structures for the pin, edge, and timing arc as identified by the *toPoint*, *sinkEdge*, and *pathData* fields in the Standard Structure. The application shall record the XWF pointers for subsequent use. The library developer shall call this function during computation of slew values before the function that calculates the slews (generated from a DCL *SLEW* statement) exits. The Standard Structure shall remain unmodified during this operation.

10.23.8.3.2 appGetXWF

Table 297 provides information on appGetXWF.

Table 297—appGetXWF

Function name	appGetXWF
Arguments	None
Result	Early and late XWF data structure
Standard Structure fields	CellName, fromPoint, calcMode, block, sourceEdge, pathData (timing-arc- or pin-specific), cellData (timing)
DCL syntax	EXTERNAL(appGetXWF): result(void: earlyXWF, lateXWF);
C syntax	<pre>typedef void DCM_STRUCT; typedef struct { DCM_STRUCT *earlyXWF; DCM_STRUCT *lateXWF; } T_XWF; int appGetXWF(const DCM_STRUCT *std_struct, T_XWF *rtn);</pre>

The library calls this to retrieve the XWF from the application. This action shall occur only if the library developer has implemented the library with an XWF data structure representation.

The library shall call this function if an application requests the calculation of the delay, slew, or check values for an arc. The application shall return the XWF data structures obtained from a previous call to *appSetXWF* (which were applied to the pin at the beginning of this arc) back to the library. The application shall return a nonzero return code if the application is unable to honor the XWF computation by the library. If the application does not implement this API, the library developer is responsible for determining the default action.

When computing an XWF for an interconnect arc, the library is unaware of whether the computed XWF is associated with the *tech_family* of the driver or the receiver, which can be different. In those situations where the driver's *tech_family* is different from that of the receiver's *tech_family*, the application shall not perform the requested *appSetXWF* call and also shall not return an error to the library.

10.23.8.3.3 dpcmCalcXWF

Table 298 provides information on dpcmCalcXWF.

Table 298—dpcmCalcXWF

Function name	dpcmCalcXWF
Arguments	Pin pointer, Edge, Early and late slews
Result	Early and late XWF data structures
Standard Structure fields	CellName, calcMode, pathData (timing-pin-specific), cellData (timing), processVariation
DCL syntax	<pre>EXPOSE(dpcmCalcXWF): passed(pin: pinPointer; int:edge; double: earlySlew, lateSlew) result(void: earlyXWF, lateXWF);</pre>
C syntax	<pre>typedef struct { DCM_STRUCT *earlyXWF; DCM_STRUCT *lateXWF; } T_XWF; int dpcmCalcXWF (const DCM_STD_STRUCT *std_struct, T_XWF *result, DCM_PIN pinPointer, DCM_EdgeTypes edge, DCM_DOUBLE earlySlew, DCM_DOUBLE lateSlew);</pre>

This function returns pointers to XWF structures for the passed pin and edge direction corresponding to the early and late slews provided. The edge direction shall be one of the following:

- DCM_RisingEdge
- DCM_FallingEdge
- DCM_OneToZ
- DCM_ZtoOne
- DCM_ZeroToZ
- DCM_ZtoZero

The application shall call this function when the library has requested XWF structures for a pin for which no appropriate XWF data have been set by the library (see 10.23.8.1). The application shall record the XWF pointers returned by this function for subsequent use.

This API allows the application to obtain XWF data for pins, such as primary inputs for which a library would otherwise never create such data. This function is also useful when XWF data are requested for an aggressor driver for which it has not yet been computed during crosstalk analysis.

10.23.9 Extensions and changes to voltages and temperature APIs

This subclause augments the APIs used to gather voltage and temperature information.

10.23.9.1 dpcmGetCellRailVoltageArray

Table 299 provides information on dpcmGetCellRailVoltageArray.

Table 299—dpcmGetCellRailVoltageArray

Function name	dpcmGetCellRailVoltageArray
Arguments	None
Result	Rail array
Standard Structure fields	block, CellName, cellData (timing)
DCL syntax	EXPOSE(dpcmGetCellRailVoltageArray): result(string[*]: railArray);
C syntax	<pre>typedef struct { DCM_STRING_ARRAY *railVoltageArray; } T_RailArray; int dpcmGetCellRailVoltageArray (const DCM_STD_STRUCT *std_struct, T_RailArray *rtn);</pre>

This returns voltage rail names for the cell identified by the *cellData* field in the Standard Structure back to the application.

Different cells in a library can contain rail voltages having the same name but referring to different voltage rails. This function differs from *dpcmGetRailVoltageArray* as it returns rail voltage names for the specified cell only, not an entire library. If *dpcmGetCellRailVoltageArray* is exposed by a DPCM, the application shall use it instead of, and shall not call, *dpcmGetRailVoltageArray*.

If *dpcmGetCellRailVoltageArray* is exposed, the DPCM shall expose the function *dpcmGetBaseCellRailVoltageArray* instead of *dpcmGetBaseRailVoltage* and the function *dpcmGetCellRailVoltageRangeArray* shall be exposed instead of *dpcmGetRailVoltageRangeArray*.

The application shall be free to assume a rail voltage name from different cells refers to the same voltage rail if all of the following conditions are met:

- The name of the voltage rail is the same for each cell.
- The default value associated with the voltage rail (obtained via *dpcmGetBaseCellRailVoltageArray*) is the same for each cell.
- The range of legal values for the voltage rail (obtained via *dpcmGetCellRailVoltageRangeArray*) is the same for each cell.

The library shall return a zero-length array for a cell where voltage effects are not modeled.

10.23.9.2 dpcmGetBaseCellRailVoltageArray

Table 300 provides information on dpcmGetBaseCellRailVoltageArray.

Table 300—dpcmGetBaseCellRailVoltageArray

Function name	dpcmGetBaseCellRailVoltageArray
Arguments	None
Result	Rail voltages
Standard Structure fields	block, CellName, cellData (timing), calcMode
DCL syntax	EXPOSE(dpcmGetBaseCellRailVoltageArray): result(double[*]: railVoltages);

C syntax	<pre>typedef struct { DCM_DOUBLE_ARRAY *railVoltages; } T_RailVoltageArray; int dpcmGetBaseCellRailVoltageArray (const DCM_STD_STRUCT *std_struct, T_RailVoltageArray *rtn);</pre>

This returns the default values for the voltage rails of the cell identified by the *cellData* field in the Standard Structure back to the application. These values can vary for different operating ranges and calculation modes.

This function differs from *dpcmGetBaseRailVoltage*, as it returns all of the default rail voltage values for the specified cell only, not the default value for a single rail voltage for an entire library. If this function is exposed by a DPCM, the application shall use it instead of, and shall not call, *dpcmGetBaseRailVoltage*. If the function *dpcmGetCellRailVoltageArray* is exposed by the DPCM, this function shall also be exposed.

The order of the values returned by this function shall be the same as the order of the voltage rail names returned by *dpcmGetCellRailVoltageArray* for the same cell.

10.23.9.3 dpcmGetBaseCellTemperature

Table 301 provides information on *dpcmGetBaseCellTemperature*.

Table 301—*dpcmGetBaseCellTemperature*

Function name	<i>dpcmGetBaseCellTemperature</i>
Arguments	None
Result	Temperature
Standard Structure fields	block, CellName, cellData (timing), calcMode
DCL syntax	<pre>EXPOSE (dpcmGetBaseCellTemperature) : result(double: temperature);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE temperature; } T_BaseTemperature; int dpcmGetBaseCellTemperature (const DCM_STD_STRUCT *std_struct, T_BaseTemperature *rtn);</pre>

This returns the default temperature value for the cell identified by the *cellData* field in the Standard Structure back to the application. This value can be different for different operating ranges and calculation modes.

This function differs from *dpcmGetBaseTemperature*, as it returns a temperature value for the specified cell only, not an entire library. If this function is exposed by a DPCM, the application shall use it instead of, and shall not call, *dpcmGetBaseTemperature*.

10.23.10 Operating conditions

This subclause contains the API modifications needed to support a more robust handling of the operating conditions for a chip. The next subclauses address the topics of on-chip process variation, PVT limits, and operating points.

10.23.10.1 Operating points

Included in this standard is the concept of *operating ranges* to represent ranges of PVT over which a classification of chips shall operate. Once an application specifies an operating range, default values for process point, rail voltages, and temperature reflect that setting.

It is also desirable to identify a specific process point or a PVT combination for which calculations shall be performed by a library. For this purpose, the concept of an operating point is defined. An *operating point* is composed of a base process point and, optionally, a set of base rail voltages, a base temperature, or both. These base values shall be returned to the application in response to calls to the functions *dpcmGetBaseProcessPoint* (see 10.23.11.1.2), *dpcmGetBaseRailVoltage*, *dpcmGetBaseTemperature*, *dpcmGetBaseCellRailVoltageArray* (see 10.23.9.2), or *dpcmGetBaseCellTemperature* (see 10.23.9.3).

Each operating point shall be identified by its name. The names of the operating points and the combination of base PVT values associated with each operating point shall be defined within the library. An operating point shall be used by the library in a consistent manner across all modeling domains. Once an operating point has been set by the application, the base process point in that operating point shall be used for all library calculations performed in each domain.

If an operating point does not include rail voltages or a temperature, the library shall obtain them from the application using the interface functions *appGetCurrentRailVoltage()* or *appGetCurrentTemperature()*.

The library can choose to include perturbations about an operating point's base PVT values in the early and late results calculated for that operating point.

10.23.10.1.1 dpcmGetOpPointArray

Table 302 provides information on *dpcmGetOpPointArray*.

Table 302—dpcmGetOpPointArray

Function name	dpcmGetOpPointArray
Arguments	None
Result	Array of operating points
Standard Structure fields	None
DCL syntax	<pre>EXPOSE(dpcmGetOpPointArray): result(string[*]: opPointArray);</pre>
C syntax	<pre>typedef struct { DCM_STRING_ARRAY *opPointIndex; } T_OpPointArray; int dpcmGetOpPointArray(const DCM_STD_STRUCT *std_struct, T_OpPointArray *rtn);</pre>

The application can obtain the names of the operating points within a library using this function.

If the library supports multiple operating ranges, the set of operating points returned shall be associated with the current operating range in effect at the time of the call. Different operating ranges can have separate sets of operating points.

10.23.10.1.2 dpcmGetBaseOpPoint

Table 303 provides information on *dpcmGetBaseOpPoint*.

Table 303—dpcmGetBaseOpPoint

Function name	dpcmGetBaseOpPoint
Arguments	None
Result	Operating point index
Standard Structure fields	None
DCL syntax	EXPOSE(dpcmGetBaseOpPoint): result(integer: opPointIndex);
C syntax	typedef struct { DCM_INTEGER opPointIndex; } T_OpPointIndex; int dpcmGetBaseOpPoint(const DCM_STD_STRUCT *std_struct, T_OpPointIndex *rtn);

The application can use this to obtain the index for the default operating point.

If the library does not support distinct operating points, a value of -1 shall be returned as the index. Otherwise, an index number between 0 and $n - 1$, inclusive (where n is the number of operating points) shall be returned. This number shall be used as an index into the array returned by *dpcmGetOpPointArray* to obtain the name of the default operating point.

10.23.10.1.3 dpcmSetCurrentOpPoint

Table 304 provides information on dpcmSetCurrentOpPoint.

Table 304—dpcmSetCurrentOpPoint

Function name	dpcmSetCurrentOpPoint
Arguments	Operating point index
Result	Old operating point index
Standard Structure fields	None
DCL syntax	EXPOSE(dpcmSetCurrentOpPoint): passed(int: opPointIndex) result(int: oldOpPointIndex);
C syntax	typedef struct { DCM_INTEGER opPointIndex; } T_OpPointIndex; int dpcmSetCurrentOpPoint (const DCM_STD_STRUCT *std_struct, T_OpPointIndex *rtn, DCM_INTEGER opPointIndex);

The application shall use this to set an operating point.

The operating point index passed to this function shall be between 0 and $n - 1$, inclusive, where n is the number of operating point names returned by *dpcmGetOpPointArray*.

If this function is called successfully, the library shall perform all subsequent calculations using the process point and any rail voltages or temperature associated with the specified operating point. These values shall then be returned to the application in response to calls to the interface functions *dpcmGetBaseProcessPoint* (10.23.11.1.2), *dpcmGetBaseRailVoltage*, *dpcmGetBaseTemperature*, *dpcmGetBaseCellRailVoltageArray* (10.23.9.2), or *dpcmGetBaseCellTemperature* (10.23.9.3).

10.23.11 On-chip process variation

The concept of *early* and *late* times are the first and last possible times a signal transition can arrive at a given point in a circuit. Corresponding to these times are *early* and *late* delays and slews for a given propagation arc. These different values can correspond to different physical paths between the two end points of the arc (e.g., between an input and an output pin on an instance).

The concept of early and late values shall be extended to include on-chip process variation. The additional terms *min* and *max* have been added. These terms shall be used to represent the minimum and maximum, respectively, of the uncertainty in a calculated value due to on-chip process variation. These values shall be relative to a particular process point representing a chip that can be produced using a given process technology.

Use of *min* process variation during timing calculations results in shorter delays and faster slews. Conversely, use of *max* process variation yields longer delays and slower slews.

A new enumerated type is also added to for specifying how the effects of on-chip process variation shall be included in early and late value calculation. This enumerated type is shown in Table 305.

Table 305—DCM_ProcessVariations

Enumerators	Enumeration	Description
DCM_NoVariation	0	No process variation.
DCM_MinEarly_MaxLate	1	On-chip minimum reflected in early values and on-chip maximum reflected in late values.
DCM_MaxEarly_MinLate_EdgesSame	2	On-chip maximum reflected in early values and on-chip minimum reflected in late values in a manner suitable for comparison with values calculated using <i>DCM_MinEarly_MaxLate</i> for a timing segment common to two circuit paths that traverse the segment using the same edge directions.
DCM_MaxEarly_MinLate_EdgesOpposite	3	On-chip maximum reflected in early values and on-chip minimum reflected in late values in a manner suitable for comparison with values calculated using <i>DCM_MinEarly_MaxLate</i> for a timing segment common to two circuit paths that traverse the segment using opposite edge directions.

The first enumerator *DCM_NoVariation* (which has an integer value of 0) is used to indicate process variation and shall not be included in early and late values.

A *DCM_ProcessVariations* value shall be included in the *Standard Structure* passed from the application to an library. The library shall use this information to determine how the effects of on-chip process variation are reflected in calculated values for delay and slew.

The identifiers *PROCESS_VARIATION*, *process_variation*, *PROCESS_VARIATION_SCALAR*, and *process_variation_scalar* are required for implementation of on-chip process variation. These identifiers can be set to the values shown in Table 306.

Table 306—New predefined identifiers

New predefined identifier	Value	Meaning
PROCESS_VARIATION <i>process_variation</i>	NoVariation	No variation
	MinEarly_MaxLate	Min early, max late
	MaxEarly_MinLate_EdgesSame	Max early, min late, edges same
	MaxEarly_MinLate_EdgesOpposite	Max early, min late, edges opposite
PROCESS_VARIATION_SCALAR <i>process_variation_scalar</i>	0	No variation
	1	Min early, max late
	2	Max early, min late, edges same
	3	Max early, min late, edges opposite

All *EXTERNAL* and *EXPOSE* API functions that require the *calcMode* field in the *Standard Structure* shall also require the *processVariation* field.

When an application compares the timing between two paths during the evaluation of a timing check, it often uses the early timing values from one path and the late values from the other.

To produce worst-case timing results when on-chip process variation is considered, the minimum variation is included in the early values and the maximum variation in the late values. This shall be done by the library when the application sets the *processVariation* field in the *Standard Structure* to *DCM_MinEarly_MaxLate*.

For certain circuit configurations, this can produce a result that is unduly pessimistic. For example, when two paths share a common set of arcs, the effects of on-chip process variation on these arcs should be less when these two paths are compared than if either were compared with a completely unrelated path.

To take into account the *common ambiguity* of these arcs due to on-chip process variation without changing its basic approach to path comparison, the application can instruct the library to include these effects differently for the common arcs when the timing for one of these paths is calculated.

If the signals propagated along both paths pass through a common arc with the same edge polarities (rising or falling) at each end, the application can set the *processVariation* field to *DCM_MaxEarly_MinLate_EdgesSame* when the timing of that arc is calculated for one of the paths. This instructs the library to include the maximum on-chip process variation in the early timing values and the minimum in the late values, in a manner such that excess on-chip variation is canceled out when the results are compared with those of the other path (for which a *processVariation* value of *DCM_MinEarly_MaxLate* was used). The library determines the amount of variation that is canceled.

Similarly, a *processVariation* value of *DCM_MaxEarly_MinLate_EdgesOpposite* can be used for one of the paths when the two traverse a common arc but have opposite edge polarities. Again, the library determines the amount of variation that is canceled, but this is perhaps less than when the signal edges were the same for the two paths.

The library is given complete control over how much the common ambiguity between the two paths is reduced when they are compared.

10.23.11.1 Process points

The concept of best-case, nominal, and worst-case calculation modes represent process variation across all of the possible chips produced using a given process technology.

Specifically, best-case, nominal, and worst-case calculation modes shall represent three process points in the technology represented by a library. The contributions of minimum and maximum on-chip process variation to calculated values shall be relative to the process point specified via the calculation mode.

For some technologies and design flows, three possible process points might not be sufficient. To address these circumstances, an additional mechanism is added so the application can set an alternative process point. In such cases, a process point shall be an abstract floating point number associated by the library with a single set of process variables for all modeling domains. In contrast with the best-case, nominal, and worst-case process points, there shall be no single qualitative measure of circuit performance associated across all modeling domains with a process-point number.

There shall be only one process-point number for an entire chip at any given time. Unlike voltage and temperature, which are at least in some sense external stimulants to a chip, the process-point number is inherently a characteristic of the chip itself.

When more than three possible process points are needed, the application shall use *dpcmSetCurrentProcessPoint* (see 10.23.11.1.1) to set the current process point.

Once the application successfully sets a process point, it can request timing and power values be calculated relative to that process point through use of a new calculation mode value, *DCM_ProcessPoint*. This new value shall be appended to the *DCM_CalculationModes* enumeration, as shown in Table 307.

Table 307—DCM_CalculationModes

Enumerator	Enumeration	Description
DCM_BestCase	0	Best case
DCM_WorstCase	1	Worst case
DCM_NominalCase	2	Nominal case
DCM_ProcessPoint	3	Process point

Use of this value when the library has not accepted a process point via a call to *dpcmSetCurrentProcessPoint* (see 10.23.11.1.1) shall result in an error.

10.23.11.1.1 dpcmSetCurrentProcessPoint

Table 308 provides information on *dpcmSetCurrentProcessPoint*.

Table 308—dpcmSetCurrentProcessPoint

Function name	dpcmSetCurrentProcessPoint
Arguments	Process point
Result	Old process point
Standard Structure fields	None
DCL syntax	EXPOSE(dpcmSetCurrentProcessPoint) : passed(double: processPoint) result(double: oldProcessPoint);

C syntax	<pre>typedef struct { DCM_DOUBLE processPoint; } T_ProcessPoint; int dpcmSetCurrentProcessPoint (const DCM_STD_STRUCT *std_struct, T_ProcessPoint *rtn, DCM_DOUBLE processPoint);</pre>

The application can use this to set the current process point.

The value set by the application via *dpcmSetCurrentProcessPoint* shall fall within the range obtained from the library via *dpcmGetProcessPointRange* (see 10.23.11.1.2). If the application cannot call *dpcmGetProcessPointRange* successfully, a call to *dpcmSetCurrentProcessPoint* shall result in an error.

10.23.11.1.2 dpcmGetBaseProcessPoint

Table 309 provides information on *dpcmGetBaseProcessPoint*.

Table 309—dpcmGetBaseProcessPoint

Function name	dpcmGetBaseProcessPoint
Arguments	None
Result	Process point
Standard Structure fields	None
DCL syntax	<pre>EXPOSE(dpcmGetBaseProcessPoint): result(double: processPoint);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE processPoint; } T_ProcessPoint; int dpcmGetBaseProcessPoint (const DCM_STD_STRUCT *std_struct, T_ProcessPoint *rtn);</pre>

The application shall obtain a base process point from the library. The value returned by this function shall also fall within the range that can be obtained via *dpcmGetProcessPointRange* (see 10.23.11.2.1).

10.23.11.2 PVT ranges

A set of functions that return process point, voltage, and temperature ranges for the current operating range is defined in the next subclauses.

10.23.11.2.1 dpcmGetProcessPointRange

Table 310 provides information on *dpcmGetProcessPointRange*.

Table 310—dpcmGetProcessPointRange

Function name	dpcmGetProcessPointRange
Arguments	None
Result	Minimum process point, Maximum process point
Standard Structure fields	None
DCL syntax	<pre>EXPOSE(dpcmGetProcessPoint): result(double: minProcessPoint, maxProcessPoint);</pre>

C syntax	<pre>typedef struct { DCM_DOUBLE minProcessPoint; DCM_DOUBLE maxProcessPoint; } T_ProcessPointRange; int dpcmGetProcessPointRange (const DCM_STD_STRUCT *std_struct, T_ProcessPointRange *rtn);</pre>

This returns the extrema defining a range of legal values for a process point other than one represented by the best-case, nominal, or worst-case calculation modes back to the application. This range can be a function of an operating range. The value the application obtains using *dpcmGetBaseProcessPoint* and the value the application sets via *dpcmSetCurrentProcessPoint* (see 10.23.11.1.1) shall both fall within this range. If the application cannot obtain this range from the library, use of *dpcmSetCurrentProcessPoint* or the *DCM_ProcessPoint* calculation mode value shall result in an error.

These extrema allow the application to give the user a range from which to choose a process point, with the user's choice being relative to the extrema only and not in any absolute sense.

10.23.11.3 Rail voltage range¹⁸⁷

The calls discussed in the next subclauses can be used to determine a rail voltage range.

10.23.11.3.1 dpcmGetRailVoltageRangeArray

Table 311 provides information on *dpcmGetRailVoltageRangeArray*.

Table 311—*dpcmGetRailVoltageRangeArray*

Function name	<i>dpcmGetRailVoltageRangeArray</i>
Arguments	None
Result	Minimum rail voltages, Maximum rail voltages
Standard Structure fields	None
DCL syntax	<pre>EXPOSE(dpcmGetRailVoltageRangeArray) : result(double[*]: minRailVoltages, maxRailVoltages);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE_ARRAY *minRailVoltages; DCM_DOUBLE_ARRAY *maxRailVoltages; } T_RailVoltageRangeArray; int dpcmGetRailVoltageRangeArray (const DCM_STD_STRUCT *std_struct, T_RailVoltageRangeArray *rtn);</pre>

This returns the extrema defining the range of legal values for each rail voltage in a library back to the application. Each voltage range can be a function of operating range, but shall not be a function of calculation mode. The value returned by *appGetCurrentRailVoltage()*, which can be a function of the calculation mode, shall fall within this range. The value returned by *dpcmGetBaseRailVoltage()* shall also fall within this range. The order of the ranges returned by this function shall be the same as the order of the voltage rail names returned by *dpcmGetRailVoltageArray()*.

10.23.11.3.2 dpcmGetCellRailVoltageRangeArray

Table 312 provides information on *dpcmGetCellRailVoltageRangeArray*.

Table 312—dpcmGetCellRailVoltageRangeArray

Function name	dpcmGetCellRailVoltageRangeArray
Arguments	None
Result	Minimum rail voltages, Maximum rail voltages
Standard Structure fields	block, CellName, cellData (timing)
DCL syntax	EXPOSE(dpcmGetCellRailVoltageRangeArray): result(double[*]: minRailVoltages, maxRailVoltages);
C syntax	typedef struct { DCM_DOUBLE_ARRAY *minRailVoltages; DCM_DOUBLE_ARRAY *maxRailVoltages; } T_RailVoltageRangeArray; int dpcmGetCellRailVoltageRangeArray (const DCM_STD_STRUCT *std_struct, T_RailVoltageRangeArray *rtn);

This returns the extrema defining the range of legal values for each voltage rail of the cell identified by the *cellData* field in the *Standard Structure* back to the application. These values can be different for different operating ranges but shall not vary with calculation mode.

This function differs from *dpcmGetRailVoltageRangeArray*, as it returns the rail voltage ranges for the specified cell only, not the rail voltage ranges for an entire library. If this function is exposed by a DPCM, the application shall use it instead of, and shall not call, *dpcmGetRailVoltageRangeArray*. If the function *dpcmGetCellRailVoltageArray* is exposed by the DPCM, this function shall also be exposed in lieu of *dpcmGetRailVoltageRangeArray*.

The order of the ranges returned by this function shall be the same as the order of the voltage rail names returned by *dpcmGetCellRailVoltageArray*.

The value returned by *appGetCurrentRailVoltage*, which can be a function of the calculation mode, shall fall within the range returned by this function (if it is exposed) for the voltage rail specified. The values returned by *dpcmGetBaseCellRailVoltageArray* shall also fall within the corresponding ranges returned by this function.

10.23.11.4 Temperature range

The calls described in the next subclauses can be used to determine a temperature range.

10.23.11.4.1 dcmGetTemperatureRange

Table 313 provides information on dpcmGetTemperatureRange.

Table 313—dpcmGetTemperatureRange

Function name	dpcmGetTemperatureRange
Arguments	None
Result	Minimum temperature, Maximum temperature
Standard Structure fields	None
DCL syntax	EXPOSE(dpcmGetTemperatureRange): result(double: minTemperature, maxTemperature);

C syntax	<pre>typedef struct { DCM_DOUBLE minTemperature; DCM_DOUBLE maxTemperature; } T_TemperatureRange; int dpcmGetTemperatureRange (const DCM_STD_STRUCT *std_struct, T_TemperatureRange *rtn);</pre>

This returns the extrema defining a range of legal values for temperature back to the application. This range can be a function of operating range but shall not be a function of calculation mode. The value returned by *appGetCurrentTemperature()*, which can be a function of calculation mode, shall fall within this range. The value returned by *dpcmGetBaseTemperature* shall also fall within this range.

10.23.11.4.2 dpcmGetCellTemperatureRange

Table 314 provides information on dpcmGetCellTemperatureRange.

Table 314—dpcmGetCellTemperatureRange

Function name	dpcmGetCellTemperatureRange
Arguments	None
Result	Minimum temperature, Maximum temperature
Standard Structure fields	block, CellName, cellData (timing)
DCL syntax	<pre>EXPOSE(dpcmGetCellTemperatureRange): result(double: minTemperature, maxTemperature);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE minTemperature; DCM_DOUBLE maxTemperature; } T_CellTemperatureRange; int dpcmGetCellTemperatureRange (const DCM_STD_STRUCT *std_struct, T_CellTemperatureRange *rtn);</pre>

This returns the extrema defining a range of legal values for the temperature of the cell identified by the *cellData* field in the *Standard Structure* back to the application. These values can be different for different operating ranges but shall not vary with the calculation mode.

This function differs from *dpcmGetTemperatureRange*, as it returns a temperature range for the specified cell only, not an entire library. If this function is exposed by a DPCM, the application shall use it instead of, and shall not call, *dpcmGetTemperatureRange*.

The value returned by *appGetCurrentTemperature*, which can be a function of calculation mode, shall fall within the range returned by this function (if it is exposed). The value returned by *dpcmGetBaseCellTemperature* shall also fall within this range.

10.23.12 Accessing properties and attributes

This subclause describes APIs for accessing cell or pin properties that are constant. The properties are accessed through a pair of functions for each property. One function, the properties array call, returns an array of strings in which each string element represents a possible property used within the technology. The other function, the cell- or pin-specific property call, returns an array of integer indices into the array of strings returned by the first call in the pair.

The array of strings returned from a properties array call shall contain one element for each possible value a property or its modifier can have for the entire technology. There is no imposed or implied order to the elements returned in the array. The string array returned by the library shall not contain duplicates. An empty array indicates this type of property is not supported by the technology. The application shall not call for cell- or pin-specific properties when the matching array properties call has returned an empty array.

The array of integers returned by the cell- or pin-specific property call represents the cell's or pin's specific property when taken in its entirety. A cell or pin property can consist of a property index and zero or more modifier indices. The property and modifier indices can appear in any order. The cell or pin property array shall not contain duplicates. An empty array indicates there is no specific cell or pin property of this type for this instance.

For all APIs, a nonzero return code indicates that a particular property or attribute for a given vector, pin, or cell does not exist. In addition, for an API that returns an array of indices to a string array, an empty array shall indicate that a particular property or attribute for a given vector, pin, or cell does not exist.

10.23.12.1.1 APIs for annotations within PIN object

APIs have been defined to access annotations attached to pin objects. APIs generally appear in pairs, the first returning an array of available items for an annotation type and the second returning an array of indices within that array indicating those items that are “attached” to the pin object.

10.23.12.1.2 dpcmGetPinPinTypeArray

Table 315 provides information on dpcmGetPinPinTypeArray.

Table 315—dpcmGetPinPinTypeArray

Function name	dpcmGetPinPinTypeArray
Arguments	None
Result	Array of pin types
Standard Structure fields	None
DCL syntax	EXPOSE(dpcmGetPinPinTypeArray): result(string[*]: stringData);
C syntax	<pre>typedef struct { DCM_STRING_ARRAY *stringData; } T_GetPinPinTypeArray; int dpcmGetPinPinTypeArray (const DCM_STD_STRUCT *std_struct, T_GetPinPinTypeArray *rtn);</pre>

This returns all possible values for the *PinType* property that can be used by a technology back to the application. The string array returned by the library shall contain string elements from the following list.

- *Analog*—a pin that can have a signal with an arbitrary voltage or current.
- *Differential*—a pin that requires a signal that is determined by the difference of either voltage or current between two pins.
- *Digital*—a pin that can have a signal that changes from *Vss* to *Vdd* and back connected to it.
- *Reference*—a pin that requires a constant voltage as a reference, but this voltage is not to be connected from the power rails.
- *Supply*—a pin that supplies power.

- *Row*—a pin that indicates a physical row of a memory.
- *Column*—a pin that indicates a physical column of a memory.
- *Bank*—a pin that indicates a physical bank of a memory.

10.23.12.1.3 dpcmGetPinPinType

Table 316 provides information on dpcmGetPinPinType.

Table 316—dpcmGetPinPinType

Function name	dpcmGetPinPinType
Arguments	Pin pointer
Result	Array of pin type properties
Standard Structure fields	block, CellName, cellData, pathData (pin-specific)
DCL syntax	<pre>EXPOSE(dpcmGetPinPinType): passed(pin: pinPointer) result(integer[*]: arrayIndices);</pre>
C syntax	<pre>typedef struct { DCM_INTEGER_ARRAY *arrayIndices; } T_GetPinPinType; int dpcmGetPinPinType(const DCM_STD_STRUCT *std_struct, T_GetPinPinType *rtn, DCM_PIN pinPointer);</pre>

This returns the *PinType* property for a pin back to the application. *dpcmGetPinPinType* returns an array of indices into the string array returned by *dpcmGetPinPinTypeArray*, which apply to the passed pin pointer argument.

10.23.12.1.4 dpcmGetPinSignalTypeArray

Table 317 provides information on dpcmGetPinSignalTypeArray.

Table 317—dpcmGetPinSignalTypeArray

Function name	dpcmGetPinSignalTypeArray
Arguments	None
Result	Array of signal types
Standard Structure fields	None
DCL syntax	<pre>EXPOSE(dpcmGetPinSignalTypeArray): result(string[*]: stringData);</pre>
C syntax	<pre>typedef struct { DCM_STRING_ARRAY *stringData; } T_GetPinSignalTypeArray; int dpcmGetPinSignalTypeArray (const DCM_STD_STRUCT *std_struct, T_GetPinSignalTypeArray *rtn);</pre>

This returns all possible values for the *SignalType* property used by the technology back to the application. A *signal type* is the type of signal that shall be present on the pin of interest. A signal type for a pin consists of a type and zero or more modifiers.

The *DATA* signal type is assumed to be present in the list and does not require a bit value.

The possible signal types are as follows:

- *Clock*—a signal type indicating a signal that initiates or enables the state change in a memory device.
- *Tie*—a signal type indicating this signal is attached to an input pin and has a constant level: a logical one or a logical zero.
- *Refresh*—a signal type indicating this signal is attached to an input pin that maintains the contents of the memory without modifying it.
- *Load*—a signal type indicating this signal is attached to an input pin that loads control registers.
- *IDDQ*—a signal type indicating this signal is attached to an input pin that supplies the current measurement.

The possible modifiers are as follows:

- *Address*—a modifier indicating the signal on this pin is used to select the specific storage element in the array where data is written or read.
- *Control*—a modifier indicating the signal is used to select one or more functions inside the cell.
- *Enable*—a modifier indicating the signal, when active, allows the device to perform its intended function.
- *Master*—a modifier indicating this signal controls the primary latch in an LSSD latch pair.
- *Read*—a modifier indicating this signal corresponds to the read activity of a memory device.
- *Clear*—a modifier indicating the signal forces the latch or memory device into a “well known” zero state.
- *Reset*—a modifier indicating the signal forces the latch or memory device into a predefined state.
- *Set*—a modifier indicating the signal forces the latch or memory device into a “well known” one state.
- *Slave*—a modifier indicating this signal controls the secondary, or slave, latch in an LSSD latch pair.
- *Scan*—a modifier indicating the signal attached to the pin is used for sequentially initializing or reading the combined state of a system of memory devices.
- *Write*—a modifier indicating this signal corresponds to the write activity of a memory device.
- *Schmitt*—a modifier indicating the input pin where the signal is attached requires a significant amount of hysteresis in its operation.
- *Tristate*—a modifier indicating the pin under analysis can have three states: a logical zero, a logical one, or a high impedance state.
- *Xtal*—a modifier indicating the signal connected to the pin repeats its waveform on a uniform period.
- *Pad*—a modifier attached to an output pin indicating this signal is from a circuit driving an off-chip net.
- *Test*—a modifier indicating the pin is associated with a TEST operation.
- *BIST*—a modifier indicating the pin is associated with a built-in-self-test (*BIST*) operation.

10.23.12.1.5 dpcmGetPinSignalType

Table 318 provides information on dpcmGetPinSignalType.

Table 318—dpcmGetPinSignalType

Function name	dpcmGetPinSignalType
Arguments	Pin pointer
Result	Array of indices to string array
Standard Structure fields	block, CellName, cellData, pathData (pin-specific)
DCL syntax	<pre>EXPOSE(dpcmGetPinSignalType): passed(pin: pinPointer) result(int[*]: arrayIndices);</pre>
C syntax	<pre>typedef struct { DCM_INTEGER_ARRAY *arrayIndices; } T_GetPinSignalType; int dpcmGetPinSignalType (const DCM_STD_STRUCT *std_struct, T_GetPinSignalType *rtn, DCM_PIN pinPointer);</pre>

This returns an array of indices into the signal type array (which constitutes the complete signal type for the pin of interest) back to the application. If the index array returned by this call is empty or contains only modifiers, the application shall assume the signal type is *data*.

10.23.12.1.6 dpcmGetPinActionArray

Table 319 provides information on dpcmGetPinActionArray.

Table 319—dpcmGetPinActionArray

Function name	dpcmGetPinActionArray
Arguments	None
Result	Array of pin actions
Standard Structure fields	None
DCL syntax	<pre>EXPOSE(dpcmGetPinActionArray): result(string[*]: stringData);</pre>
C syntax	<pre>typedef struct { DCM_STRING_ARRAY *stringData; } T_GetPinActionArray; int dpcmGetPinActionArray (const DCM_STD_STRUCT *std_struct, T_GetPinActionArray *rtn);</pre>

This returns all possible values for the *Action* property back to the application. *dpcmGetPinActionArray* returns a string array of possible pin actions, where the possible elements are from the following:

- *Synchronous*—synchronous signals have a relationship in time to a clock.
- *Asynchronous*—asynchronous signals have no relationship to a clock.

10.23.12.1.7 dpcmGetPinAction

Table 320 provides information on dpcmGetPinAction.

Table 320—dpcmGetPinAction

Function name	dpcmGetPinAction
Arguments	Pin pointer
Result	Array of indices to string array
Standard Structure fields	block, CellName, cellData, pathData (pin-specific)
DCL syntax	<pre>EXPOSE(dpcmGetPinAction): passed(pin: pinPointer) result(int[*]: arrayIndices);</pre>
C syntax	<pre>typedef struct { DCM_INTEGER_ARRAY *arrayIndices; } T_GetPinAction; int dpcmGetPinAction(const DCM_STD_STRUCT *std_struct, T_GetPinAction *rtn, DCM_PIN pinPointer);</pre>

This returns an array of indices, which comprise the synchronous/asynchronous properties of a signal at the pin of interest, back to the application.

10.23.12.1.8 dpcmGetPinPolarityArray

Table 321 provides information on dpcmGetPinPolarityArray.

Table 321—dpcmGetPinPolarityArray

Function name	dpcmGetPinPolarityArray
Arguments	None
Result	Array of pin polarities
Standard Structure fields	None
DCL syntax	<pre>EXPOSE(dpcmGetPinPolarityArray): result(string[*]: stringData);</pre>
C syntax	<pre>typedef struct { DCM_STRING_ARRAY *stringData; } T_GetPinPolarityArray; int dpcmGetPinPolarityArray (const DCM_STD_STRUCT *std_struct, T_GetPinPolarityArray *rtn);</pre>

This returns the *Polarity* property for a pin back to the application. *dpcmGetPinPolarityArray* returns a string array of pin polarities that can be used in the technology. The possible pin polarities and their meanings are contained in the following:

- *High*—the pin is active during the period of time when the signal is at a level one.
- *Low*—the pin is active during the period of time when the signal is at a level zero.
- *Falling_edge*—the pin is active when the signal is transitioning from a level one to a level zero.
- *Rising_edge*—the pin is active when the signal is transitioning from a level zero to a level one.
- *Double_edge*—the pin is active when the signal is transitioning from a level one to a level zero or from a level zero to a level one.

10.23.12.1.9 dpcmGetPinPolarity

Table 322 provides information on dpcmGetPinPolarity.

Table 322—dpcmGetPinPolarity

Function name	dpcmGetPinPolarity
Arguments	Pin pointer
Result	Array of indices to string array
Standard Structure fields	block, CellName, cellData, pathData (pin-specific)
DCL syntax	<pre>EXPOSE(dpcmGetPinPolarity): passed(pin: pinPointer) result(int[*]: arrayIndices);</pre>
C syntax	<pre>typedef struct { DCM_INTEGER_ARRAY *arrayIndices; } T_GetPinPolarity; int dpcmGetPinPolarity (const DCM_STD_STRUCT *std_struct, T_GetPinPolarity *rtn, DCM_PIN pinPointer);</pre>

This returns an array of indices that make up the pin polarity for the input pin of interest. The application can only call this function for input pins.

10.23.12.1.10 dpcmGetPinEnablePin

Table 323 provides information on dpcmGetPinEnablePin.

Table 323—dpcmGetPinEnablePin

Function name	dpcmGetPinEnablePin
Arguments	Pin pointer
Result	Array of pin names
Standard Structure fields	block, CellName, cellData, pathData (pin-specific)
DCL syntax	<pre>EXPOSE(dpcmGetPinEnablePin): passed(pin: pinPointer) result(string[*]: pinNames);</pre>
C syntax	<pre>typedef struct { DCM_STRING_ARRAY *pinNames; } T_GetPinEnablePin; int dpcmGetPinEnablePin (const DCM_STD_STRUCT *std_struct, T_GetPinEnablePin *rtn, DCM_PIN pinPointer);</pre>

This returns the list of pins that enable the pin identified by the pin pointer argument back to the application.

10.23.12.1.11 dpcmGetPinConnectClass

Table 324 provides information on dpcmGetPinConnectClass.

Table 324—dpcmGetPinConnectClass

Function name	dpcmGetPinConnectClass
Arguments	None
Result	Index into library, connect class array
Standard Structure fields	block, CellName, cellData, pathData (pin-specific)
DCL syntax	EXPOSE(dpcmGetPinConnectClass): result(int: connectClassIndex);
C syntax	typedef struct { DCM_INTEGER connectClassIndex; } T_GetPinConnectClass; int dpcmGetPinConnectClass (const DCM_STD_STRUCT *std_struct, T_GetPinConnectClass *rtn);

This returns an index into the array of library connect classes back to the application. The API that obtains this connect class array is specified in dpcmGetLibraryConnectClassArray. The returned library level connect class index is used in conjunction with that of another pin to index into the connect class-based connectivity rule array (see 10.23.14.3) to determine the allowable connectivity between the pins.

10.23.12.1.12 dpcmGetPinScanPosition

Table 325 provides information on dpcmGetPinScanPosition.

Table 325—dpcmGetPinScanPosition

Function name	dpcmGetPinScanPosition
Arguments	Pin pointer
Result	Scan pin position
Standard Structure fields	block, CellName, cellData, pathData (pin-specific)
DCL syntax	EXPOSE(dpcmGetPinScanPosition): passed(pin: pinPointer) result(int: scanPositionValue);
C syntax	typedef struct { DCM_INTEGER scanPositionValue; } T_GetPinScanPosition; int dpcmGetPinScanPosition (const DCM_STD_STRUCT *std_struct, T_GetPinScanPosition *rtn, DCM_PIN pinPointer);

This returns the position of a pin in the scan for a cell back to the application. A value of zero (0) indicates this pin does not appear in the scan chain.

10.23.12.1.13 dpcmGetPinStuckArray

Table 326 provides information on dpcmGetPinStuckArray.

Table 326—dpcmGetPinStuckArray

Function name	dpcmGetPinStuckArray
Arguments	None
Result	Array of stuck at types
Standard Structure fields	None
DCL syntax	EXPOSE(dpcmGetPinStuckArray): result(string[*]: stringData);
C syntax	typedef struct { DCM_STRING_ARRAY *stringData; } T_GetPinStuckArray; int dpcmGetPinStuckArray (const DCM_STD_STRUCT *std_struct, T_GetPinStuckArray *rtn);

This returns all possible stuck-at-fault values used within the technology back to the application. The legal values are as follows:

- *stuck_at_0*—the pin of a cell has a potential failure mode where a pin can remain at a logical zero, regardless of the internal state of the cell.
- *stuck_at_1*—the pin of a cell has a potential failure mode where a pin can remain at a logical one, regardless of the internal state of the cell.

The application shall not call *dpcmGetPinStuckArray* for pins that are not outputs.

10.23.12.1.14 dpcmGetPinStuck

Table 327 provides information on *dpcmGetPinStuck*.

Table 327—dpcmGetPinStuck

Function name	dpcmGetPinStuck
Arguments	Pin pointer
Result	Array of stuck type indices
Standard Structure fields	block, CellName, cellData, pathData(pin-specific)
DCL syntax	EXPOSE(dpcmGetPinStuck): passed(pin: pinPointer) result(int[*]: arrayIndices);
C syntax	typedef struct { DCM_INTEGER_ARRAY *arrayIndices; } T_GetPinStuck; int dpcmGetPinStuck(const DCM_STD_STRUCT *std_struct, T_GetPinStuck *rtn, DCM_PIN pinPointer);

This returns a list of the stuck failure types for a pin back to the application.

10.23.12.1.15 dpcmGetDifferentialPairPin

Table 328 provides information on *dpcmGetDifferentialPairPin*.

Table 328—dpcmGetDifferentialPairPin

Function name	dpcmGetDifferentialPairPin
Arguments	Pin pointer
Result	Differential pair pin name
Standard Structure fields	block, CellName, cellData, pathData(pin-specific)
DCL syntax	<pre>EXPOSE(dpcmGetDifferentialPairPin): passed(pin: pinPointer) result(string: differentialPairPinName);</pre>
C syntax	<pre>typedef struct { DCM_STRING differentialPairPinName; } T_GetDifferentialPairPin; int dpcmGetdifferentialPairPin (const DCM_STD_STRUCT *std_struct, T_GetDifferentialPairPin *rtn, DCM_PIN pinPointer);</pre>

Cells with pins that require logic signals to be wired as differential pairs exist in many technologies. *dpcmGetDifferentialPairPin* returns the name of the pin that shall be paired with the known pin back to the application. The application calls this function when it knows the signal on a pin is a differential signal but does not know the corresponding differential pin.

10.23.12.2 APIs for annotations within VECTOR objects

APIs have been defined to access annotations and attributes attached to vector objects.

10.23.12.2.1 dpcmGetPathLabel

Table 329 provides information on dpcmGetPathLabel.

Table 329—dpcmGetPathLabel

Function name	dpcmGetPathLabel
Arguments	None
Result	Path label
Standard Structure fields	block, CellName, cellData, pathData (vectorTiming pin-specific)
DCL syntax	<pre>EXPOSE(dpcmGetPathLabel): result(string: pathLabel);</pre>
C syntax	<pre>typedef struct { DCM_STRING pathLabel; } T_GetPathLabel; int dpcmGetPathLabel(const DCM_STD_STRUCT *std_struct, T_GetPathLabel *rtn);</pre>

This returns the SDF label for a vector expression that represents at least one state for a path back to the application.

10.23.12.2.2 dpcmGetPowerStateLabel

Table 330 provides information on dpcmGetPowerStateLabel.

Table 330—dpcmGetPowerStateLabel

Function name	dpcmGetPowerStateLabel
Arguments	Group index, Condition index
Result	State label
Standard Structure fields	block, CellName, cellData (power), modelDomain (power)
DCL syntax	<pre>EXPOSE(dpcmGetPowerStateLabel): passed(int: groupIndex, conditionIndex) result(string: StateLabel);</pre>
C syntax	<pre>typedef struct { DCM_STRING StateLabel; } T_GetPowerStateLabel; int dpcmGetPowerStateLabel (const DCM_STD_STRUCT *std_struct, T_GetPowerStateLabel *rtn, DCM_INTEGER groupIndex, DCM_INTEGER conditionIndex);</pre>

This returns the label for the particular group and condition back to the application. This call is only valid for a cell-level state label.

10.23.12.3 APIs for annotations within CELL objects

APIs have been defined to access annotations attached to cell objects. APIs generally appear in pairs, the first returning an array of available items for an annotation type and the second returning an array of indices within that array indicating those items that are “attached” to the cell object.

10.23.12.3.1 dpcmGetCellTypeArray

Table 331 provides information on dpcmGetCellTypeArray.

Table 331—dpcmGetCellTypeArray

Function name	dpcmGetCellTypeArray
Arguments	None
Result	Cell types
Standard Structure fields	None
DCL syntax	<pre>EXPOSE(dpcmGetCellTypeArray): result(string[*]: stringData);</pre>
C syntax	<pre>typedef struct { DCM_STRING_ARRAY *stringData; } T_GetCellTypeArray; int dpcmGetCellTypeArray (const DCM_STD_STRUCT *std_struct, T_GetCellTypeArray *rtn);</pre>

This returns all possible cell types used within a technology back to the application. A cell type consists of a primary definition and zero or more adjectives, which further describe the cell type.

The primary cell types are as follows:

- *Buffer*—cells that are used for line conditioning or drive strength changes. This primary cell type typically includes both inverting and noninverting circuits. The allowable adjectives that can accompany a buffer cell type are as follows:
- *Internal* indicates the cell shall be used to send and receive logical signals from other cells on the same chip.
- *Input* indicates the cell shall be used to receive logical signals from other chips.
- *Output* indicates the cell shall be used to transmit logical signals to other chips. The allowable adjectives that can accompany any buffer adjective, except *Internal*, are:
- *Predriver* indicates the load on this buffer is a buffer that drives “off-chip” circuitry.
- *Slotdriver* indicates this driver is driving “off-chip” circuitry.
- *Combinatorial*—cells that perform Boolean operations, but not those cells that have more explicit definitions defined in this subclause (e.g., a cell that performs the Boolean AND function on two pins).
- *Multiplexer*—cells that perform the function of selection.
- *Latch*—cells that perform memory functions controlled by the level of the clock signal.
- *Flipflop*—cells that perform memory functions controlled by the transition of the clock signal.
- *Memory*—cells that perform mass storage of information controlled both by an address and a clock. They can have two basic operations: reading, which extracts information already stored in the cell, and writing, which stores new information in the cell. The location where the information is stored within the cell is controlled by the bit pattern of the address.
- *Block*—cells that perform very complex logic functions, including finite state machines. Blocks are composed completely of other library elements. Some applications can request further information about the configuration of the block by requesting the internal cells to use in constructing the block, as well as their interconnection details, from the library.
- *Core*—cells that perform very complex logic functions, including finite state machines. These cells are typically made using custom transistor design techniques or are prepackaged collections of application specific integrated circuit (ASIC) cells designed to perform the desired function. The application shall not request further details about the internal design of core.
- *Special*—cells that cannot be defined by the other cell types. As an example, some typical cells that can be considered special are line hold and clamping or suppression devices.
- *PLL*—cells that form the phase lock loop (PLL).
- Their possible modifiers are as follows:
- *RAM*—a modifier to 'memory' that indicates the cell is a random access memory (RAM) with both read and write capabilities.
- *ROM*—a modifier to 'memory' that indicates the cell is a random access memory with read, but no write, capability [read only memory (ROM)] in a manner that associates the bit pattern with the data.
- *CAM*—a modifier to 'memory' that indicates the cell is a content accessible memory (CAM). This type of memory is associative in nature. The access is by a bit pattern rather than being an absolute address. When reading, the bit pattern is supplied and the data associated with the bit pattern are returned. When writing to the memory, the data are stored in a manner that associates bit pattern with the data.

- *Static*—a modifier that indicates the cell is a static device capable of holding a given state indefinitely.
- *Dynamic*—a modifier that indicates the cell is a device that holds a state based on the charging of a capacitance. These devices can hold a state for a limited period of time before the state of the device becomes undefined.
- *Asynchronous*—a modifier that indicates the cell operates without synchronization with respect to any controlling signal.
- *Synchronous*—a modifier that indicates the cell operates in synchronization with a controlling signal(i.e., a clock).

10.23.12.3.2 dpcmGetCellType

Table 332 provides information on dpcm GetCellType.

Table 332—dpcmGetCellType

Function name	dpcmGetCellType
Arguments	None
Result	Array of indices to cell type string array
Standard Structure fields	block, CellName, cellData
DCL syntax	EXPOSE(dpcmGetCellType): result(int[*]: arrayIndices);
C syntax	typedef struct { DCM_INTEGER_ARRAY *arrayIndices; } T_GetCellType; int dpcmGetCellType(const DCM_STD_STRUCT *std_struct, T_GetCellType *rtn);

This returns an array of the cell type indices for a cell back to the application.

10.23.12.3.3 dpcmGetCellSwapClassArray

Table 333 provides information on dpcmGetCellSwapClassArray.

Table 333—dpcmGetCellSwapClassArray

Function name	dpcmGetCellSwapClassArray
Arguments	None
Result	Swap class string array
Standard Structure fields	None
DCL syntax	EXPOSE(dpcmGetCellSwapClassArray): result(string[*]: swapClassArray);
C syntax	typedef struct { DCM_STRING_ARRAY *swapClassArray; } T_GetCellSwapClassArray; int dpcmGetCellSwapClassArray (const DCM_STD_STRUCT *std_struct, T_GetCellSwapClassArray *rtn);

This returns all possible *SwapClass* names back to the application. Cells that are members of the same swap class can be exchanged in the design by the application. The application shall not substitute cells that are not members of the same swap class. A cell can be a member of more than one swap class.

An application is responsible for performing the proper substitution of cells as the design process proceeds toward completion. Provided is a set of swap classes to assist in this process. A *swap class* is a named collection of zero or more cells that can be interchanged. Swap classes convey information about a cell's suitable replacements to the application that cannot be readily determined through other APIs.

Cell swapping is only allowed under the following conditions:

- The RestrictClass annotation (see 10.23.12.4) authorizes usage of the cell.
- The cells to be swapped are compatible from an application standpoint.

Each swap class is named. The library is responsible for providing the list of swap classes supported by the technology as an array of strings, where each string is the name of a swap class, back to the application.

Each cell in the library is responsible for providing the application a list of swap classes to which it belongs, as an array of indices into the swap class array provided by the library.

10.23.12.3.4 dpcmGetCellSwapClass

Table 334 provides information on dpcmGetCellSwapClass.

Table 334—dpcmGetCellSwapClass

Function name	dpcmGetCellSwapClass
Arguments	None
Result	Array of indices to string
Standard Structure fields	array, block, CellName, cellData
DCL syntax	EXPOSE(dpcmGetCellSwapClass): result(int[*], arrayIndices);
C syntax	typedef struct { DCM_INTEGER_ARRAY *arrayIndices; } T_GetCellSwapClass; int dpcmGetCellSwapClass (const DCM_STD_STRUCTURE *std_struct, T_GetCellSwapClass *rtn);

This returns the set of swap class values, which apply to a cell, back to the application.

10.23.12.4 dpcmGetCellRestrictClassArray

Table 335 provides information on dpcmGetCellRestrictClassArray.

Table 335—dpcmGetCellRestrictClassArray

Function name	dpcmGetCellRestrictClassArray
Arguments	None
Result	Restrict class string array
Standard Structure fields	None

DCL syntax	<code>EXPOSE(dpcmGetCellRestrictClassArray): result(string[*]: restrictClassArray)</code>
C syntax	<code>typedef struct { DCM_STRING_ARRAY *restrictClassArray; } T_GetCellRestrictClassArray; int dpcmGetCellRestrictClassArray (const DCM_STD_STRUCT *std_struct, T_GetCellRestrictClassArray *rtn);</code>

This returns an array of restrict class strings supported by the library back to the application, as follows:

- *Synthesis*—the cell is restricted to applications capable of performing logical synthesis.
- *Scan*—the cell is restricted to applications capable of performing scan tree insertion.
- *Datapath*—the cell is restricted to applications capable of performing data path synthesis.
- *Clock*—the cell is restricted to applications capable of performing clock tree synthesis.
- *BIST*—the cell is restricted to applications capable of performing *BIST*.
- *Layout*—the cell is restricted to applications capable of performing placement, routing, and other layout tasks.

A wide variety of applications can be used during the design process. Typical logic design operations have been categorized as *restrict classes*. Each tool operation that alters the design logic falls into one or more of these categories depending on the type of operation they are performing at the time. For example, a layout tool may want to expand the cellset with those cells restricted only to layout, as well as use the more general cellset of synthesis and scan if it is capable of performing the similar operations of those tools.

In many technologies, cells have already been designed for special portions of the design process, such as clock tree synthesis. These special purpose cells shall not be used by applications that are not performing the specific design operation for which these cells are intended. The subset of restrict classes associated with a cell is used to convey in which design operations the cell can be used to the application.

A *restrict class* is a collection of zero or more cells that can be used for a specific design operation. Each restrict class is named. The library is responsible for giving a list of all the restrict classes used by that technology, as an array of strings, back to the application. If a cell has no or any unknown values for *RestrictClass*, the application shall not modify any instantiation or allow creation of that cell in the design. However, the cell shall still be considered for analysis and linking.

A cell that is a member of this class has a reference to zero or more restrict classes with which it is associated as part of its description. Each cell in the library is responsible for identifying to which of the library supported restrict classes it belongs, as an array of indices into the restrict class array for the library.

The application shall use a cell if, and only if, that cell contains the restrict classes known by the library and matches the restrict classes known to the application.

10.23.12.4.1 dpcmGetCellRestrictClass

Table 336 provides information on dpcmGetCellRestrictClass.

Table 336—dpcmGetCellRestrictClass

Function name	dpcmGetCellRestrictClass
Arguments	None
Result	Array of indices to string array
Standard Structure fields	block, CellName, cellData
DCL syntax	EXPOSE(dpcmGetCellRestrictClass): result(int[*]: arrayIndices);
C syntax	<pre>typedef struct { DCM_INTEGER_ARRAY *arrayIndices; } T_GetCellRestrictClass; int dpcmGetCellRestrictClass (const DCM_STD_STRUCT *std_struct, T_GetCellRestrictClass *rtn);</pre>

This returns the set of restrict class values that apply to a cell back to the application.

10.23.12.4.2 dpcmGetCellScanTypeArray

Table 337 provides information on dpcmGetCellScanTypeArray.

Table 337—dpcmGetCellScanTypeArray

Function name	dpcmGetCellScanTypeArray
Arguments	None
Result	Scan type string array
Standard Structure fields	None
DCL syntax	EXPOSE(dpcmGetCellScanTypeArray): result(string[*]: scanTypeArray);
C syntax	<pre>typedef struct { DCM_STRING_ARRAY *scanTypeArray; } T_GetCellScanTypeArray; int dpcmGetCellScanTypeArray (const DCM_STD_STRUCT *std_struct, T_GetCellScanTypeArray *rtn);</pre>

This returns an array of all possible scan type values used within a technology back to the application.

- *muxscan*—the cell uses a multiplexer device to control the scan.
- *clocked*—the cell uses a special dedicated clock to perform the scan.
- *lssd*—the cell uses a pair of latches and a special dedicated clock to perform the scan.
- *control_0*—the scan control pin shall be at a logical zero level for the scan to be active.
- *control_1*—the scan control pin shall be at a logical one level for the scan to be active.

10.23.12.4.3 dpcmGetCellScanType

Table 338 provides information on dpcmGetCellScanType.

Table 338—dpcmGetCellScanType

Function name	dpcmGetCellScanType
Arguments	None
Result	Array of indices to string array
Standard Structure fields	block, CellName, cellData
DCL syntax	EXPOSE(dpcmGetCellScanType): result(integer[*]: arrayIndices);
C syntax	typedef struct { DCM_INTEGER_ARRAY *arrayIndices; } T_GetCellScanType; int dpcmGetCellScanType (const DCM_STD_STRUCT *std_struct, T_GetCellScanType *rtn);

This returns an array of the scan type property indices for a cell back to the application.

10.23.12.4.4 dpcmGetCellNonScanCell

Table 339 provides information on dpcmGetCellNonScanCell.

Table 339—dpcmGetCellNonScanCell

Function name	dpcmGetCellNonScanCell
Arguments	None
Result	Cell or primitive name, Non-scan cell pins, Scan cell pins, Pin mapping array
Standard Structure fields	block, CellName, cellData
DCL syntax	typedef(nonScanCellStruct): result(string: cellOrPrimitiveIdentifier; string[*]: nonScanIdentifierPins, scanIdentifierPins; int[*]: pinMappingArray; nonScanCellStruct[*]: nextHierarchicalLevel); EXPOSE(dpcmGetCellNonScanCell): result(nonScanCellStruct[*]: nonScanCellAttribute);

C syntax	<pre> typedef enum DCM_PinMappingTypes { DCM_MAPS, DCM_DONT_CONNECT, DCM_DONT_CARE, DCM_TIE_LOW, DCM_TIE_HI } DCM_PinMappingTypes; typedef DCM_PinMappingTypes DCM_PinMappingTypes_ARRAY; struct DCM_NonScanCellStruct; typedef struct DCM_NonScanCellStruct *DCM_NonScanCellStruct_ARRAY; typedef struct DCM_NonScanCellStruct { DCM_STRING cellOrPrimitiveIdentifier; DCM_STRING_ARRAY *nonScanIdentifierPins; DCM_STRING_ARRAY *scanIdentifierPins; DCM_PinMappingTypes_ARRAY *pinMappingArray; DCM_NonScanCellStruct_ARRAY *nextHierarchicalLevel; } DCM_NonScanCellStruct; typedef struct { DCM_NonScanCellStruct_ARRAY *nonScanCellStructArray; } T_GetCellNonScanCellResults; int dpcmGetCellNonScanCell (const DCM_STD_STRUCT *std_struct, T_GetCellNonScanCellResults *rtn); </pre>

This relates a scan cell and a component cell that does not support scan. More complex cells, which are made from simpler library cells, can exist in many libraries. One such typical case is a scan flipflop can be made from a non-scan flipflop and some control logic. This call identifies the non-scan flipflop used to construct the scan flipflop.

A scan to non-scan pin map consists of three pieces of information carried in the scan pins' array, non-scan pins' array, and pin mapping arrays. Each element in these arrays correspond to a single pin within either cell.

All three arrays shall be the same length, as follows:

- The *i*th element in the scan pins array shall be the pin name, which corresponds to the pin name contained in the *i*th element in the non-scan pins array.
- The *i*th element of the pin map indicates the pins' correlation.
- If the *i*th element of the pin map array is not the value DCM_MAPS, then the *i*th element of either the scan pins' array or the non-scan pins' array, which does not contain a valid pin, shall be represented by the empty string.

An empty returned array or a non-zero return code shall indicate the *NonScanCell* property is not valid for the requested cell.

DCM_NonScanCellStruct_ARRAY is a *DCM_ARRAY* of pointers to *DCM_NonScanCellStruct*. Both the array and the structures whose addresses it contains are allocated by the DPCM. The application shall not allocate nor free the array or these structures. If the application needs to retain the array itself or any of

these structures, it shall call *dcm_lock_DCM_ARRAY* or *dcm_lock_DCM_STRUCT* (for each structure to be retained), respectively.

The results are the cell or primitive name, and a mapping of pin names that correspond between the scan and non-scan member, as shown in the following list:

- *Cell*—the name of the non-scan cell used to construct this scan cell. The *sub_qual* and domain are assumed to be the same as that of the scan cell.
- *Non-scan pins*—an array of pin names contained in the non-scan cell that have a corresponding pin in the scan cell. If there is no corresponding match for a given pin, this specific entry shall be an empty string.
- *Scan pins*—an array of pin names contained in the scan cell that have a corresponding pin in the non-scan cell. If there is no corresponding match for a given pin, this specific entry shall be an empty string.
- *Pin mapping array* (see below)—identifies the map type for each pin in the scan and non-scan pins' array.

Table 340 provides information on DCM_PinMappingTypes.

Table 340—DCM_PinMappingTypes

Enumerator	Enumeration	Description
DCM_MAPS	0	A matching pin exists on both the scan and non-scan cell.
DCM_DONT_CONNECT	1	The pins do not match and do not connect.
DCM_DONT_CARE	2	The pins do not match and whether they connect is optional.
DCM_TIE_LOW	3	The pins do not match and the non-scan pin shall be connected to <i>ground</i> .
DCM_TIE_HI	4	The pins do not match and the non-scan pin shall be connected to a logical high.

10.23.12.4.5 appSetVectorOperations

Table 341 provides information on appSetVectorOperations.

Table 341—appSetVectorOperations

Function name	appSetVectorOperations
Arguments	Array of vector operations
Result	None
Standard Structure fields	block, CellName, cellData, pathData (vectorTiming-arc-specific)
DCL syntax	EXTERNAL (appSetVectorOperations) : passed(int[*]: vectorOperations) result(int: ignore);

C syntax	<pre> typedef enum DCM_VectorOperations { DCM_VectorOperationRead, DCM_VectorOperationWrite, DCM_VectorOperationReadModifyWrite, DCM_VectorOperationStart, DCM_VectorOperationEnd, DCM_VectorOperationRefresh, DCM_VectorOperationLoad, DCM_VectorOperationIddq, DCM_VectorOperationIllegal } DCM_VectorOperations typedef DCM_VectorOperations DCM_VECTOR_OPERATIONS_ARRAY; typedef struct { DCM_VECTOR_OPERATIONS_ARRAY *vectorOperations; } T_VectorOperations; typedef struct { DCM_INTEGER ignore; } T_Ignore; int appSetVectorOperations (const DCM_STD_STRUCT *std_struct, T_Ignore *rtn, T_VectorOperations *vectorOperations); </pre>

This requests from the application store an array of vector operations (corresponding to values shown below), which are associated with the vector specified by the segment last defined during vector timing model elaboration.

Table 342 provides information on DCM_VectorOperations.

Table 342—DCM_VectorOperations

Enumerator	Enumeration	Description
DCM_VectorOperationRead	0	A read operation at one address.
DCM_VectorOperationWrite	1	A write operation at one address.
DCM_VectorOperationReadModifyWrite	2	A read followed by a write at the same address.
DCM_VectorOperationStart	3	The first operation required in a particular mode.
DCM_VectorOperationEnd	4	The last operation required in a particular mode.
DCM_VectorOperationRefresh	5	An operation required to maintain the contents of a dynamic element without modification.
DCM_VectorOperationLoad	6	An operation for loading control registers.
DCM_VectorOperationIddq	7	An operation for quiescent state supply current measurement.
DCM_VectorOperationIllegal	8	The operation is illegal.

10.23.12.4.6 dpcmGetLevelShifter

Table 343 provides information on dpcmGetLevelShifter.

Table 343—dpcmGetLevelShifter

Function name	dpcmGetLevelShifter
Arguments	Driver voltage low, Driver voltage high, Receiver voltage low, Receiver voltage high, Driver cell name, Receiver cell name
Result	New driver cell name, New receiver cell name, Replace or add driver cell, Replace or add receiver cell
Standard Structure fields	None
DCL syntax	<pre>EXPOSE(dpcmGetLevelShifter): passed(double: driverVol, driverVoh, receiverVol, receiverVoh; string: driverCell, receiverCell) result(string: newDriverCell, newReceiverCell; int: driverCellAction, receiverCellAction);</pre>
C syntax	<pre>enum DCM_LevelShifterAction { DCM_NO_CELL_CHANGE, DCM_REPLACE_CELL, DCM_ADD_CELL }; typedef struct { DCM_STRING newDriverCell, newReceiverCell; DCM_LevelShifterAction driverCellAction, receiverCellAction; } T_NewLevelShifter; int dpcmGetLevelShifter(const DCM_STD_STRUCT *std_struct, T_NewLevelShifter *rtn, DCM_DOUBLE driverVol, DCM_DOUBLE driverVoh, DCM_DOUBLE receiverVol, DCM_DOUBLE receiverVoh, DCM_STRING driverCell, DCM_STRING receiverCell);</pre>

This function provides a way for the application to identify new cells to be used to convert signal levels between two different regions of a design, commonly referred to as *voltage islands*, in which a given supply voltage has different values. To begin this process, the application shall first obtain the voltage thresholds for both the driving pin and receiving pins of a pair cells that share an interconnect network between two voltage islands. The application shall then call *dpcmGetLevelShifter*, passing in the driving-cell threshold, receiving-cell threshold, the driving-cell name, and the receiving-cell name. *DpcmGetLevelShifter* shall return any new voltage-shifter cells that are needed and an indication of how each of these cells is to be used.

The library shall determine whether level-shifting cell(s) are needed based on the thresholds provided. The library shall identify up to two such cells to the application and provide indication as to whether these cells shall replace the driver and/or receiver cells identified by the application or be inserted between them.

The library shall set *newDriverCell*, *newReceiverCell*, *driverCellAction* and *receiverCellAction* to communicate this information to the application. *driverCellAction* and *receiverCellAction* shall both have one of the following values (with associated actions):

- 0—no change to an existing cell nor the addition of a new cell
- 1—replace an existing cell with *newDriverCell* or *newReceiverCell*
- 2—add *newDriverCell* or *newReceiverCell* between the existing cells

10.23.13 APIs for attribute within a PIN object

APIs have been defined to access attributes attached to pin objects. The APIs return an enumeration value of the attribute “attached” to the pin object.

10.23.13.1 dpcmGetPinTiePolarity

Table 344 provides information on dpcmGetPinTiePolarity.

Table 344—dpcmGetPinTiePolarity

Function name	dpcmGetPinTiePolarity
Arguments	Pin pointer
Result	Index into an array of pin-polarity strings
Standard Structure fields	block, CellName, cellData, pathData (pin-specific)
DCL syntax	<pre>EXPOSE(dpcmGetPinTiePolarity): passed(pin: pinPointer) result(int: arrayIndex);</pre>
C syntax	<pre>typedef struct { DCM_INTEGER arrayIndex; } T_GetPinTiePolarity; int dpcmGetPinTiePolarity (const DCM_STD_STRUCT *std_struct, T_GetPinTiePolarity *rtn, DCM_PIN pinPointer);</pre>

This returns the *TiePolarity* property for a pin back to the application. This index shall be used with the array of pin-polarity strings returned by *dpcmGetPinPolarityArray* (see 10.23.12.1.8) for that same pin.

10.23.13.2 dpcmGetPinReadPolarity

Table 345 provides information on dpcmGetPinReadPolarity.

Table 345—dpcmGetPinReadPolarity

Function name	dpcmGetPinReadPolarity
Arguments	Pin pointer
Result	Index into an array of pin-polarity strings
Standard Structure fields	block, CellName, cellData, pathData (pin-specific)
DCL syntax	<pre>EXPOSE(dpcmGetPinReadPolarity): passed(pin: pinPointer) result(int: arrayIndex);</pre>
C syntax	<pre>typedef struct { DCM_INTEGER arrayIndex; } T_GetPinReadPolarity; int dpcmGetPinReadPolarity (const DCM_STD_STRUCT *std_struct, T_GetPinReadPolarity *rtn, DCM_PIN pinPointer);</pre>

This returns the *ReadPolarity* property for a pin back to the application. This index shall be used with the array of pin-polarity strings returned by *dpcmGetPinPolarityArray* (see 10.23.12.1.8) for that same pin.

10.23.13.3 dpcmGetPinWritePolarity

Table 346 provides information on dpcmGetPinWritePolarity.

Table 346—dpcmGetPinWritePolarity

Function name	dpcmGetPinWritePolarity
Arguments	Pin pointer
Result	Index into an array of pin-polarity strings
Standard Structure fields	block, CellName, cellData, pathData (pin-specific)
DCL syntax	<pre>EXPOSE(dpcmGetPinWritePolarity): passed(pin: pinPointer) result(int: arrayIndex);</pre>
C syntax	<pre>typedef struct { DCM_INTEGER arrayIndex; } T_GetPinWritePolarity; int dpcmGetPinWritePolarity (const DCM_STD_STRUCT *std_struct, T_GetPinWritePolarity *rtn, DCM_PIN pinPointer);</pre>

This returns the *WritePolarity* property for a pin back to the application. This index shall be used with the array of pin-polarity strings returned by *dpcmGetPinPolarityArray* (see 10.23.12.1.8) for that same pin.

10.23.13.4 dpcmGetSimultaneousSwitchTimes

Table 347 provides information on dpcmGetSimultaneousSwitchTimes.

Table 347—dpcmGetSimultaneousSwitchTimes

Function name	dpcmGetSimultaneousSwitchTimes
Arguments	None
Result	Array of skew limits
Standard Structure fields	CellName, block, cellData (vectorTiming or vectorPower), pathData (vectorTiming or vectorPower), calcMode
DCL syntax	<pre>typedef(skewLimit): result(string[*]: listOfPins; double: skewLimit); EXPOSE(dpcmGetSimultaneousSwitchTimes): result(skewLimit[*]: skewLimitArray);</pre>
C syntax	<pre>typedef struct { DCM_STRING_ARRAY *listOfPins; DCM_DOUBLE skewLimit; } DCM_SkewLimits; typedef DCM_SkewLimits *DCM_SkewLimits_ARRAY; typedef struct { DCM_SkewLimits_ARRAY *skewLimitsArray; } T_SkewLimitsArray; int dpcmGetSimultaneousSwitchTimes (const DCM_STD_STRUCT *std_struct, T_SkewLimitsArray *rtn);</pre>

This returns an array of skew limits associated with a vector expression graph back to the application.

Each *skewLimit* structure shall contain a list of pin names and a *skewLimit* value. If the separation in time between transitions on two or more of these pins is less than or equal to the *skewLimit*, then those transitions shall be considered simultaneous. An expression can have more than one set of skew limits. Each skew limit is treated independently. Pins that are listed in more than one skew limit shall satisfy all the skew limits where they are listed.

The *skewLimit* value shall be no smaller than the time resolution specified in *dpcmGetTimeResolution* (see 10.18.8.10). If this function is not implemented or returns an error code, the application shall consider two signal transitions to be simultaneous if the time difference between them is less than or equal to the time resolution.

If a vector expression graph contains one or more simultaneous switching operators, simultaneous switching shall be interpreted according to the definitions of these operators.

10.23.13.5 appGetSwitchingBits

Table 348 provides information on *appGetSwitchingBits*.

Table 348—*appGetSwitchingBits*

Function name	<i>appGetSwitchingBits</i>
Arguments	None
Result	Number of switching bits
Standard Structure fields	CellName, block, cellData (power), pathData (pin-specific)
DCL syntax	EXTERNAL(<i>appGetSwitchingBits</i>): passed(string[*]: pinsToMonitorForSwitching) result(int: NumBits);
C syntax	typedef struct { DCM_INTEGER numBits; } T_appGetSwitchingBits; int <i>appGetSwitchingBits</i> (const DCM_STD_STRUCTURE *std_struct, T_appGetSwitchingBits *rtn DCM_STRING_ARRAY *pinsToMonitorForSwitching);

This is defined in the *power* and *vectorPower* domains. *appGetSwitchingBits* requests the application supply the number of bits switching on a bus during processing of a *dpcmGetCellPowerWithState* or a *dpcmGetCellVectorPower*, respectively. The bus specification is different for *power* and *vectorPower* domains.

For the *power* domain, the bus is specified by a set of pins whose names are contained in the *pinsToMonitorForSwitching* array. The array needs to contain more than one entry for it to specify a bus. The application shall assume the pins whose names are specified in the array constitute a bus and return the number of the pins from the array that switched state during the applicable interval.

For the *vectorPower* domain, the *pinsToMonitorForSwitching* array shall contain the name of a single pin, which specifies a bus pin. It is an error to include more than one name of a pin in the array. The application shall return the number of bits of the specified bus which switched the state during the applicable interval.

10.23.13.6 dpcmGetFrequencyLimit

Table 349 provides information on *dpcmGetFrequencyLimit*.

Table 349—dpcmGetFrequencyLimit

Function name	dpcmGetFrequencyLimit
Arguments	None
Result	Minimum frequency limit, Maximum frequency limit
Standard Structure fields	CellName, block, cellData, pathData, slew->early, slew->late, calcMode
DCL syntax	<pre>EXPOSE(dpcmGetFrequencyLimit): result(double: minFrequency, maxFrequency);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE minFrequency; DCM_DOUBLE maxFrequency; } T_GetFrequencyLimitResults; int dpcmGetFrequencyLimit (const DCM_STD_STRUCT *std_struct, T_GetFrequencyLimitResults *rtn);</pre>

This returns the minimum and maximum frequency limit for a given path or pin back to the application.

This new call does not need to be used in conjunction with *dpcmGetCellPowerInfo()*. The vector where the frequency limit applies is identified by *pathData* in the *Standard Structure* (see 10.23.13.15).

10.23.13.7 appGetPinFrequency

Table 350 provides information on appGetPinFrequency.

Table 350—appGetPinFrequency

Function name	appGetPinFrequency
Arguments	Pin pointer
Result	Frequency value
Standard Structure fields	CellName, block, cellData, pathData (pin-specific), calcMode
DCL syntax	<pre>EXTERNAL(appGetPinFrequency): passed(pin: pinPointer) result(double: frequency);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE frequency; } T_GetPinFrequency_Results; int appGetPinFrequency (const DCM_STD_STRUCT *std_struct, T_GetPinFrequency_Results *rtn, DCM_PIN pinPointer);</pre>

This permits the library to request the current frequency on a particular pin from the application. *Frequency* is defined as the total number of rise-fall transition pairs per time unit.

10.23.13.8 dpcmGetBasePinFrequency

Table 351 provides information on dpcmGetBasePinFrequency.

Table 351—dpcmGetBasePinFrequency

Function name	dpcmGetBasePinFrequency
Arguments	Pin pointer
Result	Frequency value
Standard Structure fields	calcMode
DCL syntax	<pre>EXPOSE(dpcmGetBasePinFrequency): passed(pin: pinPointer) result(double: frequency);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE frequency; } T_GetBasePinFrequency_Results; int dpcmGetBasePinFrequency (const DCM_STD_STRUCT *std_struct, T_GetBasePinFrequency_Results *rtn, DCM_PIN pinPointer);</pre>

This returns the default pin frequency value back to the application.

10.23.13.9 dpcmGetPinJitter

Table 352 provides information on dpcmGetPinJitter.

Table 352—dpcmGetPinJitter

Function name	dpcmGetPinJitter
Arguments	Pin pointer
Result	Jitter value
Standard Structure fields	CellName, block, cellData, pathData (pin-specific), slew->early, slew->late, sinkEdge, calcMode
DCL syntax	<pre>EXPOSE(dpcmGetPinJitter): passed(pin: pinPointer) result(double: jitter);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE jitter; } T_GetPinJitterResults; int dpcmGetPinJitter (const DCM_STD_STRUCT *std_struct, T_GetPinJitterResults *rtn, DCM_PIN pinPointer);</pre>

This returns the jitter for a given pin back to the application. Pin *jitter* is the window of uncertainty associated with a clock edge. The library shall determine what action to perform when the returned values are outside the library's *min/max* validity range, but the library shall not return a non-zero return code when validity ranges are violated. The validity ranges shall not be exposed to the application.

10.23.13.10 dpcmGetInductanceLimit

Table 353 provides information on dpcmGetInductanceLimit.

Table 353—dpcmGetInductanceLimit

Function name	dpcmGetInductanceLimit
Arguments	Pin pointer
Result	Minimum inductance limit, Maximum inductance limit
Standard Structure fields	CellName, block, cellData, pathData (pin-specific), calcMode
DCL syntax	<pre>EXPOSE(dpcmGetInductanceLimit): passed(pin: pinPointer) result(double: minInductance, maxInductance);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE minInductance; DCM_DOUBLE maxInductance; } T_GetInductanceLimitResults; int dpcmGetInductanceLimit (const DCM_STD_STRUCT *std_struct, T_GetInductanceLimitResults *rtn, DCM_PIN pinPointer);</pre>

This returns the minimum and maximum series inductance permitted on the pin back to the application.

10.23.13.11 dpcmGetOutputSourceResistances

Table 354 provides information on dpcmGetOutputSourceResistances.

Table 354—dpcmGetOutputSourceResistances

Function name	dpcmGetOutputSourceResistances
Arguments	None
Result	Early edge resistance value, Late edge resistance value
Standard Structure fields	CellName, fromPoint, toPoint, calcMode, slew.early, slew.late, block, sourceEdge, sinkEdge, sourceMode, sinkMode, pathData (arc-specific), cellData
DCL syntax	<pre>EXPOSE(dpcmGetOutputSourceResistances): result(double: earlyResistance, lateResistance);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE earlyResistance; DCM_DOUBLE lateResistance; } T_OutputSourceResistances; int dpcmGetOutputSourceResistances (const DCM_STD_STRUCT *std_struct, T_OutputSourceResistances *rtn);</pre>

For a modeled arc, this returns two output source resistance values, an early value and a late value, back to the application. These values are the equivalent linear resistances presented by a transitioning output pin at the end of an arc. This function shall not be called for interconnect arcs.

10.23.13.12 appSetPull

Table 355 provides information on appSetPull.

Table 355—appSetPull

Function name	appSetPull
Arguments	Pull type
Result	None
Standard Structure fields	CellName, block, calcMode, cellData, pathData (pin-specific)
DCL syntax	EXTERNAL (appSetPull) : passed(int: pull_type) result(int: ignore);
C syntax	<pre>typedef enum DCM_PullType { DCM_PULL_UP, DCM_PULL_DOWN, DCM_PULL_BOTH } DCM_PullType; typedef struct { DCM_INTEGER ignore; } T_Ignore; int appSetPull(const DCM_STD_STRUCT *std_struct, T_Ignore *rtn, DCM_PullType pull_type);</pre>

The library calls *appSetPull* to register with the application that a pull up/down resistance combination exists in series with a rail voltage. *appSetPull* passes the enumeration *DCM_PullType*, which is defined in Table 356.

During model elaboration, the library shall call this function immediately after calling *newNetSinkPropagateSegment* or *newNetSourcePropagateSegment*.

Table 356—DCM_PullType

Enumerator	Enumeration	Description
DCM_PULL_UP	0	Only pull-up exists.
DCM_PULL_DOWN	1	Only pull-down exists.
DCM_PULL_BOTH	2	Both pull-up and pull-down exist.

10.23.13.13 dpcmGetPull

Table 357 provides information on dpcmGetPull.

Table 357—dpcmGetPull

Function name	dpcmGetPull
Arguments	Pin pointer
Result	Pull-up resistance value, Pull-down resistance, value, Index to rail 1, Index to rail 2
Standard Structure fields	CellName, block, calcMode, cellData, pathData (pin-specific)
DCL syntax	EXPOSE (dpcmGetPull) : passed(pin: pinPointer) result(double: pullUpResistance, pullDownResistance; int: pullUpRailIndex, pullDownRailIndex);

C syntax	<pre>typedef struct { DCM_DOUBLE pullUpResistance; DCM_DOUBLE pullDownResistance; DCM_INTEGER pullUpRailIndex; DCM_INTEGER pullDownRailIndex; } T_PullResults; int dpcmGetPull(const DCM_STD_STRUCT *std_struct, T_PullResults *rtn, DCM_PIN pinPointer);</pre>

The application calls *dpcmGetPull* to determine the value of the pull up/down resistance in series with a rail voltage associated with the *pinPointer*.

This call shall only be used after *pull_type* has been set by *appSetPull* for this pin. The value of *pull_type* determines the corresponding valid resistances and rail indices. The values of invalid resistances and rail indices shall be ignored.

If the library contains a cell with a tie down to *ground*, then *dpcmGetRailVoltageArray()* shall name *ground* as a rail.

10.23.13.14 dpcmGetPinDriveStrength

Table 358 provides information on *dpcmGetPinDriveStrength*.

Table 358—dpcmGetPinDriveStrength

Function name	dpcmGetPinDriveStrength
Arguments	Pin pointer
Result	Pin drive strength
Standard Structure fields	CellName, block, cellData, pathData (pin-specific), calcMode
DCL syntax	<pre>EXPOSE(dpcmGetPinDriveStrength): passed(pin: pinPointer) result(double drive_strength);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE drive_strength; } T_GetPinDriveStrengthResults; int dpcmGetPinDriveStrength (const DCM_STD_STRUCT *std_struct, T_GetPinDriveStrengthResults *rtn, DCM_PIN pinPointer);</pre>

This returns the relative pin drive strength back to the application. The drive strength has no units of measurement; it is merely a ranking of a cell's drive strength for a particular transition relative to others in the library. The library shall determine what action to perform when the values returned are outside the *min/max* validity range, but the library shall not return a non-zero return code when validity ranges are violated. The validity ranges shall not be exposed to the application.

10.23.13.15 dpcmGetCellVectorPower

Table 359 provides information on *dpcmGetCellVectorPower*.

Table 359—dpcmGetCellVectorPower

Function name	dpcmGetCellVectorPower
Arguments	None
Result	Energy per rail, Static power per rail, Total energy, Total static power
Standard Structure fields	CellName, block, cellData (power), pathData (for vectorPower), calcMode
DCL syntax	<pre>EXPOSE(dpcmGetCellVectorPower): result(double[*]: energyPerRail, staticPowerPerRail; double: totalEnergy, totalStaticPower);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE_ARRAY *energyPerRail; DCM_DOUBLE_ARRAY *staticPowerPerRail; DCM_DOUBLE totalEnergy; DCM_DOUBLE totalStaticPower; } T_cellVectorPower; int dpcmGetCellVectorPower (const DCM_STD_STRUCT *std_struct, T_cellVectorPower *rtn);</pre>

This returns the power associated with a cell for a given state vector back to the application.

The vector used in calculating the power is identified by *pathData* in the *Standard Structure*, as opposed to *groupIndex/conditionIndex* in *dpcmGetPowerWithState*. The application is responsible for storing *pathData* pointers for power vectors obtained during the execution of *modelSearch* and initializing the *pathData* field of the *Standard Structure* with valid data before calling *dpcmGetCellVectorPower*. The energy/power per rail is optional. If the energy and static power per rail are not computed, their corresponding arrays shall be empty.

The application shall not call this API in conjunction with *dpcmGetCellPowerInfo*.

10.23.14 Connectivity

This subclause defines all the APIs for connectivity.

10.23.14.1 dpcmGetPinCellConnectivityArrays

Table 360 provides information on dpcmGetPinConnectivityArrays.

Table 360—dpcmGetPinConnectivityArrays

Function name	dpcmGetPinConnectivityArrays
Arguments	None
Result	Cell level connectivity
Standard Structure fields	block, CellName, cellData, pathData (pin-specific)
DCL syntax	<pre>EXPOSE(dpcmGetPinCellConnectivityArrays): result(int[*]: dontConnectPinArray, mustConnectPinArray, mayConnectPinArray);</pre>

C syntax	<pre>typedef struct { DCM_INTEGER_ARRAY *dontConnectPinArray, *mustConnectPinArray, *mayConnectPinArray; } T_GetPinCellConnectivityArrays; int dpcmGetPinConnectivityArrays (const DCM_STD_STRUCT *std_struct, T_GetPinCellConnectivityArrays *rtn);</pre>

This returns arrays of pin indices, which have defined cell level connectivity rules relative to the specified pin, back to the application. Pins within the same cell can appear in one of the three arrays, the connectivity rule of which is applied with respect to the pin specified in the *Standard Structure*.

10.23.14.2 dpcmGetLibraryConnectClassArray

Table 361 provides information on dpcmGetLibraryConnectClassArray.

Table 361—dpcmGetLibraryConnectClassArray

Function name	dpcmGetLibraryConnectClassArray
Arguments	None
Result	List of pins subject to connectivity rules
Standard Structure fields	None
DCL syntax	EXPOSE(dpcmGetLibraryConnectClassArray): result(string[*]: connectClassNames);
C syntax	<pre>typedef struct { DCM_STRING_ARRAY *connectClassNames; } T_GetLibraryConnectClassArray; int dpcmGetLibraryConnectClassArray (const DCM_STD_STRUCT *std_struct, T_GetLibraryConnectClassArray *rtn);</pre>

This returns the list of all possible *ConnectClass* names contained within a library representing a technology back to the application. Each *ConnectClass* represents a set of pins subjected to connectivity rules. This API is called prior to calling *dpcmGetLibraryConnectivityRules* (see Table 362) in order to obtain the set of connectivity rules based on connect class.

10.23.14.3 dpcmGetLibraryConnectivityRules

Table 362—dpcmGetLibraryConnectivityRules

Function name	dpcmGetLibraryConnectivityRules
Arguments	None
Result	Connectivity object name and lists of indexing pin connection classes
Standard Structure fields	None
DCL syntax	EXPOSE(dpcmGetLibraryConnectivityRules): result(int[*,*]: connectivityRuleArray);

C syntax	<pre> typedef enum DCM_ConnectRules { DCM_DontConnect, DCM_MustConnect, DCM_MayConnect } DCM_ConnectRules; typedef struct { DCM_ConnectRules *connectivityRuleArray; } T_GetLibraryConnectivityRules; int dpcmGetLibraryConnectivityRules (const DCM_STD_STRUCT *std_struct, T_GetLibraryConnectivityRules *rtn); </pre>

This returns a matrix of connectivity rules between all connect classes in the library back to the application. This API is called after calling *dpcmGetLibraryConnectClassArray* (see 10.23.14.2) in order to obtain the set of connectivity rules based on connect class.

The returned data is in the form of a two-dimensional square array of *DCM_ConnectRules* enumeration values (integers) as defined in Table 363. The indices into this array correspond directly to the order of connect class names as obtained in 10.23.14.2.

Table 363—DCM_ConnectRules

Enumerator	Enumeration	Description
DCM_DontConnect	0	Do not connect
DCM_MustConnect	1	Must/shall connect
DCM_MayConnect	2	Might connect

Once the connect class-based rule set is obtained, the application can determine interconnect level connectivity by obtaining the library connect class array indices of the pins in question using *dpcmGetPinConnectClass* (see 10.23.12.1.11). These indices index into the connectivity rules array, resulting in the corresponding rule for these pins.

10.23.15 Control of timing arc existence and state

An application shall use the procedure described in the following list to obtain information on the conditional behavior of a timing segment from the *timing* domain of a DPCM. The following steps shall be completed as part of the application's response to a call made by the DPCM to *newPropagateSegment* (for a timing measurement) or *newAltTestSegment* (for a timing check):

- The application calls *dpcmGetExistenceGraph* to obtain a graphical description or function graph of the existence expression for the segment. In response, the DPCM shall call *newPropagateSegment* and *newTimingPin* as needed to describe this expression. The application shall use the resulting existence condition graph to determine, in part, whether the segment is active for any instance of the timing model to which the segment belongs.
- The application calls *dpcmGetControlExistence* to obtain the mode settings where the segment is active and a textual representation of the segment's existence expression. The latter is intended for informational purposes only, and the application shall not use it to determine whether the segment is active. The application shall use the mode settings so obtained to determine, in part, whether the segment is active for any instance of the timing model to which the segment belongs.
- The application calls *dpcmGetTimingStateGraphs* to obtain function graphs of the expressions used to determine the timing state of the segment. In response, the DPCM shall call *newPropagateSegment* and *newTimingPin* as needed to describe these expressions. There shall be

one state condition graph for each timing state of the segment other than the default state. The application shall use the resulting graphs to determine the timing state of the segment for any instance of the timing model to which the segment belongs.

- d) If textual representations of the timing state expressions are needed, the application can call *dpcmGetTimingStateStrings* for each timing state of the segment. These descriptions are intended for informational purposes only, and the application shall not use them to determine the timing state of the segment.

10.23.15.1 dpcmGetExistenceGraph

Table 364 provides information on *dpcmGetExistenceGraph*.

Table 364—*dpcmGetExistenceGraph*

Function name	<i>dpcmGetExistenceGraph</i>
Arguments	None
Result	Boolean value indicating whether the graph has been created
Standard Structure fields	<i>block</i> , <i>CellName</i> , <i>fromPoint</i> , <i>toPoint</i> , <i>pathData</i> (timing-arc-specific), <i>cellData</i> (timing)
DCL syntax	<pre>EXPOSE (dpcmGetExistenceGraph) : result (int: graphCreated);</pre>
C syntax	<pre>typedef struct { DCM_INTEGER graphCreated; } T_graphCreated; int dpcmGetExistenceGraph (const DCM_STD_STRUCT *std_struct, T_graphCreated *rtn);</pre>

This causes the library to supply a function graph of the existence condition for the segment identified by the *pathData* field in the *Standard Structure*.

When this function is called, the library shall call *newPropagateSegment* and *newTimingPin* as needed to represent the conditional expression that controls the existence of the specified segment. A *pathData* pointer present in the *Standard Structure* when each of these functions is called shall be associated with each behavior arc and node, respectively, so defined.

The *PathDataBlock* structures addressed by these pointers shall contain data used by the application to determine the role each arc and node plays in the existence-condition function graph, as described in 10.23.3.1. A zero-valued *pathData* pointer shall not be used when these functions are called in this context, except for when *newTimingPin* is called to define the terminal node for the graph. For this node, and only this node, the *pathData* pointer shall have a value of zero (0). *The application shall identify this node as the terminal node of the graph.*

In DCL, an existence-condition function graph can be constructed using a *FUNCTION* clause as specified in . For this purpose, such clauses shall be limited to the assignment form; use of a precedence expression shall be illegal.

The modeling of the existence-condition function graph for a segment shall occur in a separate elaboration sequence initiated by the application during the elaboration of the timing model for a cell and after elaboration of the segment. The resulting graph shall be contained within the “timing” domain.

If, and only if, the expression represented by this graph evaluates at *its terminal* node to *false*, the segment for which *dpcmGetExistenceGraph* was called shall be disabled by the application.

This evaluation shall be independent of whether the cell is in a steady state or transitioning into that state. In particular, the pins at the ends of the segment shall have, for the purpose of this evaluation, the values associated with the completion of the signal transitions that define, in part, the segment. For example, the pins at the ends of a segment defined for rising edges at both pins shall be assigned values of logical one during evaluation of this graph.

In all other respects, the semantics of this graph shall correspond to those of the “Group Condition Language” (see 8.11). Specifically, the node operators shall be limited to those corresponding to entries in Table 20 of that standard.

Until the call to *dpcmGetExistenceGraph* returns, all data related to the application shall be used for this purpose only and shall not have any other effect on the application.

This function shall return an integer value indicating whether an existence condition function graph has been created. If a function graph has been created, a value of 1 shall be returned. If there is no existence condition for the specified segment, a value of 0 shall be returned. This shall not be considered to be an error condition and the error code returned by the function shall be 0 in such cases. The existence control string returned by the DPCM to the application via the interface function *dpcmGetControlExistence* is supplied for informational purposes only and shall not be used by the application to determine the existence of the segment. The application shall use the existence condition function graph returned by *dpcmGetExistenceGraph* for this purpose.

10.23.15.2 dpcmGetTimingStateGraphs

Table 365 provides information on *dpcmGetTimingStateGraphs*.

Table 365—dpcmGetTimingStateGraphs

Function name	<i>dpcmGetTimingStateGraphs</i>
Arguments	None
Result	Boolean value indicating whether the graphs have been created
Standard Structure fields	block, CellName, fromPoint toPoint, pathData (timing-arc-specific), cellData (timing)
DCL syntax	EXPOSE(<i>dpcmGetTimingStateGraphs</i>) : result(int: graphsCreated);
C syntax	typedef struct { DCM_INTEGER graphsCreated; } T_graphsCreated; int <i>dpcmGetTimingStateGraphs</i> (const DCM_STD_STRUCT *std_struct, T_graphsCreated *rtn);

This causes the library to supply function graphs of the state condition expressions for the segment identified by the *pathData* field in the *Standard Structure*.

When this function is called, the library shall call *newPropagateSegment*, *newNetSinkPropagateSegments*, and *newTimingPin* as needed to represent the state-condition expressions for the specified segment. Each such expression shall be represented by a separate state-condition function graph. A *pathData* pointer present in the *Standard Structure* when each of these functions is called shall be associated with each behavior arc and node, respectively, so defined.

The *PathDataBlock* structures addressed by these pointers shall contain data used by the application to determine the role each arc and node plays in each state-condition function graph, as described in 10.23.3.1. A zero-valued *pathData* pointer shall not be used when these functions are called in this context, except when *newTimingPin* is called for the terminal node of each graph. For these nodes, and only these nodes, the *pathData* pointer shall have a value of zero (0) when *newTimingPin* is called.

In DCL, a state-condition function graph can be constructed using a degenerative form of a *VECTOR* clause, as specified in 7.16.9.14.

The modeling of the state-condition function graphs for a timing segment shall occur in a separate elaboration sequence initiated by the application during the elaboration of the timing model for a cell and after elaboration of that segment. These graphs shall be contained within the “timing” domain, and each graph shall terminate in a node having one of the following operators:

- DCM_PRIMITIVE_VECTOR_DELAY_TARGET
- DCM_PRIMITIVE_VECTOR_CHECK_TARGET

There shall be only one such node in each graph. Each terminal node shall be created using a two-step process, as follows:

- a) The library shall call *newTimingPin* with three arguments: a *Standard Structure*, a string representing the node’s name, and a pointer to the location into which the application shall place a newly created pin handle for the node. The *pathData* pointer in the *Standard Structure* argument shall be zero (0). In response to this call, the application shall create a node with a name that is the same as the string argument and return the new pin handle for this node as described previously.
- b) The library shall then call *newNetSinkPropagateSegments* with the terminal node as the *importPin* argument. The *sinkPin* and *delayMatrix* arguments shall both have values of zero (0). The *fromPoint* and *toPoint* fields in the *Standard Structure* shall be the same as for the timing segment for which the graph is being defined. When this call is made to the application, the *pathData* field of the *Standard Structure* shall contain data for the terminal node, and the application shall associate this *pathData* with the node created via the previous *newTimingPin* call.

The order in which these terminal nodes are presented to the application shall define an ordinal state space for the segment for which *dpcmGetTimingStateGraphs* was called. To determine the timing state for this segment, the application shall evaluate the state-condition function graphs in this order. The first graph that evaluates to *true* at its terminal node shall determine the timing state of the segment.

The application shall then use the *pathData* pointer associated with the terminal node from that function graph when obtaining the delay, slew, or bias values of the segment from the DPCM. This *pathData* pointer shall be communicated to the application in the *Standard Structure* passed via the call to *newTimingPin*, which is used to define the terminal node. If none of the function graphs evaluate to *true*, the application shall use the *pathData* pointer supplied by the DPCM when the segment itself is defined. The DPCM shall then return the default values associated with the segment.

This evaluation shall be independent of whether the cell is in a steady state or transitioning into that state. In particular, the pins at the ends of the segment shall have, for the purpose of this evaluation, the values associated with the completion of the signal transitions that define, in part, the segment. For example, the pins at the ends of a segment defined for rising edges at both pins shall be assigned values of logical one during evaluation of these function graphs.

In all other respects, the semantics of each function graph shall correspond to those of the “Group Condition Language” (see 8.11). Specifically, the node operators shall be limited to those corresponding to the entries in Table 20 of that standard.

This representation shall be contained within the *timing* domain. Until the call to *dpcmGetTimingStateGraphs* returns, all data related to the application shall be used for this purpose only and shall not have any other effect on the application.

The application shall obtain the condition expression string and an optional set of labels associated with each state via the function *dpcmGetTimingStateStrings*. An application that distinguishes timing states using

their labels instead of their conditions shall use the *pathData* pointers described above to obtain the segment values associated with each set of labels.

Segments having multiple states with the same condition expression, but different labels, are allowed. These states shall be distinguished from each other only by applications that use state labels for that purpose. An application that uses condition expressions to distinguish between states shall always encounter the first of these states in the state space when using the evaluation procedure described previously. The DPCM shall place the state that is appropriate for such applications before all others in the state space that has the same condition expression.

This function shall return an integer value indicating whether any state condition function graphs have been created. If function graphs have been created, a value of 1 shall be returned. If there are no timing states for the specified segment, a value of 0 shall be returned. This shall not be considered to be an error condition, and the function shall return an error code of 0 in such cases.

10.23.15.3 dpcmGetTimingStateStrings

Table 366 provides information on dpcmGetTimingStateStrings.

Table 366—dpcmGetTimingStateStrings

Function name	dpcmGetTimingStateStrings
Arguments	None
Result	Condition expression, Labels for state timing
Standard Structure fields	block, CellName, fromPoint, toPoint, pathData (state-condition-graph-specific), cellData (timing)
DCL syntax	<pre>EXPOSE(dpcmGetTimingStateStrings): result(string: conditionExpression; string[*]: stateLabels);</pre>
C syntax	<pre>typedef struct { DCM_STRING conditionExpression; DCM_STRING_ARRAY *stateLabels; } T_timingStateStrings; int dpcmGetTimingStateStrings (const DCM_STD_STRUCT *std_struct, T_timingStateStrings *rtn);</pre>

This returns the condition expression and the labels for the timing state identified by the *pathData* field in the *Standard Structure* back to the application. Each timing state shall be defined by a state condition function graph provided by the DPCM to the application via a call to *dpcmGetTimingStateGraphs*. The *pathData* field shall be obtained from the terminal node of the associated state condition function graph.

The condition-expression string is supplied by the DPCM to the application for informational purposes only and shall not be used by the latter to determine the timing state of the propagation or test segment with which it is associated. The application shall use the state condition function graphs returned by the function *dpcmGetTimingStateGraphs* for this purpose.

The condition expression shall be a textual representation of the corresponding state condition function graph. The syntax of the expression is described in 8.11. The expression shall not be an empty string.

The state labels are optional textual identifiers associated with the corresponding timing state. These labels shall be returned as an array of strings. All strings in this array shall have at least one character. A zero-length array shall indicate the absence of any labels for the timing state.

10.23.15.4 dpcmGetVectorEdgeNumbers

Table 367 provides information on dpcmGetVectorEdgeNumbers.

Table 367—dpcmGetVectorEdgeNumbers

Function name	dpcmGetVectorEdgeNumbers
Arguments	None
Result	From pin edge, To pin edge
Standard Structure fields	block, CellName, fromPoint, toPoint, pathData (timing-arc-specific), cellData (timing)
DCL syntax	EXPOSE(dpcmGetVectorEdgeNumbers): result(int: fromEdgeNumber, toEdgeNumber);
C syntax	<pre>typedef struct { DCM_INTEGER fromEdgeNumber; DCM_INTEGER toEdgeNumber; } T_vectorEdgeNumbers; int dpcmGetVectorEdgeNumbers (const DCM_STD_STRUCT *std_struct, T_vectorEdgeNumbers *rtn);</pre>

This returns numbers for the from-pin and to-pin edges in a timing vector for the segment identified by the *pathData* field in the *Standard Structure* back to the application.

A timing vector can contain multiple edges of the signals associated with the from-pins and to-pins of the related propagation or test segment. Each timing arc (described in the context of a timing vector) has a unique from-edge and to-edge number, both of which are returned by this call. Each edge for each pin shall be assigned a consecutive integer number, with the first edge for a pin being assigned the number 0 and subsequent edges for that pin being numbered in ascending order. The edges for each pin shall be numbered separately from those of the other pin. For each pin, the application shall determine when edge 0 occurs and shall do so in a manner that preserves the relative temporal relationships between all segments and signal edges in a cell whenever possible.

The application shall evaluate a propagation segment associated with a vector in response to the from-pin edge identified using this function. This evaluation shall produce the to-pin edge also identified via the function.

The application shall evaluate a propagation or a test segment associated with a vector in response to both the from-pin and to-pin edges identified using this function.

10.23.16 Modeling cores

Table 368 through Table 370 are new interface functions intended to provide signal multiplication, division, and generation characteristics of cell models to static timing applications. The information provided to the application via these functions is an abstraction of complex functional behavior that, if presented directly, would not be suitable for a static timing tool. Through the use of these data, such an application can generate more accurate timing results than is otherwise possible. Many static timing applications allow users to provide similar information as attributes to be added to timing arcs already present in cell timing models. The presentation of such information as arc attributes makes use of these data straightforward for applications that already support similar constructs.

10.23.16.1 appSetSignalDivision

Table 368—appSetSignalDivision

Function name	appSetSignalDivision
Arguments	First input/reference direction, Edge list/initial output edge direction, Repeat edge interval
Result	None
Standard Structure fields	CellName, block, fromPoint, toPoint, pathData (timing-arc-specific), cellData (timing)
DCL syntax	<pre>EXTERNAL (appSetSignalDivision) : passed(int: inputEdgeZero; int[*]: inputEdgeNumbers; int: initialOutputEdge, repeatEdgeInterval) result(int: ignore);</pre>
C syntax	<pre>typedef struct { DCM_INTEGER ignore; } T_Ignore; int appSetSignalDivision (const DCM_STD_STRUCT *std_struct, T_Ignore *trtn, DCM_EdgeTypes inputEdgeZero, DCM_INTEGER_ARRAY *inputEdgeNumbers, DCM_EdgeTypes initialOutputEdge, DCM_INTEGER repeatEdgeInterval);</pre>

This sends data describing input signal sampling or output signal rate division characteristics for the segment last defined during timing model elaboration or for which calculation is currently being performed.

During model elaboration, the library shall call this function immediately after calling *newPropagateSegment* or *newAltTestSegment* to define a timing segment. The specified characteristics shall apply to the timing segment so defined.

During an application call to the library calculation functions *delay*, *slew*, or *check*, these characteristics shall apply to the segment for which that calculation function is being called.

These data shall be expressed in terms of edges of the input or reference signal to which the segment is sensitive. Each such edge shall be assigned a consecutive integer number, with the first edge being assigned the number 0 and subsequent edges being numbered in ascending order. The application shall determine when edge 0 occurs and shall do so in a manner that preserves the relative temporal relationships between all segments in a cell whenever possible.

Without division characteristic data, a timing segment shall be evaluated by the application in response to all of these edges. However, a segment for which such data are present shall be evaluated in a manner prescribed via these data.

The division characteristics for a segment consist of four parameters.

The first parameter is the edge direction for edge 0 of the input or reference signal. This parameter is of type DCM_EdgeTypes and shall have one of the following values:

- DCM_RisingEdge
- DCM_FallingEdge

If the segment is sensitive to only one edge direction of the input or reference signal, this parameter shall be the same as that edge direction.

The second parameter, the edge list, is a list of numbers of the edges of the input or reference signal for which the segment shall be evaluated. While the first edge number in the list need not be 0, nor do these numbers need be contiguous, they shall appear in ascending order.

The third parameter is the edge direction for the signal transition at the output of a propagation segment, which is produced by the first edge in the edge list. This parameter is of type `DCM_EdgeTypes` and shall have one of the following values:

- `DCM_RisingEdge`
- `DCM_FallingEdge`
- `DCM_BothEdges`

If `DCM_RisingEdge` is specified, the first output edge shall be a rising edge. If `DCM_FallingEdge` is specified, the first output edge shall be a falling edge. If `DCM_BothEdges` is specified, the first output edge can be either rising or falling.

Each subsequent output edge shall be of opposite polarity from the previous one (i.e., each rising edge shall be followed by a falling edge and each falling edge shall be followed by a rising edge). Each input edge in the input edge list shall produce an output edge.

This parameter shall be ignored if the segment is a test segment.

The fourth parameter is the number of input edges that, together with edge 0, defines an interval for use in repetition of the pattern described via the first three parameters. If greater than 0, this parameter shall be greater than the last edge number specified in the edge list. The application shall then consider the pattern to repeat indefinitely, both before and after the original interval.

If the third parameter is `DCM_RisingEdge` or `DCM_FallingEdge` (and the segment is not a test segment), the edge list shall have an even number of entries.

If the fourth parameter is 0, the pattern shall not be repeated and the edge list can have any number of entries.

The application shall apply these characteristics to a segment as defined as a unit via a call by the DPCM to *newPropagateSegment* or *newAltTestSegment*. These characteristics shall not be applied separately to individual timing arcs that the application can generate in response to such a call. In particular, the application shall consider the segment characteristics to be those described in the delay or test matrix referenced by that call.

If the specified segment has division characteristics, it shall not also have multiplication nor signal generation characteristics.

10.23.16.2 appSetSignalMultiplication

Table 369—appSetSignalMultiplication

Function name	appSetSignalMultiplication
Arguments	Multiplication factor, Initial output edge, Duty cycle
Result	None
Standard Structure fields	block, CellName, fromPoint, toPoint, pathData (timing-arc-specific), cellData (timing)

DCL syntax	<pre>EXTERNAL(appSetSignalMultiplication): passed(int: multFactor, initialOutputEdge; float: dutyCycle) result(int: ignore);</pre>
C syntax	<pre>typedef struct { DCM_INTEGER ignore; } T_Ignore; int appSetSignalMultiplication (const DCM_STD_STRUCT *std_struct, T_Ignore *rtn, DCM_INTEGER multFactor, DCM_EdgeTypes initialOutputEdge, DCM_FLOAT dutyCycle);</pre>

This sends data describing output signal rate multiplication characteristics for the segment last defined during timing model elaboration or for which calculation is currently being performed.

During model elaboration, the library shall call this function immediately after calling *newPropagateSegment* to define a timing segment. The specified characteristics shall apply to the timing segment so defined.

During an application call to the library calculation functions *delay* or *slew*, these characteristics shall apply to the segment for which that calculation function is being called.

Without multiplication characteristic data, a propagation segment shall be considered to produce one edge transition at its output for each edge transition to which the segment is sensitive at its input. However, a segment for which such data are present shall be evaluated in a manner prescribed via these data.

The multiplication characteristics for a segment consist of three parameters. The description of these parameters is in ideal terms and does not include the effects of the segment's delay characteristics. Actual transition times for segment output edges shall be offset from their ideal times by the segment's delays and their slews shall be determined from the segment's propagation characteristics. The three parameters are as follows:

The first parameter, the multiplication factor, is a positive integer that specifies the number of edges at the segment output produced by each edge to which the segment is sensitive at its input. This parameter shall have a minimum value of 2.

The second parameter is the edge direction for the first signal transition at the output, which is produced by each edge at the input. This parameter is of type *DCM_EdgeTypes* and shall have one of the following values:

- *DCM_RisingEdge*
- *DCM_FallingEdge*
- *DCM_BothEdges*

If *DCM_RisingEdge* or *DCM_FallingEdge* is specified, the segment shall produce an edge of the opposite polarity for each consecutive output transition. The multiplication parameter shall have an even value and each pair of output edges shall form a cycle of the output waveform. The interval of time between one input edge and the next shall be equally divided among the cycles that occur between them, thus defining their period. This shall be done independently for each pair of input edges. The duty cycle of this waveform shall be defined by the third parameter, as described in item c).

If *DCM_BothEdges* is specified, the output shall be considered to be capable of producing edges of either polarity for each input transition. In this case, the output edges shall be equally spaced in the interval of time between one input edge and the next. This shall be done independently for each pair of input edges.

The third parameter is a positive floating point value that presents the duty cycle of the segment's output waveform if the second parameter does not have a value of *DCM_BothEdges*. This third parameter represents the fraction of each cycle during which the output maintains its initial value and shall be less than 1.0. If the second parameter has the value *DCM_BothEdges*, this third parameter shall be ignored.

The application shall apply these characteristics to a segment as defined as a unit via a call by the DPCM to *newPropagateSegment*. These characteristics shall not be applied separately to individual timing arcs that the application can generate in response to such a call. In particular, the application shall consider the segment characteristics to be those described in the delay matrix referenced by that call.

If the specified segment has multiplication characteristics, it shall not also have division nor signal generation characteristics.

10.23.16.3 appSetSignalGeneration

Table 370—appSetSignalGeneration

Function name	appSetSignalGeneration
Arguments	List of edge times, First input direction, Period
Result	None
Standard Structure fields	block, CellName, fromPoint, toPoint, pathData (timing-arc-specific), cellData (timing)
DCL syntax	<pre>EXTERNAL (appSetSignalGeneration) : passed(float[*]: inputEdgeTimes; int: initialInputEdge; float: period) result(int: ignore);</pre>
C syntax	<pre>typedef struct { DCM_INTEGER ignore; } T_Ignore; int appSetSignalGeneration (const DCM_STD_STRUCT *std_struct, T_Ignore *rtn, DCM_FLOAT_ARRAY *inputEdgeTimes, DCM_EdgeTypes initialInputEdge, DCM_FLOAT period);</pre>

This sends data describing input signal generation characteristics for the segment last defined during timing model elaboration or for which calculation is currently being performed.

During model elaboration, the library shall call this function immediately after calling *newPropagateSegment* to define a timing segment. The specified characteristics shall apply to the timing segment so defined.

During an application call to the library calculation functions *delay* or *slew*, these characteristics shall apply to the segment for which that calculation function is being called.

These data shall describe a signal waveform for the input of the segment relative to time 0. The application shall determine when time 0 occurs and shall do so in a manner that preserves the relative temporal relationships between all segments in a cell whenever possible.

The signal generation characteristics for a segment consist of three parameters. Transition times and slews for segment output edges shall be determined from the input waveform described via these parameters and from the segment's propagation characteristics.

The first parameter, the edge time list, is a list of times at which transitions occur at the input of the segment. These shall be non-negative floating point numbers and shall be specified in ascending order.

The second parameter is the edge direction for the first signal transition at the input. This parameter is of type *DCM_EdgeTypes* and shall have one of the following values:

- *DCM_RisingEdge*
- *DCM_FallingEdge*
- *DCM_BothEdges*

If *DCM_RisingEdge* or *DCM_FallingEdge* is specified, the segment input shall be considered to have an edge of the opposite polarity for each consecutive transition. If *DCM_BothEdges* is specified, the input shall be considered to have edges of either polarity for each time in the edge time list and any number of entries can be included in the latter.

The third parameter is a non-negative floating point value that presents the period of the segment's input waveform. If this parameter has a positive value, it shall be greater than the last time in the edge time list. If the second parameter is *DCM_RisingEdge* or *DCM_FallingEdge*, the edge time list shall have an even number of entries.

If the third parameter has a value of *0.0*, the input waveform shall be considered to have transitions at the times specified in the edge time list only. In this case, the latter can have any number of entries.

The application shall apply these characteristics to a segment as defined as a unit via a call by the DPCM to *newPropagateSegment*. These characteristics shall not be applied separately to individual timing arcs that the application can generate in response to such a call. In particular, the application shall consider the segment characteristics to be those described in the delay matrix referenced by that call.

If the specified segment has signal generation characteristics, it shall not also have multiplication or division characteristics.

10.23.17 Default pin slews and interface version calls

This subclause defines the default pin slews call and the interface version and sequencing call.

10.23.17.1 dpcmGetDefPinSlews

Table 371 provides information on *dpcmGetDefPinSlews*.

Table 371—*dpcmGetDefPinSlews*

Function name	<i>dpcmGetDefPinSlews</i>
Arguments	Pin pointer
Result	Default rise pin slew, Default fall pin slew
Standard Structure fields	block, CellName, cellData (timing), pathData (timing-pin-specific), calcMode
DCL syntax	<pre>EXPOSE(dpcmGetDefPinSlews): passed(pin: pinPointer) result(double: defRisePinSlew, defFallPinSlew);</pre>

C syntax	<pre>typedef struct { DCM_DOUBLE defRisePinSlew; DCM_DOUBLE defFallPinSlew; } T_defPinSlews; int dpcmGetDefPinSlews(const DCM_STD_STRUCT *std_struct, T_defPinSlews *rtn, DCM_PIN pinPointer);</pre>

This returns the default slews for the pin identified by the passed *pinPointer* argument of the *cell* in the *Standard Structure* back to the application.

10.23.17.2 appGetInterfaceVersion

Table 372 provides information on dpcmGetInterfaceVersion.

Table 372—appGetInterfaceVersion

Function name	appGetInterfaceVersion
Arguments	None
Result	Array of supported versions
Standard Structure fields	None
DCL syntax	<pre>EXTERNAL(appGetInterfaceVersion): result(string [*]: array_of_supported_versions);</pre>
C syntax	<pre>typedef struct { DCM_STRING_ARRAY *interfaceVersion; } T_Ver; int appGetInterfaceVersion (const DCM_STD_STRUCT *std_struct, T_Ver *rtn);</pre>

This returns an array of strings, identifying the versions of the IEEE 1481 interface that the application supports, back to the library.

The valid versions for IEEE 1481 *dpcmGetVersionInfo()* and *appGetInterfaceVersionInfo()* are shown in Table 373.

Table 373—Valid interface version strings

"IEEE 1481-1998"
"IEEE 1481-1999 V1.0"
"IEEE 1481-2003 V2.0"

An example of the call sequence is as follows:

The application calls *dpcmGetVersionInfo()*.

The DPCM calls *appGetInterfaceVersion()*.

The application returns the supported interface versions.

The DPCM selects one of the application supported versions.

The DPCM returns the selected version back to the initial *dpcmGetVersionInfo()* call.

10.23.18 API to access library required resources

The library can require the application set the generic resources it needs (e.g., the type of application where it is interfaced). This is accomplished with two resource-specific APIs. The new APIs allow the application to query the library for a valid list of resources and set the value of a resource within the library.

10.23.18.1 Expose APIs for resources

A *resource* is an arbitrarily named variable that can contain an arbitrary value. This value shall be maintained as a string. Each resource description structure contains information about the resource's type, the valid ranges, a description of each resource, and a description of each possible resource value. The application can use the resource description structure to set up a user query for the value of a resource.

The application can use *EXPOSE* APIs to obtain the resource name and description, the possible values for the resource, a description for each value, the default value, and the type for the value.

10.23.18.1.1 dpcmSetResource

Table 374 provides information on dpcmSetResource.

Table 374—dpcmSetResource

Function name	dpcmSetResource
Arguments	Resource name, Resource value
Result	Old resource value
Standard Structure fields	None
DCL syntax	<pre>EXPOSE(dpcmSetResource): passed(string: resourceName, resourceValue) result(string: oldResourceValue);</pre>
C syntax	<pre>typedef struct { STRING oldResourceValue; } T_OldResource; int dpcmSetResource(const DCM_STD_STRUCT *std_struct, T_OldResource *rtn, DCM_STRING resourceName, DCM_STRING resourceValue);</pre>

This allows the application to set the value of a resource. The *resourceValue* shall follow the resource description for the named resource. The *resourceValue* responses are described by the *resourceTypes* (see Table 376).

10.23.18.1.2 dpcmGetAllResources

Table 375 provides information on dpcmGetAllResources.

Table 375—dpcmGetAllResources

Function name	dpcmGetAllResources
Arguments	None
Result	Resource data
Standard Structure fields	None

DCL syntax	<pre> typedef(resource_data): result(string: resourceName, resourceDescription; int: resourceType; string[*]: resourceValues, resourceValueDescriptions; string: defaultResourceValue); EXPOSE(dpcmGetAllResources): result(resource_data[*]: resourceData); </pre>
C syntax	<pre> typedef enum DCM_ResourceTypes { DCM_ResourceType_arbitrary_number, DCM_ResourceType_arbitrary_string, DCM_ResourceType_enumerated_number, DCM_ResourceType_enumerated_string, DCM_ResourceType_number_range, DCM_ResourceType_character_range } DCM_ResourceTypes; typedef struct { DCM_STRING resourceName; DCM_STRING resourceDescription; DCM_ResourceTypes resourceType; DCM_STRING_ARRAY *resourceValues; DCM_STRING_ARRAY *resourceValueDescriptions; DCM_STRING defaultResourceValue; } DCM_ResourceData; typedef DCM_ResourceData *DCM_ResourceData_ARRAY; typedef struct { DCM_ResourceData_ARRAY *resourceData; } T_AllResources; int dpcmGetAllResources(const DCM_STD_STRUCT *std_struct, T_AllResources **rtn); </pre>

This allows the application to query the library for all resources. The array of structures returned by this call contains the set of resource description structures. Each resource description (*DCM_ResourceData*) contains the resource's name, a descriptive string describing the resource, an enumeration identifying the type of resource, and a matching pair of arrays: one containing the resource values and the other a description of each resource value.

10.23.19 Resource types

The resource types are defined by the enumeration name *DCM_ResourceTypes*, as shown in Table 376. They are used to indicate how the application is to treat each type of resource.

Table 376—DCM_ResourceTypes

Enumerator (enumeration)	Description
DCM_ResourceType_arbitrary_number (0)	In response to the library query <i>appGetResource</i> of a named resource of this type, the application shall return a string representation of any arbitrary number. The array of <i>resourceValues</i> and <i>resourceValue Descriptions</i> shall be empty.
DCM_ResourceType_arbitrary_string (1)	In response to the library query <i>appGetResource</i> of a named resource of this type, the application shall return a string representation of any arbitrary string. The array of <i>resourceValues</i> and <i>resourceValue Descriptions</i> shall be empty.
DCM_ResourceType_enumerated_number (2)	In response to the library query <i>appGetResource</i> of a named resource of this type, the application shall return a string representing one of the elements contained in the <i>resourceValue</i> array. The <i>resourceValue</i> array shall contain an element for each possible response by the application. In this type of resource, the <i>resourceValue Description</i> array shall contain a matching descriptive string for each possible value.
DCM_ResourceType_enumerated_string (3)	In response to the library query <i>appGetResource</i> of a named resource of this type, the application shall return a string contained as one of the elements contained in the <i>resourceValue</i> array. The <i>resourceValue</i> array shall contain an element for each possible response by the application. In this type of resource, the <i>resourceValueDescription</i> array shall contain a matching descriptive string for each possible value.
DCM_ResourceType_number_range (4)	In response to the library query <i>appGetResource</i> of a named resource of this type, the application shall return a string representing a number that is within the number range (inclusive) specified by one of the elements contained in the <i>resourceValue</i> array. The <i>resourceValue</i> array shall contain an element for each possible response by the application. In this type of resource, the <i>resourceValueDescription</i> array shall contain a matching descriptive string for each possible value. The number ranges contained in the <i>resourceValue</i> array shall consist of string representing two numbers separated by a hyphen (-).
DCM_ResourceType_character_range (5)	In response to the library query <i>appGetResource</i> of a named resource of this type, the application shall return a string representing a single character that is within the character range (inclusive) specified by one of the elements contained in the <i>resourceValue</i> array. The <i>resourceValue</i> array shall contain an element for each possible response by the application. In this type of resource, the <i>resourceValueDescription</i> array shall contain a matching descriptive string for each possible value. The character ranges contained in the <i>resourceValue</i> array shall consist of a string representing two single characters separated by a hyphen (-).

10.23.20 Library extensions for phase locked loop processing

PLLs are specialized circuits that have several purposes. Typical usage of PLL includes chip-to-chip skew control, frequency multiplication and frequency division. There also needs to be a method for describing the characteristics of a PLL to enable static timing analysis.

To do this, special timing arcs for phase locked loops are introduced. Although the PLL typically has no physical propagation path from the input to the output, there is a time relationship between these two signals. This time relationship can be represented as an arc from the input to the output. The delay placed on the arc traversing from input to output is a function of the feedback delay (the delay from the output to the feedback pin). The arc representing the input to output timing relationship can be easily understood by static timing tools.

The input to output delay of this arc causes the static timer to adjust the arrival times so they correspond to the actual arrival times that would have occurred on the implemented circuit. To make this possible, the DPCM calls *appGetExternalDelayByName* or *appGetExternalDelayByPin* to determine the delay from the PLL's output to the feedback input.

In addition to simply knowing the delays from the oscillator output to the feedback, some other properties of the feedback need to be known. These include the general condition of the feedback path, such as whether there is signal reconvergence or a single edge generates more than one edge at the reference. In addition, some PLL feed back configurations can be used to multiply or divide the input frequency.

These conditions are returned by the application when the library makes calls requesting such information (see 10.23.21).

10.23.21 API definitions for external conditions

The following set of calls in Table 377 request information regarding aspects of the external environment that affect delay calculation for a particular cell instance.

10.23.21.1 appGetExternalDelayByPin

Table 377—appGetExternalDelayByPin

Function name	appGetExternalDelayByPin
Arguments	Output pin pointer, Input pin name, Edge direction at output (enum), Edge direction at input (enum)
Result	Early external delay, Late external delay, Multi-arc, Multi-edge path
Standard Structure fields	block, CellName, calcMode, sinkEdge
DCL syntax	<pre>EXTERNAL (appGetExternalDelayByPin) : passed (pin: outputPinPointer; string: inputPinName; int: edgeDirectionAtOutput, edgeDirectionAtInput) result (double: earlyExternalDelay, lateExternalDelay; int: multiArcMultiEdgePath);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE earlyExternalDelay, lateExternalDelay; DCM_INTEGER multiArcMultiEdgePath; } T_ExternalDelay; int appGetExternalDelayByPin (const DCM_STD_STRUCT *std_struct, T_ExternalDelay *rtn, DCM_PIN outputPinPointer, DCM_STRING inputPinName, DCM_EdgeTypes edgeDirectionAtOutput, DCM_EdgeTypes edgeDirectionAtInput);</pre>

This returns the total delay and path status, from an output pin to an input pin, back to the library, as follows:

- a) Passed parameter definitions include the following:
 - 1) *outputPinPointer*: The point where the delay is to start, typically the output pin of the cell.
 - 2) *inputPinName*: The name of the pin on the same cell where the delay is to be measured, typically the reference pin of the cell.
 - 3) *edgeDirectionAtOutput*: The edge at the output used for delay calculation purposes.
 - 4) *edgeDirectionAtInput*: The output delay starting at the output edge is measured to this edge at the input pin.

NOTE—If the application cannot compute the delay using the edges supplied, it shall return an error to the library.

- b) Returned value definitions include the following:
 - 1) *earlyExternalDelay*: The time, in library units, of the shortest time period the output pin edge takes to propagate back to the input pin.
 - 2) *lateExternalDelay*: the time, in library units, of the longest time period the output pin edge takes to propagate back to the input pin.
 - 3) *multiArcMultiEdgePath*: an OR ed bit field containing the output to input test bits and the multiplication or division factor. The multiplication or division factor occupies the high order two bytes and the test bits occupy the low order two bytes, as shown below.

Table 378 provides information on multiArcMultiEdgePath.

Table 378—Description of multiArcMultiEdgePath (XXXX are do not care bits)

multiArc MultiEdgePath value	#define name	Description
0xXXXX0000	DCM_EXTERNAL_SIMPLE_PATH	The external path includes none of the conditions described for the other values below.
0xXXXX0001	DCM_EXTERNAL_RECONVERGENCE	The cone of logic starting at the output fans out to multiple paths, at least two of which reconverge and then lead to the input.
0xXXXX0002	DCM_EXTERNAL_DIV_MULT	The external path includes frequency division or multiplication.
0xXXXX0004	DCM_EXTERNAL_INJECTED_SIGNAL	The external path is affected by one or more logic signals other than that from the output.
0xXXXX0008	DCM_EXTERNAL_NO_PATH	No path.
0xXXXX0010	DCM_EXTERNAL_NO_EDGE	No edge.
0xIJKL0020	DCM_EXTERNAL_FREQUENCY_DIVISION	An external frequency division value of 0xIJKL.
0xIJKL0040	DCM_EXTERNAL_FREQUENCY_MULTIPLICATION	An external frequency multiplication value of 0xIJKL.

The library shall call *appSetSignalDivision* (see 10.23.16.1) or *appSetSignalMultiplication* (see 10.23.16.2) if the library has determined a change in signal rate division or multiplication characteristics, respectively, has occurred along the path originally supplied by the application in the interface function call that lead the library to call *appGetExternalDelayByPin*.

10.23.21.2 appGetExternalDelayByName

Table 379 provides information on *appGetExternalDelayByName*.

Table 379—*appGetExternalDelayByName*

Function name	<i>appGetExternalDelayByName</i>
Arguments	Output pin name, Input pin name, Edge direction at output (enum), Edge direction at input (enum)
Result	Early external delay, Late external delay, Multi-arc, Multi-edge path
Standard Structure fields	block, CellName, calcMode, sinkEdge
DCL syntax	<pre>EXTERNAL (appGetExternalDelayByName) : passed (string: outputPinName; string: inputPinName; int: edgeDirectionAtOutput, edgeDirectionAtInput) result (double: earlyExternalDelay, lateExternalDelay; int: multiArcMultiEdgePath);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE earlyExternalDelay, lateExternalDelay; DCM_INTEGER multiArcMultiEdgePath; } T_ExternalDelay; int appGetExternalDelayByName (const DCM_STD_STRUCT *std_struct, T_ExternalDelay *ret, DCM_STRING outputPinName, DCM_STRING inputPinName, DCM_EdgeTypes edgeDirectionAtOutput, DCM_EdgeTypes edgeDirectionAtInput);</pre>

This returns the total delay and path status, from an output pin to an input pin, back to the library, as follows:

- a) Passed parameter definitions include the following:
 - 1) *outputPinName*: The name of the pin where the delay is to start, typically the output pin of the cell.
 - 2) *inputPinName*: The name of the pin on the same cell where the delay is to be measured, typically the reference pin of the cell.
 - 3) *edgeDirectionAtOutput*: The edge at the output used for delay calculation purposes.
 - 4) *edgeDirectionAtInput*: The output delay starting at the output edge is measured to this edge at the input pin.

NOTE—If the application cannot compute the delay using the edges supplied, it shall return an error to the library.

b) Returned value definitions include the following:

- 1) *earlyExternalDelay*: The time, in library units, of the shortest time period the output pin edge takes to propagate back to the input pin.
- 2) *lateExternalDelay*: The time, in library units, of the longest time period the output pin edge takes to propagate back to the input pin.
- 3) *multiArcMultiEdgePath*: An OR ed bit field containing the output to input test bits and the multiplication or division factor. The multiplication or division factor occupies the high order two bytes and the test bits occupy the low order two bytes, as shown in Table 378.

The library shall call *appSetSignalDivision* (see 10.23.16.1) or *appSetSignalMultiplication* (see 10.23.16.2) if the library has determined a change in signal rate division or multiplication characteristics, respectively, has occurred along the path originally supplied by the application in the interface function call that lead the library to call *appGetExternalDelayByName*.

10.23.21.3 appGetLogicLevelByName

Table 380 provides information on *appGetLogicLevelByName*.

Table 380—appGetLogicLevelByName

Function name	appGetLogicLevelByName
Arguments	Pin name
Result	Logic level enumeration
Standard Structure fields	block, CellName
DCL syntax	EXTERNAL (appGetLogicLevelByName) : passed(string: pinName) result(int: level);
C syntax	<pre>typedef enum DCM_LogicLevel { DCM_LogicUnknown, DCM_LogicZero, DCM_LogicOne } DCM_LogicLevel; typedef struct { DCM_LogicLevel level; } T_logicLevel; int appGetLogicLevelByName (const DCM_STD_STRUCT *std_struct, T_logicLevel *rtn, DCM_STRING pinName);</pre>

Some cells in a library, such as PLLs, typically have control lines with fixed, yet programmable, voltage levels. The library needs to query the state of such pins from the application. *appGetLogicLevelByName* returns the logic level on the pin specified by the pin name back to the library. The logic level can have three enumerated possible states: logical one, logical zero, or unknown, as shown in Table 381.

Table 381—DCM_LogicLevel

Enumerator	Enumeration	Description
DCM_LogicUnknown	–1	Logic level unknown
DCM_LogicZero	0	Logic zero
DCM_LogicOne	1	Logic one

10.23.21.4 appGetLogicLevelByPin

Table 382 provides information on appGetLogicLevelByPin.

Table 382—appGetLogicLevelByPin

Function name	appGetLogicLevelByPin
Arguments	Pin pointer
Result	Logic level enumeration
Standard Structure fields	block, CellName
DCL syntax	<pre>EXTERNAL (appGetLogicLevelByPin) : passed(pin: pinPointer) result(int: level);</pre>
C syntax	<pre>typedef enum DCM_LogicLevel { DCM_LogicUnknown, DCM_LogicZero, DCM_LogicOne } DCM_LogicLevel; typedef struct { DCM_LogicLevel level; } T_logicLevel; int appGetLogicLevelByPin (const DCM_STD_STRUCT *std_struct, T_logicLevel *rtn, DCM_PIN pinPointer);</pre>

This returns the logic level on the pin back to the library. The enumeration returned as the logic level can have three possible states: logical one, logical zero, or unknown, as shown in Table 381.

10.23.22 Extensions for listing pins

Applications need a method for translating indexed netlist pins with the named pins of a cell. Certain netlist description languages allow pin reference by order rather than by name. The pins of a cell in the library are represented by names. To map between pin order and pin name, the application needs to query a cell for the indices of its pins. Each pin index shall be represented as an integer.

10.23.22.1 dpcmGetPinIndexArrays

Table 383 provides information on dpcmGetPinIndexArrays.

Table 383—dpcmGetPinIndexArrays

Function name	dpcmGetPinIndexArrays
Arguments	None
Result	Input pin indices, Output pin indices, Bidirectional pin indices
Standard Structure fields	CellName
DCL syntax	<pre>EXPOSE (dpcmGetPinIndexArrays) : result(int[*]: inputPinIndices, outputPinIndices, bidiPinIndices);</pre>

C syntax	<pre>typedef struct { DCM_INTEGER_ARRAY *inputPinIndices, *outputPinIndices, *bidiPinIndices; } T_PinIndices; int dpcmGetPinIndexArrays(const DCM_STD_STRUCT *std_struct, T_PinIndices *rtn);</pre>

This returns the indices of the input, output, and bidirectional pins of the cell specified in the *Standard Structure* back to the application. The order of the indices shall be the same as the order of the pins in the corresponding pin name arrays returned by *dpcmGetCellIOlists*.

10.23.22.2 dpcmGetSupplyPins

Table 384 provides information on *dpcmGetSupplyPins*.

Table 384—dpcmGetSupplyPins

Function name	dpcmGetSupplyPins
Arguments	None
Result	Array of power pins
Standard Structure fields	CellName
DCL syntax	<pre>typedef(powerPinInfo): result(string: pinName; int: railIndex); EXPOSE(dpcmGetSupplyPins): result(powerPinInfo[*]: powerPins);</pre>
C syntax	<pre>typedef struct { DCM_STRING pinName; DCM_INTEGER railIndex; } DCM_PowerInfo; typedef DCM_PowerInfo *DCM_PowerInfo_ARRAY; typedef struct { DCM_PowerInfo_ARRAY *powerInfoArray; } T_GetSupplyPins; int dpcmGetSupplyPins(const DCM_STD_STRUCT *std_struct, T_GetSupplyPins *rtn);</pre>

Certain classes of applications need access to the supply pins of a cell. *dpcmGetSupplyPins* returns the power and ground pins for a cell back to the application; it returns the individual pins as an array of structures, each containing the name of the power pin and the voltage rail with which it is associated.

DCM PowerInfo_ARRAY is a *DCM_ARRAY* of pointers to *DCM PowerInfo*. Both the array and the structures whose addresses it contains are allocated by the DPCM. The application shall neither allocate nor free the array or these structures. If the application needs to retain the array itself or any of these structures, it shall call *dcm_lock_DCM_ARRAY* or *dcm_lock_DCM_STRUCT* (for each structure to be retained), respectively.

10.23.23 Memory BIST mapping

A memory BIST tool requires information relating the logical address and data to a physical row, column, and bank in order to perform test functions, such as writing bit patterns into the memory and reading

expected bit patterns from the memory. The APIs in this subclause can be used to provide the BIST tool with this logical/physical mapping.

Because the tool needs to know whether the physical data in a specific location are inverted with respect to the corresponding logical data, the enumerations in Table 385 specify whether there is inversion between the physical/logical mapping.

Table 385—DCM_BistInversion

Enumerator	Enumeration	Description
DCM_BistInversion_False	0	No inversion
DCM_BistInversion_True	1	Inverted data

10.23.23.1 dpcmGetPhysicalBISTMap

Table 386 provides information on dpcmGetPhysicalBISTMap.

Table 386—dpcmGetPhysicalBISTMap

Function name	dpcmGetPhysicalBISTMap
Arguments	Logical address, Data word bit index
Result	Physical row, Physical column, Physical bank, Inversion
Standard Structure fields	CellName, block, cellData, pathData(pin-specific)
DCL syntax	<pre>EXPOSE(dpcmGetPhysicalBISTMap): passed(int: logicalAddress, dataIndex) result(int: physicalRow, physicalColumn, physicalBank, dataInversion);</pre>
C syntax	<pre>typedef enum DCM_BistInversion { DCM_BistInversion_False, DCM_BistInversion_True } DCM_BistInversion; typedef struct DCM_INTEGER physicalRow; DCM_INTEGER physicalColumn; DCM_INTEGER physicalBank; DCM_BistInversion dataInversion; } T_GetPhysicalBISTMap; int dpcmGetPhysicalBISTMap(const DCM_STD_STRUCT *std, T_GetPhysicalBISTMap *rtn, DCM_INTEGER logicalAddress, DCM_INTEGER dataIndex);</pre>

Given a logical address and bit index within the data word, this returns the corresponding physical row, column, and bank where the data word maps, as well as indicating whether the physical data are inverted with respect to the logical data value, back to the application.

10.23.23.2 dpcmGetLogicalBISTMap

Table 387 provides information on dpcmGetLogicalBistMap.

Table 387—dpcmGetLogicalBISTMap

Function name	dpcmGetLogicalBISTMap
Arguments	Physical row, Physical column, Physical bank
Result	Logical address, Data word bit index, Inversion
Standard Structure fields	CellName, block, cellData, pathData (pin-specific)
DCL syntax	<pre>EXPOSE(dpcmGetLogicalBISTMap): passed(int: physicalRow, physicalColumn, physicalBank) result(int: logicalAddress, dataIndex, dataInversion);</pre>
C syntax	<pre>typedef enum DCM_BistInversion { DCM_BistInversion_False, DCM_BistInversion_True } DCM_BistInversion; typedef struct { DCM_INTEGER logicalAddress; DCM_INTEGER dataIndex; DCM_BistInversion dataInversion; } T_GetLogicalBISTMap; int dpcmGetLogicalBISTMap(const DCM_STD_STRUCT *std, T_GetLogicalBISTMap *rtn, DCM_INTEGER physicalRow, DCM_INTEGER physicalColumn, DCM_INTEGER physicalBank);</pre>

Given a physical row, column, and bank, this returns the corresponding logical address and bit index where the data word maps, as well as indicating whether the logical data are inverted with respect to the physical data value, back to the application.

10.23.24 dpcmGetCellTestProcedure

Table 388 provides information on dpcmGetCellTestProcedure.

Table 388—dpcmGetCellTestProcedure

Function name	dpcmGetCellTestProcedure
Arguments	None
Result	Array of path data blocks
Standard Structure fields	CellName, block, cellData
DCL syntax	<pre>EXPOSE(dpcmGetCellTestProcedure): result(PATH_DATA [*]: testProcedurePathDataArray);</pre>
C syntax	<pre>typedef struct { DCM_PathDataBlock *testProcedurePathDataArray } T_GetCellTestProcedure; int dpcmGetCellTestProcedure (const DCM_STD_STRUCT *std, T_GetCellTestProcedure *rtn);</pre>

This obtains the test procedure defined for a cell and returns an array of path data blocks, the functionality of which, as specified by the associated function graphs, when executed in the returned order, performs the desired test sequence.

10.24 Interconnect delay calculation intraface

This subclause contains the specification of an intraface through which interconnect calculation code in a library can interact with the cell-calculation part of that library. The term intraface is employed here to identify a collection of functions and data types used for communication between modules within a library as opposed to between the library and an application. This Interconnect Delay Calculation (IDC) intraface supports both delay and slew computation for interconnect arcs and the inclusion of interconnect-network loading during cell-arc calculation.

All calculations in a library are, from the perspective of the application, performed by a cell model within the library. This is true even for interconnect calculations, for which the application provides data to the library identifying the driving cell and pin. For the purpose of delay and slew calculation, a technology family in a library can be divided into two modules, a Cell Calculation Module (CCM) and an Interconnect Calculation Module (ICM). The ICM provides a computational resource to the CCM to be used when interconnect calculation is requested by the application. The CCM exposes functions to the ICM needed by the latter during its own computations, for example during calculation of the effects of interconnect loading on cell delay and slew.

The units used for all quantities involved during these IDC computations shall be those of the CCM.

An ICM shall be written such that it can support use by multiple technology families in the library. In particular, the ICM shall not assume that the technology of the CCM that calls one of its functions is the same as for any other call to that or any other ICM function. It shall also be possible for different technology families to use different ICM implementations.

To facilitate this, each ICM shall provide a table of pointers to its functions; a CCM shall call ICM functions via these pointers. So that use of these ICM function pointers can be made transparent to the bulk of the DCL code in a CCM, DCL macros through which the ICM functions can be called shall be defined. To prevent conflicts between the names of these macro and those of the functions, the DCL function declarations shall use names ending in “_ft” (standing for “function type”), and each corresponding macro shall have a similar name without the “_ft” ending. As is the case for all DCL functions, each ICM function name is also the DCL type name for the data structure returned by that function.

It is, of course, possible to construct a library that includes interconnect calculation without using this approach. Such a library would be compliant to this standard overall. The intent behind this subclause is not to mandate that all compliant libraries be written to use this intraface. The intent is, however, to provide a normative description of the IDC intraface such that a library written to comply with it shall adhere to the specifications enumerated in the remainder of this subclause.

10.24.1 Control and data flows

Because the application interacts primarily with the CCM, the ICM shall return all of its results back to the CCM, which shall in turn be responsible for returning these data to the application. This arrangement facilitates memory management of aggregate data structures such as arrays and structures via utilities provided by the run-time environment. Such management is automatically enabled during return of data to the application in this manner.

In order to complete its calculations, the ICM might require additional data beyond what is initially provided by the application. Rather than making requests for these data via the CCM, the ICM shall call application functions directly. Although these functions shall provide the requested data, the ICM shall pass no data to the application except that needed by the latter to fulfill such requests. During this process, the ICM shall not provide data that are to be stored in or managed by the application. The overall control and data flows among the application, CCM, and ICM are illustrated in Figure 15.

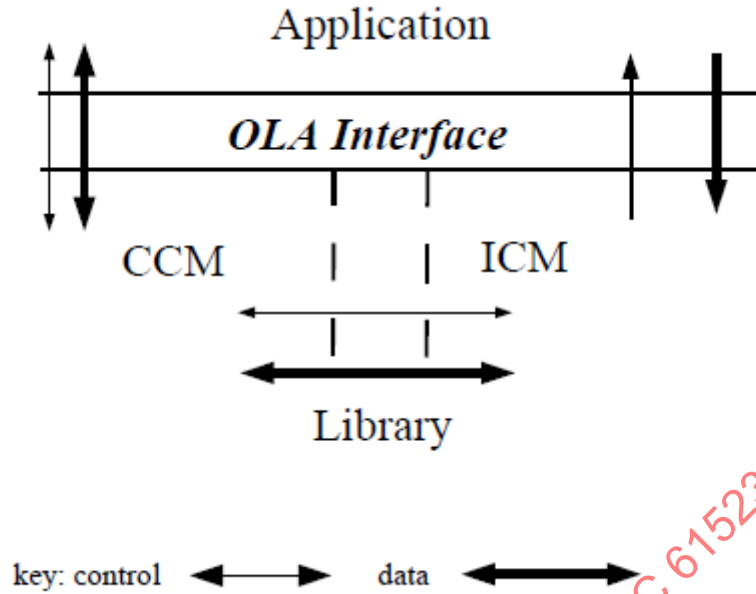


Figure 15—Application, CCM, and ICM control and data flows

The following subclauses contain specifications of the functions that constitute the IDC intraface. In general, early and late versions of all computational results are calculated, using different interconnect networks or models when available.

10.24.2 Model generation functions

So that the ICM can perform interconnect calculations, the CCM shall cede generation of the load and interconnect models associated with an interconnect network to the ICM. When the application calls library functions implemented in the CCM that return these models, these CCM functions shall call the equivalent ICM functions to generate the models requested.

These ICM functions shall generate load and interconnect models that are compatible with the specific ICM implementation to which the functions belong. The functions `dpcmBuildLoadModels` and `dpcmBuildInterconnectModels` (see 10.21.9) shall call the corresponding ICM functions to generate such models when requested by the application. When these models are needed by the ICM for computational purposes and they are not passed to it as function arguments, it shall obtain them from the application by calling `appGetLoadModels` (see 10.21.9.4) or `appGetInterconnectModels` (see 10.21.9.3).

10.24.2.1 `icmBuildLoadModels`

Table 389 provides information on `icmBuildLoadModels`.

Table 389—`icmBuildLoadModels`

Function name	<code>icmBuildLoadModels</code>
Arguments	Parasitic networks, ICM network models, Driving pin node number
Result	ICM load models, ICM network models
Standard Structure fields	<code>CellName</code> , <code>cellData</code> (timing), <code>pathData</code> (timing-pin-specific), <code>toPoint</code>

DCL syntax	<pre>forward calc(icmBuildLoadModels_ft): passed(parasiticSubnet: minParasitics, maxParasitics; void: minNetModel, maxNetModel; int: minDrivingPinNodeNumber, maxDrivingPinNodeNumber) result(void: minLoadModel, maxLoadModel, newMinNetModel, newMaxNetModel);</pre>
C syntax	<pre>typedef struct { const DCM_STRUCT *minLoadModel, *maxLoadModel; const DCM_STRUCT *newMinNetModel, *newMaxNetModel; } T_BuildLoadModels; int icmBuildLoadModels(const DCM_STD_STRUCT *std_struct, T_BuildLoadModels *rtn, const DCM_ParasiticSubnet *minParasitics, const DCM_ParasiticSubnet *maxParasitics, const DCM_STRUCT *minNetModel, const DCM_STRUCT *maxNetModel, DCM_INTEGER minDrivingPinNodeNumber, DCM_INTEGER maxDrivingPinNodeNumber);</pre>

This function creates load models for a driver of an interconnect network that are compatible with the specific ICM implementation to which the function belongs. Its semantics shall be identical to those of dpcmBuildLoadModels (see 10.21.9.1).

10.24.2.2 icmBuildInterconnectModels

Table 390 provides information on icmBuildInterconnectModels.

Table 390—icmBuildInterconnectModels

Function name	icmBuildInterconnectModels
Arguments	Parasitic networks, ICM network models, Driving pin node number, Sink pin node number
Result	ICM interconnect models, ICM network models
Standard Structure fields	CellName, cellData (timing), pathData (timing-pin specific), fromPoint, toPoint
DCL syntax	<pre>forward calc(icmBuildInterconnectModels_ft): passed(parasiticSubnet: minParasitics, maxParasitics; void: minNetModel, maxNetModel; int: minDrivingPinNodeNumber, minSinkPinNodeNumber, maxDrivingPinNodeNumber, maxSinkPinNodeNumber) result(void: minInterconnectModel, maxInterconnectModel, newMinNetModel, newMaxNetModel);</pre>

C syntax	<pre> typedef struct { const DCM_STRUCT *minInterconnectModel, *maxInterconnectModel; const DCM_STRUCT *newMinNetModel, *newMaxNetModel; } T_BuildInterconnectModels; int icmBuildInterconnectModels(const DCM_STD_STRUCT *std_struct, T_BuildInterconnectModels *rtn, const DCM_ParasiticSubnet *minParasitics, const DCM_ParasiticSubnet *maxParasitics, const DCM_STRUCT *minNetModel, const DCM_STRUCT *maxNetModel, DCM_INTEGER minDrivingPinNodeNumber, DCM_INTEGER minSinkPinNodeNumber, DCM_INTEGER maxDrivingPinNodeNumber, DCM_INTEGER maxSinkPinNodeNumber); </pre>

This function creates interconnect models for a path through an interconnect network that are compatible with the specific ICM implementation to which the function belongs. Its semantics shall be identical to those of dpcmBuildInterconnectModels (see 10.21.9.1).

10.24.3 Calculation functions

These functions calculate delays, output slews, and related data both for paths through interconnect networks and for paths through cells driving such networks. The additional data produced can include (optionally) output XWF structures (see 10.23.8) and interaction-window arrays (see 10.21.15). If such data are not computed, pointers having zero (0) values for the associated arrays or structures shall be returned.

Any XWF structures used during these calculations shall be compatible with the ICM implementation to which the functions belong. The contents of these structures shall be private to the ICM. Such structures shall be generated by the ICM via the functions icmCalcCellDelaySlew (see 10.24.4.1) or icmCalcXWF (see 10.24.4.7). In order to generate compatible structures, the function dpcmCalcXWF (see 10.23.8.3.3) shall call icmCalcXWF as needed.

Pointers to XWF structures generated by the ICM shall be passed to the application by the CCM using the function appSetXWF (see 10.23.8.3.1). These structures shall be retrieved for subsequent use by the ICM via the function appGetXWF (see 10.23.8.3.2).

Other XWF structures may not be compatible with the ICM and therefore shall not be used.

10.24.3.1 icmCalcInterconnectDelaySlew

Table 391 provides information on icmCalcInterconnectDelaySlew.

Table 391—icmCalcInterconnectDelaySlew

Function name	icmCalcInterconnectDelaySlew
Arguments	ICM interconnect, models
Result	Early and late delays, Early and late output slews, Early and late XWF data structures, Early and late interaction-window arrays
Standard Structure fields	CellName, calcMode, processVariation, block, fromPoint, toPoint, slew.early, slew.late, sourceEdge, sinkEdge, sourceMode, sinkMode, cellData (timing), pathData (timing-pin specific)

DCL syntax	<pre> typedef(timeRange): result(double var: earlyTime, lateTime); forward calc(icmCalcInterconnectDelaySlew_ft): passed(void: minInterconnectModel, maxInterconnectModel) result(float: earlyDelay, lateDelay, earlySlew, lateSlew; void: earlyXWF, lateXWF; timeRange[*]: earlyInteractWindows, lateInteractWindows); </pre>
C syntax	<pre> typedef struct { DCM_DOUBLE earlyTime, lateTime; } DCM_TimeRange; typedef DCM_TimeRange *DCM_TimeRange_ARRAY; typedef struct { DCM_FLOAT earlyDelay, lateDelay; DCM_FLOAT earlySlew, lateSlew; const DCM_STRUCT *earlyXWF, *lateXWF; const DCM_TimeRange_ARRAY *earlyInteractWindows; const DCM_TimeRange_ARRAY *lateInteractWindows; } T_InterconnectDelaySlew; int icmCalcInterconnectDelaySlew(const DCM_STD_STRUCT *std_struct, T_InterconnectDelaySlew *rtn, const DCM_STRUCT *minInterconnectModel, const DCM_STRUCT *maxInterconnectModel); </pre>

This function calculates delays, output slews, and related data for a path through an interconnect network. Interconnect models for this path generated by `icmBuildInterconnectModels` (see 10.24.2.2) shall be used during these computations. If load models for the path's source pin are also used, they shall be generated by `icmBuildLoadModels` (see 10.24.2.1).

10.24.4 Cell calculation functions

Cell delays and slews are often characterized using lumped load capacitances, and these delay and slew values are usually nonlinear functions of that capacitance. The CCM shall perform the actual delay and slew calculations for a cell path given an effective load capacitance. The ICM can also calculate delay and slew for the path using data obtained from the CCM and the load model for the interconnect network. The ICM shall be responsible for selecting a load-capacitance value such that these two sets of delay and slew values match.

Matching of network loading to these (nonlinear) cell characteristic can be an iterative process. To accomplish this, the ICM function `icmCalcCellDelaySlew` can make multiple calls to the CCM function `ccmCalcDelaySlew`. Early and late calculations can take different numbers of iterations and result in different sets of values. Consequently, early and late computations shall be made via separate calls to `ccmCalcDelaySlew` by the ICM.

10.24.4.1 `icmCalcCellDelaySlew`

Table 392 provides information on `icmCalcCellDelaySlew`.

Table 392—icmCalcCellDelaySlew

Function name	icmCalcCellDelaySlew
Arguments	ICM load models
Result	Early and late delays, Early and late output slews, Early and late XWF data structures, Early and late Ceffectives, Early and late interaction-window arrays
Standard Structure fields	CellName, calcMode, processVariation, block, fromPoint, toPoint, slew.early, slew.late, sourceEdge, sinkEdge, sourceMode, sinkMode, cellData (timing), pathData (timing-arc specific)
DCL syntax	<pre>typedef(timeRange): result(double var: earlyTime, lateTime); forward calc(icmCalcCellDelaySlew_ft): passed(void: minLoadModel, maxLoadModel) result(float: earlyDelay, lateDelay, earlySlew, lateSlew; void: earlyXWF, lateXWF; double earlyCeffective, lateCeffective; timeRange[*]: earlyInteractWindows, lateInteractWindows);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE earlyTime, lateTime; } DCM_TimeRange; typedef DCM_TimeRange *DCM_TimeRange_ARRAY; typedef struct { DCM_FLOAT earlyDelay, lateDelay; DCM_FLOAT earlySlew, lateSlew; const DCM_STRUCT *earlyXWF, *lateXWF; DCM_DOUBLE earlyCeffective, lateCeffective; const DCM_TimeRange_ARRAY *earlyInteractWindows; const DCM_TimeRange_ARRAY *lateInteractWindows; } T_CellDelaySlew; int icmCalcCellDelaySlew(const DCM_STD_STRUCT *std_struct, T_CellDelaySlew *rtn, const DCM_STRUCT *minLoadModel, const DCM_STRUCT *maxLoadModel);</pre>

This function calculates delays, output slews, and related data for a cell path, taking into account the loading on the sink pin of that path presented by the attached interconnect network. Load models for the sink pin generated by icmBuildLoadModels (see 10.24.2.1) shall be used during these computations.

In addition to those optional items listed in , the related data calculated shall include effective load capacitances seen at the sink pin. The XWF structures can contain data used to represent the sink pin during subsequent delay and slew calculations for paths through the interconnect network driven by that pin (see 10.24.3.1).

10.24.4.2 ccmCalcDelaySlew

Table 393 provides information on ccmCalcDelaySlew.

Table 393—ccmCalcDelaySlew

Function name	ccmCalcDelaySlew
Arguments	propagation mode, load capacitance
Result	Delay, Output slew, Gradients vs. capacitance
Standard Structure fields	CellName, calcMode, processVariation, block, fromPoint, toPoint, slew.early, slew.late, sourceEdge, sinkEdge, cellData (timing), pathData (timing-arc specific)
DCL syntax	<pre>EXPOSE(ccmCalcDelaySlew): passed(int: propagationMode; double: loadCapacitance) result(double: delayValue, delayCapGradient, slewValue, slewCapGradient);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE delayValue, delayCapGradient; DCM_DOUBLE slewValue, slewCapGradient; } T_DelaySlew; int ccmCalcDelaySlew(const DCM_STD_STRUCT *std_struct, T_DelaySlew *rtn, DCM_PropagationTypes propagationMode, DCM_DOUBLE loadCapacitance);</pre>

This function calculates the delay and output slew for a cell path given an effective load capacitance. It serves to make available to the ICM the delay and output-slew characteristics of the path as functions of load capacitance. The ICM shall match these characteristics with the load presented by the interconnect network driven by the path's sink pin. In order to facilitate this process, ccmCalcDelaySlew shall also return the gradients of these functions at the capacitance value provided.

10.24.4.3 ccmEarlyLateIdentical

Table 394 provides information on ccmEarlyLateIdentical.

Table 394—ccmEarlyLateIdentical

Function name	ccmEarlyLateIdentical
Arguments	None
Result	Boolean value indicating whether identical delays and slews shall be produced for early and late propagation modes
Standard Structure fields	CellName, calcMode, processVariation, block, fromPoint, toPoint, slew.early, slew.late, sourceEdge, sinkEdge, cellData (timing), pathData (timing-arc specific)
DCL syntax	<pre>EXPOSE(ccmEarlyLateIdentical): result(int: earlyLateIdentical);</pre>
C syntax	<pre>typedef struct { DCM_INTEGER earlyLateIdentical; } T_EarlyLateIdentical; int ccmEarlyLateIdentical(const DCM_STD_STRUCT *std_struct, T_EarlyLateIdentical *rtn);</pre>

This function can be called by the ICM to determine whether, given the same load capacitance, ccmCalcDelaySlew would return identical delay and slew values for both early and late propagation modes. If this is true, the function shall return a value of one (1). Otherwise, a value of zero (0) shall be returned.

When the CCM results would be identical and other factors such as the early and late load models are the same, the ICM can avoid redundant computations.

10.24.4.4 ccmGetICMcontrolParams

Table 395 provides information on ccmGetICMcontrolParams.

Table 395—ccmGetICMcontrolParams

Function name	ccmGetICMcontrolParams
Arguments	None
Result	Ceffective iteration limit, Relative tolerance, Absolute tolerance
Standard Structure fields	None
DCL syntax	EXPOSE(ccmGetICMcontrolParams): result(int: ceffIterLimit; double: relTol, absTol);
C syntax	typedef struct { DCM_INTEGER ceffIterLimit; DCM_DOUBLE relTol, absTol; } T_ICMcontrolParams; int ccmIterICMcontrolParams(const DCM_STD_STRUCT *std_struct, T_ICMcontrolParams *rtn);

This function returns parameters used by the ICM to control delay and slew calculations.

The parameters *relTol* and *absTol* are the relative and absolute tolerances, respectively, used to determine when convergence of iterations used to compute effective load capacitance and corresponding delays and slews has been reached. When the magnitudes of the differences in delay, slew, and other quantities being compared between one iteration and the next by the ICM fall below values determined using the following expression, the ICM shall terminate the iteration process and return the final quantities to the CCM:

$$\text{relTol} * \text{value} + \text{absTol}$$

Here, *value* is the smaller of the two values (*delay*, *slew*, etc.) being compared from the current and previous iterations.

ceffIterLimit is the number of iterations after which an attempt to converge shall be aborted. When this occurs, the ICM function shall return an error to the CCM.

10.24.4.5 icmCalcOutputResistances

Table 396 provides information on icmCalcOutputResistances.

Table 396—icmCalcOutputResistances

Function name	icmCalcOutputResistances
Arguments	Output-pin pointer, Early slew, Late slew, Early XWF data structure, Late XWF data structure
Result	Early and late output resistances
Standard Structure fields	CellName, calcMode, processVariation, block, pathData (timing-pin-specific), cellData (timing)
DCL syntax	EXPOSE(icmCalcOutputResistances_ft): passed(pin: outputPin; double: earlySlew, lateSlew; void: earlyXWF, lateXWF) result(double: earlyRout, lateRout);

C syntax	<pre> typedef struct { DCM_DOUBLE earlyRout; DCM_DOUBLE lateRout; } T_Rout; int icmCalcOutputResistances(const DCM_STD_STRUCT *std_struct, T_Rout *rtn, DCM_PIN outputPin, DCM_DOUBLE earlySlew, lateSlew, const DCM_STRUCT *earlyXWF, *lateXWF); </pre>

This function returns output resistances for the passed pin corresponding to the early and late slews and XWF structures provided. These structures shall be compatible with the specific ICM implementation to which the function belongs. The function's semantics shall be identical to those of *dpcmCalcOutputResistances* (see 10.21.15.6).

10.24.4.6 icmCalcTotalLoadCapacitances

Table 397 provides information on *icmCalcTotalLoadCapacitances*.

Table 397—icmCalcTotalLoadCapacitances

Function name	<i>icmCalcTotalLoadCapacitances</i>
Arguments	ICM load models
Result	Early and late load capacitances
Standard Structure fields	CellName, calcMode, processVariation, block, toPoint, sinkEdge, sourceMode, sinkMode, cellData (timing), pathData (timing-arc specific)
DCL syntax	<pre> forward calc(icmCalcTotalLoadCapacitances_ft): passed(void: minLoadModel, maxLoadModel) result(double: earlyLoadCap, lateLoadCap); </pre>
C syntax	<pre> typedef struct { DCM_DOUBLE earlyLoadCap, lateLoadCap; } T_TotalLoadCaps; int icmCalcTotalLoadCapacitances(const DCM_STD_STRUCT *std_struct, T_TotalLoadCaps *rtn, const DCM_STRUCT *minLoadModel, const DCM_STRUCT *maxLoadModel); </pre>

This function calculates total load capacitances presented to the pin identified by the *toPoint* field in the Standard Structure by an interconnect network. Different versions of the network can be used for early and late calculations. Each capacitance returned shall include the total capacitance of the corresponding network, including the capacitances of all attached pins.

10.24.4.7 icmCalcXWF

Table 398 provides information on *icmCalcXWF*.

Table 398—icmCalcXWF

Function name	icmCalcXWF
Arguments	Pin pointer, Edge, Early slew, Late slew
Result	Early and late XWF data structures
Standard Structure fields	CellName, calcMode, processVariation, pathData (timing-pin-specific), cellData (timing)
DCL syntax	<pre>forward calc(icmCalcXWF_ft): passed(pin: pinPointer; int: edge; double: earlySlew, lateSlew) result(void: earlyXWF, lateXWF);</pre>
C syntax	<pre>typedef struct { DCM_STRUCT *earlyXWF; DCM_STRUCT *lateXWF; } T_XWF; int icmCalcXWF(const DCM_STD_STRUCT *std_struct, T_XWF *result, DCM_PIN pinPointer, DCM_EdgeTypes edge, DCM_DOUBLE earlySlew, DCM_DOUBLE lateSlew);</pre>

This function returns pointers to XWF structures for the passed pin and edge direction corresponding to the early and late slews provided. These structures shall be compatible with the specific ICM implementation to which the function belongs. The function's semantics shall be identical to those of *dpcmCalcXWF* (see 10.23.8.3.3).

10.24.5 ICM initialization

Before the ICM is requested to perform any calculations, it shall be initialized by the CCM. During this initialization, the ICM shall obtain, via pointers, access to CCM functions and to application functions needed by the ICM. The ICM shall in turn provide pointers to its functions to the CCM.

The initialization procedure used shall support both the use of different ICM implementations in different technology families in the library as well as the use of a particular ICM by multiple technology families. This procedure shall also support specification of the ICM implementation used for a particular technology through modification of the bill of materials (BOM) for that technology. Recompile of the source code for either the CCM in that technology or the ICM itself shall not be required to change the ICM implementation used.

10.24.5.1 icmInit

Table 399 provides information on icmInit.

Table 399—icmInit

Function name	icmInit
Arguments	None
Result	ICM function pointers
Standard Structure fields	None
DCL syntax	<pre>typedef (icmFuncs) : result (icmBuildInterconnectModels_ft() : icmBuildInterconnectModels_fp; icmBuildLoadModels_ft() : icmBuildLoadModels_fp; icmCalcCellDelaySlew_ft() : icmCalcCellDelaySlew_fp; icmCalcInterconnectDelaySlew_ft() : icmCalcInterconnectDelaySlew_fp; icmCalcOutputResistances_ft() : icmCalcOutputResistances_fp; icmCalcTotalLoadCapacitances_ft() : icmCalcTotalLoadCapacitances_fp; icmCalcXWF_ft() : icmCalcXWF_fp;); EXPOSE(icmInit) : result (icmFuncs);</pre>
C syntax	<pre>typedef struct { DCM_GeneralFunction icmBuildInterconnectModels_fp; DCM_GeneralFunction icmBuildLoadModels_fp; DCM_GeneralFunction icmCalcCellDelaySlew_fp; DCM_GeneralFunction icmCalcInterconnectDelaySlew_fp; DCM_GeneralFunction icmCalcOutputResistances_fp; DCM_GeneralFunction icmCalcTotalLoadCapacitances_fp; DCM_GeneralFunction icmCalcXWF_fp; } ICM_Funcs;</pre>

This function initializes an ICM; during this process, it shall obtain pointers to CCM and application functions. It shall return a table of pointers to the ICM functions exposed to the CCM. So that the CCM in a technology family can call ICM functions without explicit knowledge of a specific ICM technology family and using a Standard Structure set to its own technology (not that of the ICM), these functions shall be called via the pointers returned.

Each ICM shall have its own, specific DCL implementation of *icmInit*. Different function pointers shall be returned by different ICM implementations. For an ICM that is implemented primarily in C, this function can in turn call a C function to perform the actual initialization. An example of the DCL code used to call such a C function is shown in 10.24.5.2.

10.24.5.2 ICM DCL header file (icm.h)

This subclause contains the DCL header file (icm.h) that shall be used both in the implementation of an ICM and in CCM code that interacts with that ICM. So that the use of the ICM function pointers returned by *icmInit* can be made transparent to the bulk of the CCM, macros through which ICM functions can be called are provided.

```
#ifndef _ICM_H
#define _ICM_H
```

```
typedef(ivcurve):
result(double var [*] var: voltage, current);

#define RESISTOR 0
#define CAPACITOR 1
#define INDUCTOR 2
#define MUTUAL_INDUCTANCE 3
#define LOSSLESS_TRANSMISSION_LINE_TIME_DELAY_BASED 4
#define LOSSLESS_TRANSMISSION_LINE_FREQUENCY_BASED 5
#define LOSSY_TRANSMISSION_LINE_RLC 6
#define LOSSY_TRANSMISSION_LINE_RC 7
#define LOSSY_TRANSMISSION_LINE_LC 8
#define LOSSY_TRANSMISSION_LINE_LG 9
#define DIODE 10
#define VOLTAGE_SOURCE 11

typedef(parasiticElement):
result(
    int var: elementType, node0Index, node1Index, node2Index, node3Index ;
    double var: value0, value1, value2, value3, value4 ;
    var ivcurve var: ivCurve ;
    string var: modelName ;
    int var: railIndex ;
    void var: ownerPrivate
);

#define INTERMEDIATE_NODE 0
#define SINK_NODE 1
#define SOURCE_NODE 2
#define AGGRESSOR_SOURCE_NODE 3

typedef(parasiticSubnet):
result(
    int var: changed ;
    var parasiticElement var[*] var: parasiticElementArray ;
    int var[*] var: portMap, nodeMap, nodeTypeList ;
    var parasiticSubnet var: nextSubnet, prevSubnet ;
    TECH_TYPE var: techFamily ;
    void var: ownerPrivate
);

forward calc(icmBuildInterconnectModels_ft) impure:
passed(
    parasiticSubnet: minParasitics, maxParasitics ;
    void: minNetModel, maxNetModel ;
    int: minDrivingPinNodeNumber, minSinkPinNodeNumber,
    maxDrivingPinNodeNumber, maxSinkPinNodeNumber
)
result(
    void: minInterconnectModel, maxInterconnectModel, newMinNetModel, newMaxNetModel
);

forward calc(icmBuildLoadModels_ft) impure:
passed(
    parasiticSubnet: minParasitics, maxParasitics ;
    void: minNetModel, maxNetModel ;
```

```
int: minDrivingPinNodeNumber, maxDrivingPinNodeNumber
)
result(
  void: minLoadModel, maxLoadModel, newMinNetModel, newMaxNetModel
);

typedef(timeRange):
result(double var: earlyTime, lateTime);

forward calc(icmCalcCellDelaySlew_ft) impure:
passed(void: minLoadModel, maxLoadModel)
result(
  float: earlyDelay, lateDelay, earlySlew, lateSlew ;
  void: earlyXWF, lateXWF ;
  double: earlyCeffective, lateCeffective ;
  timeRange[*]: earlyInteractWindows, lateInteractWindows
);

forward calc(icmCalcInterconnectDelaySlew_ft) impure:
passed(void: minInterconnectModel, maxInterconnectModel)
result(
  float: earlyDelay, lateDelay, earlySlew, lateSlew ;
  void: earlyXWF, lateXWF ;
  timeRange[*]: earlyInteractWindows, lateInteractWindows
);

forward calc(icmCalcOutputResistances_ft) impure:
passed(
  pin: outputPin ;
  double: earlySlew, lateSlew ;
  void: earlyXWF, lateXWF
)
result(double: earlyRout, lateRout);

forward calc(icmCalcTotalLoadCapacitances_ft) impure:
passed(void: minLoadModel, maxLoadModel)
result(double: earlyLoadCap, lateLoadCap);

forward calc(icmCalcXWF_ft) impure:
passed(
  pin: pinPointer ;
  int: edge ;
  double: earlySlew, lateSlew
)
result(void: earlyXWF, lateXWF);

typedef(icmFuncs):
result(
  icmBuildInterconnectModels_ft() impure:
  icmBuildInterconnectModels_fp ;

  icmBuildLoadModels_ft() impure:
  icmBuildLoadModels_fp ;

  icmCalcCellDelaySlew_ft() impure:
  icmCalcCellDelaySlew_fp ;
```

```

icmCalcInterconnectDelaySlew_ft() impure:
icmCalcInterconnectDelaySlew_fp ;

icmCalcOutputResistances_ft() impure:
icmCalcOutputResistances_fp ;

icmCalcTotalLoadCapacitances_ft() impure:
icmCalcTotalLoadCapacitances_fp ;

icmCalcXWF_ft() impure: icmCalcXWF_fp ;
);

#ifdef _ICM_BODY

/* ICM module body declarations */

expose(ccmCalcDelaySlew):
passed(
  int: propagationMode ;
  double: loadCapacitance
)
local(
:issue_message(0, SEVERE,
'\nICM dummy function ccmCalcDelaySlew called!\n')
)
result
  double: delayValue = 0.0, delayCapGradient = 0.0, slewValue = 0.0, slewCapGradient = 0.0
);

expose(ccmEarlyLateIdentical):
local(
:issue_message(0, SEVERE,
'\nICM dummy function ccmEarlyLateIdentical called!\n')
)
result(int: earlyLateIdentical = 0);

expose(ccmGetICMcontrolParams):
local(
:issue_message(0, SEVERE,
'\nICM dummy function ccmGetICMcontrolParams called!\n')
)
result(
  int: ceffilterLimit = 0 &
  double: relTol = 0.0, absTol = 0.0
);

#else

/* ICM client (not module body) declarations */

import expose(icmInit) optional:
result(icmFuncs: funcs);

assign(icmFuncTbl):
local(

```

```

TECH_TYPE: techType = map_tech_family(CONTROL_PARM).TT
)
result(icmFuncs: funcs = techType::icmInit().funcs);

calc(LATENT_EXPRESSION) impure:
local(:icmFuncTbl());

#define icmBuildInterconnectModels( \
minParasitics, maxParasitics, minNetModel, maxNetModel, \
minDrivingPinNodeNumber, minSinkPinNodeNumber, \
maxDrivingPinNodeNumber, maxSinkPinNodeNumber \
) \
icmFuncTbl.funcs.icmBuildInterconnectModels_fp( \
minParasitics, maxParasitics, minNetModel, maxNetModel, \
minDrivingPinNodeNumber, minSinkPinNodeNumber, \
maxDrivingPinNodeNumber, maxSinkPinNodeNumber \
)

#define icmBuildLoadModels( \
minParasitics, maxParasitics, minNetModel, maxNetModel, \
minDrivingPinNodeNumber, maxDrivingPinNodeNumber \
) \
icmFuncTbl.funcs.icmBuildLoadModels_fp( \
minParasitics, maxParasitics, minNetModel, maxNetModel, \
minDrivingPinNodeNumber, maxDrivingPinNodeNumber \
)

#define icmCalcCellDelaySlew( \
minLoadModel, maxLoadModel \
) \
icmFuncTbl.funcs.icmCalcCellDelaySlew_fp( \
minLoadModel, maxLoadModel \
)

#define icmCalcInterconnectDelaySlew( \
minInterconnectModel, maxInterconnectModel \
) \
icmFuncTbl.funcs.icmCalcInterconnectDelaySlew_fp( \
minInterconnectModel, maxInterconnectModel \
)

#define icmCalcOutputResistances( \
pinPointer, earlySlew, lateSlew, earlyXWF, lateXWF \
) \
icmFuncTbl.funcs.icmCalcOutputResistances_fp( \
pinPointer, earlySlew, lateSlew, earlyXWF, lateXWF \
)

#define icmCalcTotalLoadCapacitances( \
minLoadModel, maxLoadModel \
) \
icmFuncTbl.funcs.icmCalcTotalLoadCapacitances_fp( \
minLoadModel, maxLoadModel \
)

```

```
#define icmCalcXWF(\
    pinPointer, edge, earlySlew, lateSlew \
)\
    icmFuncTbl.funcs.icmCalcXWF_fp(\
    pinPointer, edge, earlySlew, lateSlew \
)\
#endif /* ifdef _ICM_BODY */
#endif /* _ICM_H */
```

10.24.5.3 ICM DCL initialization example

This subclause contains an example implementation of *icmInit* that calls a C function to perform the actual initialization.

```
tech_family(ACME_ICM);

#define _ICM_BODY
#include "icm.h"

%{
typedef struct {
    DCM_GeneralFunction icmBuildInterconnectModels_fp;
    DCM_GeneralFunction icmBuildLoadModels_fp;
    DCM_GeneralFunction icmCalcCellDelaySlew_fp;
    DCM_GeneralFunction icmCalcInterconnectDelaySlew_fp;
    DCM_GeneralFunction icmCalcOutputResistances_fp;
    DCM_GeneralFunction icmCalcTotalLoadCapacitances_fp;
    DCM_GeneralFunction icmCalcXWF_fp;
} ICM_Funcs;

void acme_icm_init(
    const DCM_STD_STRUCT *std_struct,
    ICM_Funcs *const rtn,
    DCM_GeneralFunction appGetAggressorOverlapWindows_fp,
    DCM_GeneralFunction appGetXWF_fp,
    DCM_GeneralFunction ccmCalcDelaySlew_fp,
    DCM_GeneralFunction ccmEarlyLateIdentical_fp,
    DCM_GeneralFunction ccmGetICMcontrolParams_fp,
    DCM_StructWizard DCM_TimeRange_wizard_fp,
    DCM_StructWizard DCT_OverlapWindow_wizard_fp,
    DCM_DataScope *dcm_rule_anchor);
}%

typedef(overlapWindow):
result(
int var: overlapPresent &
double var: earlyTime, lateTime, earlySlew, lateSlew,
earlyDriverResistance, lateDriverResistance &
void var: earlyXWF, lateXWF
);
external(appGetAggressorOverlapWindows) optional impure:
passed
var overlapWindow transient[*]: earlyOverlapArray, lateOverlapArray
```



```

)
result(int: ignore);

external(appGetXWF) optional:
passed(void: earlyXWF, lateXWF)
result(int: ignore);

expose(icmInit):
local(
appGetAggressorOverlapWindows():
appGetAggressorOverlapWindows_fp = appGetAggressorOverlapWindows;
appGetXWF() pure inconsistent: appGetXWF_fp = appGetXWF;
ccmCalcDelaySlew(): ccmCalcDelaySlew_fp = ccmCalcDelaySlew;
ccmEarlyLateIdentical():
ccmEarlyLateIdentical_fp = ccmEarlyLateIdentical;
ccmGetICMcontrolParams():
ccmGetICMcontrolParams_fp = ccmGetICMcontrolParams;
var icmFuncs: acme_icm_funcs = new(var icmFuncs);
:$acme_icm_init(
$std_struct,
$(ICM_Funcs *)$(acme_icm_funcs),
$(DCM_GeneralFunction)$(appGetAggressorOverlapWindows_fp),
$(DCM_GeneralFunction)$(appGetXWF_fp),
$(DCM_GeneralFunction)$(ccmCalcDelaySlew_fp),
$(DCM_GeneralFunction)$(ccmEarlyLateIdentical_fp),
$(DCM_GeneralFunction)$(ccmGetICMcontrolParams_fp),
$DCM_DCM_WIZ_ITEM($timeRange),
$DCM_DCM_WIZ_ITEM($overlapWindow),
$DCM_RULE_ANCHOR()
)
)
result(icmFuncs: funcs = acme_icm_funcs);

```

10.25 DCL run-time support

Subclauses 10.25.1 through 10.25.4 describe the support functions implicitly supplied by the delay calculation system. These functions are dynamically linked to the application using standard C practices.

10.25.1 Array manipulation functions

These functions allow the application to manipulate array data that is returned by the DPCM.

10.25.1.1 dcmRT_copy_DCM_ARRAY

Function name	dcmRT_copy_DCM_ARRAY
Arguments	DCM_ARRAY, DCM_AATTS
Result	DCM_ARRAY
Standard Structure fields	None
C syntax	<pre>DCM_ARRAY *dcmRT_copy_DCM_ARRAY (const DCM_STD_STRUCT *std, DCM_ARRAY *originalArray, DCM_AATTS attributes);</pre>

The application service *dcmRT_copy_DCM_ARRAY* allocates a new *DCM_ARRAY* and copies the contents of the original array into the newly allocated one. The *attributes* argument shall have the value 0xFF.

10.25.1.2 dcmRT_new_DCM_ARRAY

Table 400 provides information on `dcmRT_new_DCM_ARRAY`.

Table 400—`dcmRT_new_DCM_ARRAY`

Function name	<code>dcmRT_new_DCM_ARRAY</code>
Arguments	Standard Structure, number of dimensions, vector of elements per dimension, size of each element,
Result	<code>DCM_ARRAY</code>
Standard Structure fields	None
C syntax	<pre>DCM_ARRAY *dcmRT_new_DCM_ARRAY (const DCM_STD_STRUCT *std_struct, int numDims, int *elementsPer, size_t elementSize, DCM_ATYPE elementType, DCM_AATTS attributes, DCM_AINIT initialize, DCM_ArrayInitUserFunction initializer);</pre>

The application service `dcm_new_DCM_ARRAY` allocates a new array according to the number of dimensions, the number of elements in each dimension, and the size of each element. There are options to control the how an array is initialized. The maximum value for `numDims` is 255. When the system does not allocate the required space, an error is generated. A newly created array is locked once.

The `DCM_ATYPE` is an enumeration that enumerates the possible types of `DCM_ARRAY` data; see Table 401.

Table 401—`DCM_ATYPE` enumeration

C syntax	<pre>typedef enum DCM_Array_Element_Types {DCM_ATYPE_ERROR, /* Error */ DCM_ATYPE_Integer = 1, /* INTEGER */ DCM_ATYPE_String = 2, /* STRING */ DCM_ATYPE_Double = 3, /* DOUBLE */ DCM_ATYPE_Float = 4, /* FLOAT or NUMBER in TABLEDEF DATA*/ DCM_ATYPE_Function = 5, /* Function array. */ DCM_ATYPE_Complex = 6, /* complex number */ DCM_ATYPE_Void = 7, /* void */ DCM_ATYPE_Structure = 8, /* dcm structure pointer/ DCM_ATYPE_Function_PureC = 9, /* array of pure consistent function pointers */ DCM_ATYPE_Function_PureI = 10, /* array of pure inconsistent function pointers */ DCM_ATYPE_Character = 11, /* array of char */ DCM_ATYPE_Short = 12, /* array of short */ DCM_ATYPE_Long = 13, /* array of long */ DCM_ATYPE_Pin = 14, /* array of pin */ DCM_ATYPE_Function_Launchable = 15, /* array of launchable function pointer structures */ DCM_ATYPE_Array = 0x10, /* array of arrays */ /***** ** Future additions go here. *****/ DCM_ATYPE_MAX} /* Ceiling. */ DCM_ATYPE;</pre>

This enumeration represents a switch on which the allocator knows the initialization values for the DCM types.

The *attributes* argument shall have the value 0xFF.

The *DCM_AINIT* (Table 402) represents an enumeration that controls the initialization of *DCM_ARRAY*s.

Table 402—DCM_AINIT enumeration

C syntax	<pre>typedef enum DCM_Array_Initialization {DCM_AINIT_doNotInitialize, DCM_AINIT_initAllZeroes, DCM_AINIT_initByType, DCM_AINIT_useFunction, DCM_AINIT_compilerInits, DCM_AINIT_debugInits, DCM_AINIT_MAX} DCM_AINIT;</pre>

- *DCM_AINIT_doNotInitialize* indicates no initialization is to be performed on the array returned. With this option, the data space is left in the same state as the operating system that furnished it.
- *DCM_AINIT_initAllZeroes* initializes all the data bytes to the value of zero.
- *DCM_AINIT_initByType* causes the data bytes to be initialized to bit pattern corresponding to the *DCM_ATYPE* and its related pattern.
- *DCM_AINIT_useFunction* causes the *dcmRT_new_DCM_ARRAY* to call the supplied initializer function. If this scalar value is present and the initializer parameter is 0 (zero), then no initialization of the array elements takes place.
- *DCM_AINIT_compilerInits* causes the *dcmRT_new_DCM_ARRAY* to insert the values the compiler would initialize the variables to.
- *DCM_AINIT_debugInits* causes the *dcmRT_new_DCM_ARRAY* to insert the values the compiler would initialize the variables to if the debug flags were set during compilation.

If *DCM_AINIT_initByType* is passed in, then the following type preinitialization patterns occur:

- *INTEGER* sets each element to *MININT*
- *CHAR* set each element to the minimum value a char type can hold.
- *SHORT* set each element to the minimum value a short type can hold.
- *LONG* set each element to the minimum value a long type can hold.
- *PIN* sets each element to *NULL* (0)
- *VOID* sets each element to *NULL*; (0) *STRING* sets each element to *NULL* (0)
- *STRUCTURE* sets each element to *NULL* (0)
- *FUNCTION* sets each element to *NULL* (0). *DOUBLE* sets each element to *NaN*. *FLOAT* sets each element to *NaN*

The *DCM_ArrayInitUserFunction* (Table 403) is a prototype definition for an application supplied function to initialize the data elements of a *DCM_ARRAY*. When supplied, this function shall accept a *DCM_ARRAY* pointer that the application uses to initialize the data members.

Table 403—DCM_ArrayInitUserFunction

C syntax	<code>typedef int(*DCM_ArrayInitUserFunction)(DCM_ARRAY *);</code>

10.25.1.3 dcm_sizeof_DCM_ARRAY

Table 404 provides information on `dcm_RT_sizeof_DCM_ARRAY`.

Table 404—dcmRT_sizeof_DCM_ARRAY

Function name	<code>dcmRT_sizeof_DCM_ARRAY</code>
Arguments	<code>DCM_STD_STRUCT *</code> , <code>DCM_ARRAY</code>
Result	Size of the DCM array
Standard Structure fields	None
C syntax	<code>int dcmRT_sizeof_DCM_ARRAY(const DCM_STD_STRUCT *std, DCM_ARRAY *array);</code>

The application service `dcmRT_sizeof_DCM_ARRAY` returns the number of bytes the `DCM_ARRAY`'s data elements consume. The application shall pass in the `DCM_ARRAY` pointer to be evaluated. If there is an error, a value of `-1` is returned.

NOTE—Zero is a valid size for an empty array.

10.25.1.4 dcmRT_claim_DCM_ARRAY

Table 405 provides information on `dcmRT_claim_DCM_ARRAY`.

Table 405—dcmRT_claim_DCM_ARRAY

Function name	<code>dcmRT_claim_DCM_ARRAY</code>
Arguments	<code>DCM_STD_STRUCT *</code> , <code>DCM_ARRAY</code>
Result	Return code
Standard Structure fields	None
C syntax	<code>int dcmRT_claim_DCM_ARRAY(const DCM_STD_STRUCT *std, DCM_ARRAY *array);</code>

`dcmRT_claim_DCM_ARRAY` claims the array. The array shall persist until it is disclaimed. The array may be claimed multiple times, by both the application and the DPCM.

If for any reason the system encounters an error, a nonzero value is returned; otherwise, a successful return value of zero is returned.

10.25.1.5 dcmRT_disclaim_DCM_ARRAY

Table 406 provides information on `dcmRT_disclaim_DCM_ARRAY`.

Table 406—dcmRT_disclaim_DCM_ARRAY

Function name	dcmRT_disclaim_DCM_ARRAY
Arguments	DCM_STD_STRUCT *, DCM_ARRAY *
Result	Return code
Standard Structure fields	None
C syntax	int dcmRT_disclaim_DCM_ARRAY(const DCM_STD_STRUCT *std, DCM_ARRAY *array);

dcmRT_disclaim_DCM_ARRAY disclaims the array. The array shall be deleted when it has been disclaimed as many times as it was claimed. Neither the application nor the *DPCM* shall disclaim the array more times than the application or the *DPCM* respectively claimed it.

If for any reason the system encounters an error, a nonzero value is returned; otherwise, a successful return value of zero is returned.

10.25.2 Memory management

A *DCM_STRUCT* is a library-specific collection of data that includes memory management support. Memory management within the *DPCM* shall conform to the following behavior:

- When the *DPCM* creates a new array or structure, it keeps a *count* (initialized to 1) indicating how many references to that object exist.
- When the *DPCM* returns or passes an *DCM_ARRAY* or *DCM_STRUCT* to the application, the claim count for that object is incremented. During the application's next call to the *DPCM* at a primary entry point, such as *modelSearch*, *delay*, *slew*, or *check*, the claim count is decremented.
- When the application claims the object or the library creates a reference to the object, the claim count is incremented. When the application disclaims the object or the library removes a reference to the object, the claim count is decremented.
- The application is responsible for disclaiming the object as many times as it has claimed it. The library is responsible for removing references to the object it no longer needs.
- Once the reference count becomes 0, the memory is returned to the system.

10.25.3 Structure manipulation functions

DCM_STRUCT type represents a pointer to a C style data structure. *DCM* Structures like *DCM_ARRAY* contain additional information in a header. The following functions manipulate the data contained in these headers. The application shall not directly allocate, free, or manipulate structure header information.

The *DPCM* supports concurrent operations on separate contexts. To prevent data corruption structures that may be accessed concurrently on more than one context shall have the *SYNC* attribute. The *SYNC ATTRIBUTE* indicates the structure shall be locked before accessing its data fields and unlocked after the last field update or access is completed.

10.25.3.1 dcmRT_claim_DCM_STRUCT

Table 407 provides information on *dcmRT_claim_DCM_STRUCT*.

Table 407—dcmRT_claim_DCM_STRUCT

Function name	dcmRT_claim_DCM_STRUCT
Arguments	DCM_STD_STRUCT *, DCM_STRUCT *
Result	Return code
Standard Structure fields	None
C syntax	<pre>int dcmRT_claim_DCM_STRUCT (const DCM_STD_STRUCT *std, DCM_STRUCT *structure);</pre>

dcmRT_claim_DCM_STRUCT claims the structure. The shall persist until it is disclaimed. The structure may be claimed multiple times by both the application and the DPCM.

If for any reason the system encounters an error, a nonzero value is returned; otherwise, a successful return value of zero is returned.

10.25.3.2 dcmRT_disclaim_DCM_STRUCT

Table 408 provides information on dcmRT_disclaim_DCM_STRUCT.

Table 408—dcmRT_disclaim_DCM_STRUCT

Function name	dcmRT_disclaim_DCM_STRUCT
Arguments	DCM_STD_STRUCT *, DCM_STRUCT*
Result	Return code
Standard Structure fields	None
C syntax	<pre>int dcmRT_disclaim_DCM_STRUCT (const DCM_STD_STRUCT *std, DCM_STRUCT *structure);</pre>

dcmRT_disclaim_DCM_STRUCT disclaims the structure. The structure shall be deleted when it has been disclaimed as many times as it was claimed. Neither the application nor the *DPCM* shall disclaim the structure more times than the application or the *DPCM* respectively claimed it.

If for any reason the system encounters an error, a nonzero value is returned; otherwise, a successful return value of zero is returned.

10.25.3.3 Locking options

The serialization options can take on several forms, as follows:

- Exclusive access can be granted that allows for both safe reading and safe writing of the structure's data fields.
- Nonexclusive access for the purpose of reading the structure's data fields. With this option, changing the values in the structure's data fields may not be safe, but reading the fields is.

10.25.3.3.1 dcmRT_longlock_DCM_STRUCT

Table 409 provides information on dcmRT_longlock_DCM_STRUCT.

Table 409—dcmRT_longlock_DCM_STRUCT

Function name	dcmRT_longLock_DCM_STRUCT
Arguments	Standard Structure pointer, Structure pointer, Options
Result	error indicator
Standard Structure fields	None
C syntax	<pre>int dcmRT_longLock_DCM_STRUCT (const DCM_STD_STRUCT *context, DCM_STRUCT *dcmStruct, unsigned int lockOptions);</pre>

The application shall call *dcmRT_longlock_DCM_STRUCT* prior to accessing a *DCM_STRUCT* with the SYNC attribute. *dcmRT_longlock_DCM_STRUCT* serializes the access to data that may be updated on more than one context. If no other context has blocked access to the *DCM_STRUCT*, then *dcmRT_longlock_DCM_STRUCT* grants access to the structure, blocks access to all other contexts, and returns. When another context has blocked access to the *DCM_STRUCT*, *dcmRT_longlock_DCM_STRUCT* waits until the existing block is removed, grants access, blocks all other contexts, and then returns. *dcmRT_longlock_DCM_STRUCT* shall not block if it is called more than once for the same structure on the same context that it has already been granted access to. *dcmRT_longlock_DCM_STRUCT* returns a value of zero unless an error is encountered such as making this call with a *DCM_STRUCT* argument that does not have the SYNC attribute.

10.25.3.3.2 dcmRT_longunlock_DCM_STRUCT

Table 410 provides information on *dcmRT_longunlock_DCM_STRUCT*.

Table 410—dcmRT_longunlock_DCM_STRUCT

Function name	dcmRT_longUnlock_DCM_STRUCT
Arguments	Standard Structure pointer, Structure pointer, Options
Result	error code
Standard Structure fields	None
C syntax	<pre>int dcmRT_longUnlock_DCM_STRUCT (const DCM_STD_STRUCT *context, DCM_STRUCT *dcmStruct, unsigned int lockOptions);</pre>

The application shall call *dcmRT_longunlock_DCM_STRUCT* to remove a lock on the structure set by *dcmRT_lock_DCM_STRUCT*. The application shall call *dcmRT_longunlock_DCM_STRUCT* as many times as it has called *dcmRT_longlock_DCM_STRUCT* on the same structure.

10.25.3.3.2.1 Context

A *context* is a space, plane pair. The *Standard Structure* contains the context is the application is operating on. The is the context the application is requesting access for.

10.25.3.3.2.2 Sync structures

Sync structures are *DCM_STRUCT* types that have the SYNC attribute. The SYNC attribute indicates the data contained within the structure shall only be accessed or updated by one context at a time. To gain access, the application shall call *dcmRT_lock_DCM_STRUCT* supplying the access context prior to accessing the structure's data elements. When a context has finished accessing the structure's data, the context shall release the block by calling *dcmRT_unlock_DCM_STRUCT*.

10.25.3.3.2.3 Options

The application shall set the options argument to the value of 0x00000002. All other values are reserved for use by the compiler.

10.25.3.3.2.4 Error code

If the lock is successful, then `dcmRT_longunlock_DCM_STRUCT` shall return a value of zero. Any other value shall be an error.

10.25.3.4 dcmRT_getNumDimensions

Table 411 provides information on `dcmRT_getNumDimensions`.

Table 411—`dcmRT_getNumDimensions`

Function name	<code>dcmRT_getNumDimensions</code>
Arguments	<code>DCM_STD_STRUCT *</code> , <code>DCM_ARRAY *</code>
Result	Number of dimensions
Standard Structure fields	None
C syntax	<pre>int dcmRT_getNumDimensions(const DCM_STD_STRUCT *std, DCM_ARRAY *array);</pre>

The application service `dcmRT_getNumDimensions` returns the number of dimensions defined for the `DCM_ARRAY` passed in by the application.

If for any reason there is an error in determining the number of dimensions, a value of `-1` is returned, otherwise the number of dimensions is returned.

10.25.3.5 dcmRT_getNumElementsPer

Table 412 provides information on `dcmRT_getNumElementsPer`.

Table 412—`dcmRT_getNumElementsPer`

Function name	<code>dcmRT_getNumElementsPer</code>
Arguments	<code>DCM_STD_STRUCT *</code> , <code>DCM_ARRAY *</code> , <code>int *</code>
Result	Number of elements in each dimension
Standard Structure fields	None
C syntax	<pre>int *dcmRT_getNumElementsPer(const DCM_STD_STRUCT *std, DCM_ARRAY *array, int *answer);</pre>

The application service `dcmRT_getNumElementsPer` returns an array whose elements are the length of each dimension of the array argument.

The application shall supply a `DCM_ARRAY` and an integer array where the application service can place its results. `dcmRT_getNumElementsPer` places in each element of the answer array the number of elements in the corresponding `DCM_ARRAY`, where the zeroth index of the `DCM_ARRAY` corresponds to the zeroth element of the answer array. If the service detects an error, it returns `(int*) 0`; otherwise, it returns the answer.

10.25.3.6 dcmRT_getNumElements

Table 413 provides information on dcmRT_getNumElements.

Table 413—dcmRT_getNumElements

Function name	dcmRT_getNumElements
Arguments	DCM_STD_STRUCT *, DCM_ARRAY *, int
Result	Number of dimensions
Standard Structure fields	None
C syntax	<pre>int dcmRT_getNumElements(const DCM_STD_STRUCT *std, DCM_ARRAY *array, int dimension);</pre>

The application service dcmRT_getNumElements returns the number of elements for the dimension specified. If an error is encountered, the value returned is –1. The dimension parameter passed in from the application shall be between 0 and the *number_of_dimensions* –1.

10.25.3.7 dcmRT_getElementType

Table 414 provides information on dcmRT_getElementType.

Table 414—dcmRT_getElementType

Function name	dcmRT_getElementType
Arguments	DCM_STD_STRUCT *, DCM_ARRAY *
Result	Type of element
Standard Structure fields	None
C syntax	<pre>DCM_ATYPE dcmRT_getElementType(const DCM_STD_STRUCT *std, DCM_ARRAY *array);</pre>

The application service dcmRT_getElementType is passed a DCM_ARRAY and returns the type of elements stored.

If the application service detects an error, the element type DCM_ATYPE_ERROR is returned.

10.25.3.8 dcmRT_arraycmp

Table 415 provides information on dcmRT_arraycmp.

Table 415—dcmRT_arraycmp

Function name	dcmRT_arraycmp
Arguments	DCM_STD_STRUCT *, Two DCM_ARRAYs *
Result	evaluation
Standard Structure fields	None
C syntax	<pre>int dcmRT_arraycmp(const DCM_STD_STRUCT *std, DCM_ARRAY *a1, DCM_ARRAY *a2);</pre>

The application service dcmRT_arraycmp is passed two DCM_ARRAYs and compares them for equality. If the two arrays contain (bit-by-bit) identical data, the value of zero is returned; otherwise, a nonzero value is returned.

10.25.4 Initialization functions

Initialization functions are called by an application to load or unload a DPCM, or to set a universal storage manager or message handler. These functions are called as part of the process of preparing the system to accept a DPCM or to clean up after one has been terminated. They are available to the application because the dynamically loaded modules that make up a DPCM are not yet in memory and cannot perform these operations.

An application shall call *dcmSetNewStorageManager* to assert common storage management between the DPCM and the application.

10.25.4.1 dcmRT_InitRuleSystem

Table 416 provides information on *dcmRT_InitRuleSystem*.

Table 416—dcmRT_InitRuleSystem

Function name	dcmRT_InitRuleSystem
Arguments	Function pointer to a memory allocator, Function pointer to a free function, Function pointer to a reallocation function, Pointer to a location where a return code may be stored.
Result	Standard Structure pointer
Standard Structure fields	None
C syntax	<pre>typedef void * (*DCM_Malloc_Type) (size_t); typedef void (*DCM_Free_Type) (void *); typedef void * (*DCM_Realloc_Type) (void *, size_t); DCM_STD_STRUCT *dcmRT_InitRuleSystem (DCM_Malloc_Type new_malloc, DCM_Free_Type new_free, DCM_Realloc_Type new_realloc, int* rc);</pre>

Creates and returns a Standard Structure representing the initialized run-time support system. This Standard Structure contains no context; that is, it is not associated with any space or plane. The primary purpose for this Standard Structure is to load the initial rule system by calling *dcmRT_BindRule*. The standard returned from this call shall not be used to call any dpcm or modeling function.

The input arguments initialize in the run-time system the proper malloc, free, and realloc functions for the library to use. The application shall either supply all three function pointers or three pointer values of null (0). If these function pointers are null (0), the library shall insert its own run-time versions of these calls. The run-time versions of these calls shall be parallel thread-safe.

The integer pointer parameter *rc* shall contain the return code this function establishes. After a call to *dcmRT_InitRuleSystem*, the value of *rc* shall be zero in the cases where the function completes successfully; all other values of *rc* shall be considered an error.

10.25.4.2 dcmRT_BindRule

Table 417 provides information on *dcmRT_BindRule*.

Table 417—dcmRT_BindRule

Function name	dcmRT_BindRule
Arguments	Standard Structure pointer, Rule name, Rule environment variable name, Table environment variable name, Control parameter, Space name, Plane name, Message handler function, Application services, Library services
Result	Standard Structure pointer
Standard Structure fields	None
C syntax	<pre>DCM_STD_STRUCT *dcmBindRule(const DCM_STD_STRUCT *context, const char *rootSubruleName, const char *rulePathEnvName, const char *tablePathEnvName, const char *controlParm, const char *spaceName, const char *planeName, DCM_Message_Intercept_Type intercept, DCM_FunctionTable *externals, DCMTransmittedInfo *xmit);</pre>

This creates a new space then loads and links the specified primary rule system into that space. dcmRT_BindRule also creates the first plane associated with the new space.

The context argument is any valid Standard Structure.

The rootSubruleName argument is a pointer to a string containing the name of the primary (*root*) library rule to be loaded into the space.

The rulePathEnvName argument is a pointer to a string containing the name of an environment variable. This environment variable shall contain the paths the library uses to locate modules during the loading process.

The tablePathEnvName argument is a pointer to a string containing the name of an environment variable. This environment variable shall contain the paths the library is to use during the loading of tables.

The controlParm parameter is a pointer to a string that the root rule receives during the loading process.

The spaceName parameter is a pointer to a string that contains the name to be associated with the space created by the call to dcmRT_BindRule. If a zero (NULL) is supplied, the name of the space defaults to the index associated with the space.

The planeName parameter is a pointer to a string that contains the name to be associated with the plane created by the call to dcmRT_BindRule. If a zero (NULL) is supplied, the name of the plane defaults to the index associated with the plane.

The intercept parameter is a function pointer the library can use to display any messages it may generate. If no function pointer is supplied default operating services shall be utilized for this purpose.

The externals parameter is an array of structures containing the function pointer name pairs the library uses to link application services to external statements of the same name. The *DCM_FunctionTable* structure contains the application's *EXTERNAL* function pointer pairs. It is the application's responsibility to create this structure.

The xmit argument is an address where the library can place an array of function pointer name pairs the application uses to link exposed library services of the same name. The *DCMTransmittedInfo* is a structure containing all the *EXPOSE* function pointer pairs along with a pointer to DPCM functions *modelSearch*,

delay, *slew*, and *check*. Each *EXPOSE* function pointer pair consists of a string containing the name of the *EXPOSE* as it is seen in the subrule and a pointer to that function's entry point.

dcmRT_BindRule returns a Standard Structure pointer initialized to the context the root rule was loaded into. This context shall consist of a new space and a new plane. If the rule cannot be found or there was an error in loading, the pointer returned is zero. When the application communicates with the root rule on this space, it shall supply Standard Structures that are initialized to the same space.

After the *root* subrule is loaded and its initialization function has been called, the application then does the following:

- Uses the *DCMTransmittedInfo* to initialize its pointers to the DPCM services required. *dcmRT_FindFunction* and *dcmRT_QuietFindFunction* are used to locate the function pointers associated with each *EXPOSE* desired.
- Initializes its modeling function pointer by the named field within the *DCMTransmittedInfo* for DPCM functions *modelSearch*, *delay*, *slew*, and *check*.

10.25.4.3 dcmRT_AppendRule

Table 418 provides information on *dcmRT_AppendRule*.

Table 418—dcmRT_AppendRule

Function name	<i>dcmRT_AppendRule</i>
Arguments	Standard Structure pointer, rule name, Rule environment variable name, Table environment variable name, Control parameter, flags, options, Application services, Library services
Result	return code
Standard Structure fields	None
C syntax	<pre>int dcmRT_AppendRule(DCM_STD_STRUCT *context, const char *rulename, const char *rulePathEnvName, const char *tablePathEnvName, const char *controlParm, unsigned int flags, unsigned int options, DCM_FunctionTable *externals, DCMTransmittedInfo *xmit);</pre>

This adds additional DCL subrules to the *DPCM* during execution. *dcmRT_AppendRule* does not alter subrules in the current *DPCM*.

The context argument is a valid Standard Structure created on the context the subrules are to be added to.

The ruleName argument is a pointer to a string containing the name of the library rule to be added to the space identified in the context argument.

The rulePathEnvName argument is a pointer to a string containing the name of the environment variable. The environment variable shall contain the paths to the library modules to be loaded.

The tablePathEnvName argument is a pointer to a string containing the name of the environment variable. The environment variable shall contain the paths to the library tables to be loaded.

The controlParm parameter is a pointer to a string that the added rule receives during the loading process.

The flags argument shall have a value of zero.

The options argument shall have a value of zero.

The externals parameter is a delta array of structures containing the function pointer name pairs the library uses to link application services to external statements of the same name. The *DCM_FunctionTable* structure contains the additional application *EXTERNAL* function pointer pairs. It is the application's responsibility to create this structure. A value of zero/NULL shall mean no additional services are provided.

The xmit argument is an address where the library can place the array of function pointer name pairs the application uses to link exposed library services of the same name. The *DCMTransmittedInfo* is a structure containing all the *EXPOSE* function pointer pairs along with a pointer to DPCM functions *modelSearch*, *delay*, *slew*, and *check*. Each *EXPOSE* function pointer pair consists of a string containing the name of the *EXPOSE* as it is seen in the subrule and as a pointer to that function's entry point.

This function may only be called following a successful call to *dcmRT_BindRule* and preceding a successful call to *dcmRT_UnbindRule* on the associated space.

The *dcmRT_AppendRule* function returns an int return code. A value of zero indicates the subrule was loaded successfully; any other value indicates an error occurred during the load process.

After the additional subrule is loaded and its initialization function has been called, the application then uses the *DCMTransmittedInfo* to initialize its pointers to the DPCM services required. *dcmRT_FindFunction* and *dcmRT_QuickFindFunction* are used to locate the function pointers associated with each additional *EXPOSE* desired.

10.25.4.4 dcmRT_UnbindRule

Table 419 provides information on *dcmRT_UnbindRule*.

Table 419—*dcmRT_UnbindRule*

Function name	<i>dcmRT_UnbindRule</i>
Arguments	Standard Structure pointer
Result	None
Standard Structure fields	None
C syntax	<code>int dcmRT_UnbindRule(DCM_STD_STRUCT *context);</code>

This unloads the subrules associated with a context.

The context argument is a Standard Structure associated with the space the subrules are to be unloaded from. A way to meet this requirement is to use the Standard Structure returned from the *dcmRT_BindRule* that loaded these subrules as the context argument to this call.

dcmRT_UnbindRule returns an integer return code with a zero value when the function completes without error; otherwise, a nonzero value is returned.

10.25.4.5 dcmRT_FindFunction

Table 420 provides information on *dcmRT_FindFunction*.

Table 420—dcmRT_FindFunction

Function name	dcmRT_FindFunction
Arguments	Standard Structure pointer, EXPOSE function name, Function table
Result	None
Standard Structure fields	None
C syntax	<pre>DCM_GeneralFunction dcmRT_FindFunction (const DCM_STD_STRUCT *context, char *fcnName, DCM_FunctionTable exposes);</pre>

Locates the passed *EXPOSE* function within the loaded DPCM and returns a pointer to the function.

The context argument is a Standard Structure associated with the space the function table returned by the dcmRT_BindRule call earlier. A way to meet this requirement is to use the Standard Structure returned from the dcmRT_BindRule as the context argument to this call; however, any Standard Structure from any plane on this space is acceptable.

The fcnName argument is a pointer to the requested *EXPOSE* name.

The exposes argument is the initialization table set returned by the dcmRT_BindRule call.

The result is a pointer to the *EXPOSE* function within the DPCM.

When the matching function cannot be found, an error message is issued and the returned pointer is zero.

10.25.4.6 dcmRT_FindAppFunction

Table 421 provides information on dcmRT_FindAppFunction.

Table 421—dcmRT_FindAppFunction

Function name	dcmRT_FindAppFunction
Arguments	Standard Structure pointer EXTERNAL function name
Result	None
Standard Structure fields	None
C syntax	<pre>int dcmRT_FindAppFunction (const DCM_STD_STRUCT *context, char *fcnName);</pre>

The context argument identifies the space the test shall be performed on.

Determines whether the application defined the indicated *EXTERNAL* function. This function returns a nonzero value if the application did define the function; otherwise, a zero value is returned.

10.25.4.7 dcmRT_QuietFindFunction

Table 422 provides information on dcmRT_QuietFindFunction.

Table 422—dcmRT_QuietFindFunction

Function name	dcmRT_QuietFindFunction
Arguments	Standard Structure pointer EXPOSE function name, Function table
Result	None
Standard Structure fields	None
C syntax	<pre>DCM_GeneralFunction dcmRT_QuietFindFunction (const DCM_STD_STRUCT *context, char *fcnName, DCM_FunctionTable *exposes);</pre>

This function is the same as *dcmRT_FindFunction*, except no error is issued if the function is not found

10.25.4.8 dcmRT_MakeRC

Table 423 provides information on dcmRT_MakeRC.

Table 423—dcmRT_MakeRC

Function name	dcmRT_MakeRC
Arguments	Standard Structure pointer, Message number, Message severity, Error code address
Result	complete error code
Standard Structure fields	None
C syntax	<pre>int dcmRT_MakeRC(const DCM_STD_STRUCT *context, int messageNumber, DCM_Message_Severities severity, int *errorCode);</pre>

This returns an error code constructed from the message number and severity arguments that does not conflict with internal DCL reserved codes (such as those returned from *dcmRT_HardErrorRC*).

The function returns as an integer value from the constructed error code by taking the absolute value of messageNumber and adding 10 000 (the upper limit of the message numbers reserved for DCL system itself). The constructed code is also copied to the address specified by the third argument. If severity is zero or one, then the returned value shall be all zeros; otherwise, the severity byte is used as the most significant byte of the return value and the constructed error code is use as the least significant bytes.

If the message number contains any bits in the high-order byte, an informative message is issued.

The context argument is any valid Standard Structure.

10.25.4.9 dcmRT_HardErrorRC

Table 424 provides information on dcmRT_HardErrorRC.

Table 424—dcmRT_HardErrorRC

Function name	dcmRT_HardErrorRC
Arguments	Standard Structure pointer, Message severity
Result	None
Standard Structure fields	None
C syntax	<pre>int dcmHardErrorRC(const DCM_STD_STRUCT * context, DCM_Message_Severities severity);</pre>

This returns a return code constructed from the message severity argument. If the message severity is inform or warning (see Table 101), the return code is 0. Otherwise, the return code has the passed message severity with a message number of 0x00EEEEEE.

The context argument is any valid Standard Structure.

10.25.4.10 dcmRT_SetMessageIntercept

Table 425 provides information on dcmRT_SetMessageIntercept.

Table 425—dcmRT_SetMessageIntercept

Function name	dcmRT_SetMessageIntercept
Arguments	Standard Structure pointer, Application-defined function for printing messages
Result	None
Standard Structure fields	None
C syntax	<pre>DCM_Message_Intercept_Type dcmRT_SetMessageIntercept (const DCM_STD_STRUCT * context, DCM_Message_Intercept_Type msgfn);</pre>

This sets the function pointer to be used to print messages generated by the DPCM or library. This function may be called at any time. This function is a pointer to the previous message intercept function or is NULL if there was no prior function.

For consistent message handling, the application shall set the message handler before it loads a DPCM. Message handlers can be changed at any time the application chooses; however, if a change is done after the DPCM is loaded, only those messages occurring after the change are directed to the new handler. Messages prior to that shall be handled in a default manner as determined by DCL (if the message handler was not set) or as dictated by the previous call to *dcmSetMessageIntercept*.

The context argument identifies the space this message intercept function shall be applied to.

10.25.4.11 dcmRT_IssueMessage

Table 426 provides information on dcmRT_IssueMessage.

Table 426—dcmRT_IssueMessage

Function name	dcmRT_IssueMessage
Arguments	Standard Structure pointer, Message number, Message severity, Message format string [format arguments]
Result	None
Standard Structure fields	None
C syntax	<pre>int dcmRT_IssueMessage (const DCM_STD_STRUCT *context, int msgNum, DCM_Message_Severities msgSev, char* msgFormat [, ...]);</pre>

This prints a message using the current message function in effect. This function assembles a complete DCL message from the severity, message number, format, and format arguments. *dcmRT_IssueMessage* can be called by an application as well as by the DPCM (inside *INTERNAL* or in-line C code) to generate a DCL-style message. Use this function, instead of direct calls to *printf()* or *fprintf()*, to help ensure proper ordering of both DPCM and application messages in the same output stream (see 10.25.4.10).

This function takes a minimum of four arguments. The first is the *DCM_STD_STRUCT* pointer. This argument contains the context this message is to be associated with. The message number and severity arguments shall follow the interface conventions defined in integer return code. The fourth argument is a format string that follows the conventions of the *C printf* function. The remaining arguments, if any, fulfill the conversion specifications identified in the format string.

This function returns the same integer value as *dcmRT_MakeRC* would if it were passed the same severity and message number (see 10.25.4.8).

This function shall not buffer any messages.

10.25.4.12 dcmRT_new_DCM_STD_STRUCT

Table 427 provides information on *dcmRT_new_DCM_STD_STRUCT*.

Table 427—dcmRT_new_DCM_STD_STRUCT

Function name	<i>dcmRT_new_DCM_STD_STRUCT</i>
Arguments	Standard Structure pointer
Result	None
Standard Structure fields	None
C syntax	<pre>DCM_STD_STRUCT *dcmRT_new_STD_STRUCT (const DCM_STD_STRUCT * context);</pre>

This is a constructor function to allocate and properly initialize a Standard Structure. The newly allocated Standard Structure shall be associated with the same context as the Standard Structure argument.

10.25.4.13 dcmRT_delete_DCM_STD_STRUCT

Table 428 provides information on *dcmRT_delete_DCM_STD_STRUCT*.

Table 428—dcmRT_delete_DCM_STD_STRUCT

Function name	<i>dcmRT_delete_DCM_STD_STRUCT</i>
Arguments	Standard Structure pointer
Result	None
Standard Structure fields	None
C syntax	<pre>void dcmRT_delete_DCM_STD_STRUCT (DCM_STD_STRUCT *std_struct);</pre>

This is a destructor function to free a Standard Structure.

10.25.4.14 dcmRT_setTechnology

Table 429 provides information on *dcmRT_setTechnology*.

Table 429—dcmRT_setTechnology

Function name	dcmRT_setTechnology
Arguments	Standard Structure pointer, Pointer to technology name
Result	None
Standard Structure fields	None
C syntax	<pre>const char* dcmRT_setTechnology (DCM_STD_STRUCT *std_struct, const char *tech_name);</pre>

A DPCM can contain one or more technologies. If no technology was specified, then a DPCM contains the GENERIC technology. If a single technology was specified, then the DPCM contains that specified technology. If multiple technologies were specified, then the DPCM contains the GENERIC technology (at least for the root subrule) as well as the other specified technologies.

At any time, there is a current technology set in the Standard Structure; the DPCM as a whole has no notion of what technology is considered current. A newly created Standard Structure selects a technology according to the following rules:

- If the DPCM has no technology or has a single technology, that technology is selected.
- If the DPCM has multiple technologies, the first technology loaded is selected.

An application can change the technology selected by a Standard Structure by calling either this function (dcmRT_setTechnology) or dcmRT_takeMappingOfNugget (see 10.25.4.10) to modify the passed Standard Structure to select the specified technology.

An application can switch between technologies by either using a single Standard Structure and calling dcmRT_setTechnology or dcmRT_takeMappingOfNugget or maintaining multiple Standard Structures, each of which has been modified to select a different technology, and choosing the appropriate structure to pass across the PI.

10.25.4.15 dcmRT_getTechnology

Table 430 provides information on dcmRT_getTechnology.

Table 430—dcmRT_getTechnology

Function name	dcmRT_getTechnology
Arguments	Standard Structure pointer
Result	None
Standard Structure fields	None
C syntax	<pre>const char* dcmRT_getTechnology (const DCM_STD_STRUCT *std_struct);</pre>

This returns the technology name of the Standard Structure in use and returns a 0 value if completion is unsuccessful.

NOTE—Do not free the result string as it is constant.

10.25.4.16 dcmRT_getAllTechs

Table 431 provides information on dcmRT_getAllTechs.

Table 431—dcmRT_getAllTechs

Function name	dcmRT_getAllTechs
Arguments	Standard Structure pointer
Result	None
Standard Structure fields	None
C syntax	char ** dcmRT_getAllTechs (const DCM_STD_STRUCT *std_struct);

This returns an array of all technologies named within the current DPCM and Returns a 0 value if completion is unsuccessful.

NOTE—Do not free the result within the calling application; use dcmRT_FreeAllTechs instead.

10.25.4.17 dcmRT_freeAllTechs

Table 432 provides information on dcmRT_freeAllTechs.

Table 432—dcmRT_freeAllTechs

Function name	dcmRT_freeAllTechs
Arguments	Standard Structure pointer, Array pointer
Result	None
Standard Structure fields	None
C syntax	void dcmRT_freeAllTechs (const DCM_STD_STRUCT *std_struct, char **techArray);

This frees storage occupied by the string array returned by a call to *dcmRT_getAllTechs*. Although the first argument is required to be a Standard Structure, this function ignores the structure's contents. This function is passed a pointer to the pointer array to be freed.

10.25.4.18 dcmRT_isGeneric

Table 433 provides information on dcmRT_isGeneric.

Table 433—dcmRT_isGeneric

Function name	dcmRT_isGeneric
Arguments	Standard Structure pointer
Result	None
Standard Structure fields	None
C syntax	int dcmRT_isGeneric(const DCM_STD_STRUCT *std_struct);

This returns whether or not the Standard Structure is currently pointing to the generic technology. A nonzero return value indicates the Standard Structure is pointing to the generic technology. A zero return value indicates the Standard Structure is not pointing to the generic technology.

10.25.4.19 dcmRT_takeMappingOfNugget

Table 434 provides information on dcmRT_takeMappingOfNugget.

Table 434—dcmRT_takeMappingOfNugget

Function name	dcmRT_takeMappingOfNugget
Arguments	Standard Structure pointer, Technology nugget
Result	None
Standard Structure fields	None
C syntax	<pre>int dcmRT_takeMappingOfNugget (DCM_STD_STRUCT *std_struct, DCM_TechFamilyNugget *tech_nugget);</pre>

This sets the Standard Structure argument to use the technology for which *tech_nugget* was computed (using *dcmRT_mapNugget*).

A nonzero return code indicates the function did not successfully complete. A zero return code indicates success.

10.25.4.20 dcmRT_registerUserObject

Table 435 provides information on dcmRT_registerUserObject.

Table 435—dcmRT_registerUserObject

Function name	dcmRT_registerUserObject
Arguments	Standard Structure pointer, Pointer to application structure to be registered
Result	None
Standard Structure fields	None
C syntax	<pre>int dcmRT_registerUserObject (DCM_STD_STRUCT *std_struct, void *app_struct_to_register);</pre>

This registers an application-specific data structure with the passed Standard Structure. A registered user object is a structure that is application-private with the provision the first member of that structure is a function pointer to the destructor function, which takes as its only argument the pointer to the registered user object. This registered structure can be deleted later by the application (see 10.25.4.21). This function is passed by a pointer to the application structure to be registered. A nonzero return code indicates the function did not successfully complete. A zero return code indicates successful registration.

10.25.4.21 dcmRT_DeleteRegisteredUserObjects

Table 436 provides information on dcmRT_DeleteRegisteredUserObjects.

Table 436—dcmRT_DeleteRegisteredUserObjects

Function name	dcmRT_DeleteRegisteredUserObjects
Arguments	Standard Structure pointer
Result	None
Standard Structure fields	None
C syntax	<pre>void dcmRT_DeleteRegisteredUserObjects (DCM_STD_STRUCT *std_struct);</pre>

This deletes all the registered user objects that were registered to the specified Standard Structure.

10.25.4.22 dcmRT_DeleteOneUserObject

Table 437 provides information on dcmRT_DeleteOneUserObject.

Table 437—dcmRT_DeleteOneUserObject

Function name	dcmRT_DeleteOneUserObject
Arguments	Standard Structure pointer, Pointer to object to be deleted
Result	None
Standard Structure fields	None
C syntax	<pre>void dcmRT_DeleteOneUserObject (DCM_STD_STRUCT *std_struct, void *userObject);</pre>

This locates and deletes the user object contained within the specified Standard Structure. See 10.25.4.21 for a description of registered user object.

10.26 Calculation functions

These predefined functions allow the application to request basic functions from the DPCM. These functions are used by an application to call for the delay, slew, and timing checks of a cell, as well as cell modeling.

10.26.1 delay

Table 438 provides information on delay.

Table 438—delay

Function name	delay
Arguments	Standard Structure pointer
Result	DCM_DELAY_REC pointer
Standard Structure fields	CellName, fromPoint, toPoint, calcMode, slew.early, slew.late, block, sourceEdge, sinkEdge, sourceMode, sinkMode, pathData (timing arc or pin specific), cellData (timing)
C syntax	<pre>typedef struct { float early, late; } DCM_DELAY_REC; int delay(const DCM_STD_STRUCT *std_struct, DCM_DELAY_REC *delay_value);</pre>

This function is called by the application to calculate the delay for a modeled arc. the values are returned through a pointer to *DCM_DELAY_REC*, which is a structure containing two floats, one for early delay and one for late delay.

During model elaboration (see 10.27.1), the DPCM passes a *PATH_DATA* pointer to the application for each timing arc. An application shall save the *PATH_DATA* pointer values and put the appropriate one into the Standard Structure before calling the DPCM to calculate a delay value.

The DPCM recognizes a *PATH_DATA* pointer value of 0 (zero) as a special indicator the DPCM shall evaluate the default *DELAY* function (identified by the *DEFAULT* modifier). An error occurs if the *PATH_DATA* pointer is 0 and no such *DEFAULT* function was specified within a DCL subrule of the DPCM.

This function is not called explicitly by name but is accessed via a pointer supplied in the DCMTransmittedInfo structure as a result of the first call to the subrule returned by *dcmRT_BindRule* (see 10.25.4.2).

10.26.2 slew

Table 439 provides information on slew.

Table 439—slew

Function name	slew
Arguments	Standard Structure pointer
Result	DCM_SLEW_REC pointer
Standard Structure fields	CellName, fromPoint, toPoint, calcMode, slew.early, slew.late, block, sourceEdge, sinkEdge, sourceMode, sinkMode, pathData (timing arc or pin specific), cellData (timing)
C syntax	<pre>typedef struct { float early, late; } DCM_SLEW_REC; int slew (const DCM_STD_STRUCTURE *std_struct, DCM_SLEW_REC *slew_value);</pre>

This function is called by the application to calculate the slew for a modeled arc. The values are returned through a pointer to *DCM_SLEW_REC*, which is a structure containing two floats, one for early slew and one for late slew.

During model elaboration (see 10.27.1), the DPCM passes a *PATH_DATA* pointer to the application for each timing arc. The application shall save the *PATH_DATA* pointer values and put the appropriate one into the Standard Structure before calling the DPCM to calculate a slew value.

The DPCM recognizes a *PATH_DATA* pointer value of 0 (zero) as a special indicator; the DPCM shall evaluate the default *SLEW* function (identified by the *DEFAULT* modifier). An error occurs if the *PATH_DATA* pointer is 0 and no such *DEFAULT* function was specified within a DCL subrule of the DPCM.

This function is not called explicitly by name but is accessed via a pointer supplied in the DCMTransmittedInfo structure as a result of the first call to the subrule returned by *dcmRT_BindRule* (see 10.25.4.2).

10.26.3 check

Table 440 provides information on check.

Table 440—check

Function name	check
Arguments	Standard Structure pointer
Result	DCM_CHECK_REC pointer
Standard Structure fields	CellName, fromPoint, toPoint, calcMode, slew.early, slew.late, sourceEdge, sinkEdge, sourceMode, sinkMode, pathData (timing-arc-specific), cellData (timing)

C syntax	<pre>typedef struct { float bias; } DCM_CHECK_REC; int check (const DCM_STD_STRUCT *std_struct, DCM_CHECK_REC *test_bias);</pre>

This function is called by the application to compute the timing offset between signals. The bias value returned represents the difference in arrival times between the specified signal (or data) pin and the specified reference (or clock) pin.

This function is called by the application to perform timing checks (time offset between signals) for a modeled test arc. The function result, conveyed through a pointer to the *DCM_CHECK_REC*, is a float containing the bias, or offset, for the requested timing check.

The slew, edge, and mode fields of the *DCM_STD_STRUCT* are the same locations as those for the *delay* and *slew* functions, but the interpretations are different (see 10.12.1).

Bias values represent the minimum time between the reference and the signal. A bias may be computed for all the different types of tests. For example, a positive bias value for a hold test represents the minimum time the data shall remain stable after the clock has transitioned, whereas a positive bias value for a setup test represents the minimum time the data shall remain stable before the clock transitions (see Figure 16). Setup and hold tests assume the reference is a clock and the signal is the data.

This function is not called explicitly by name, but is accessed via a pointer supplied in the *DCMTransmittedInfo* structure as a result of the first call to the subrule returned by *dcmRT_BindRule*.

Clock separation implies the reference and signal edges are two different clocks, as shown in Figure 17.

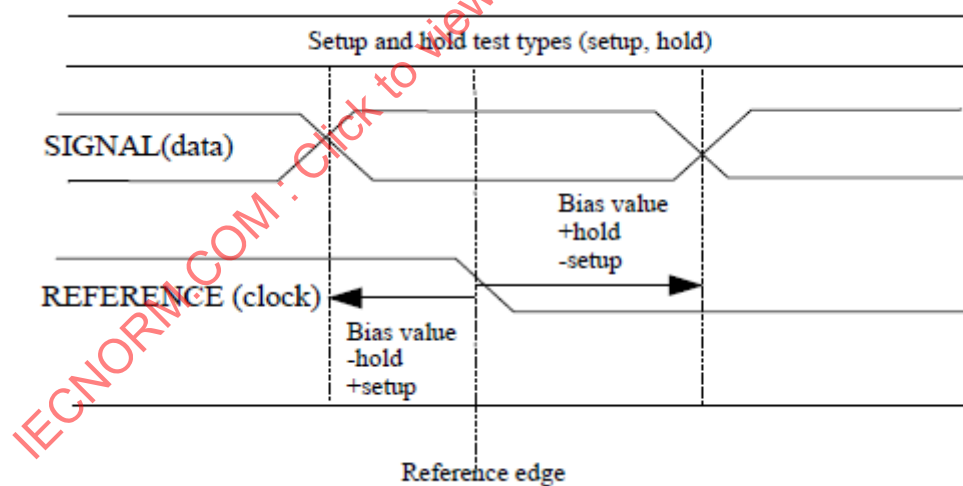


Figure 16—Clock separation

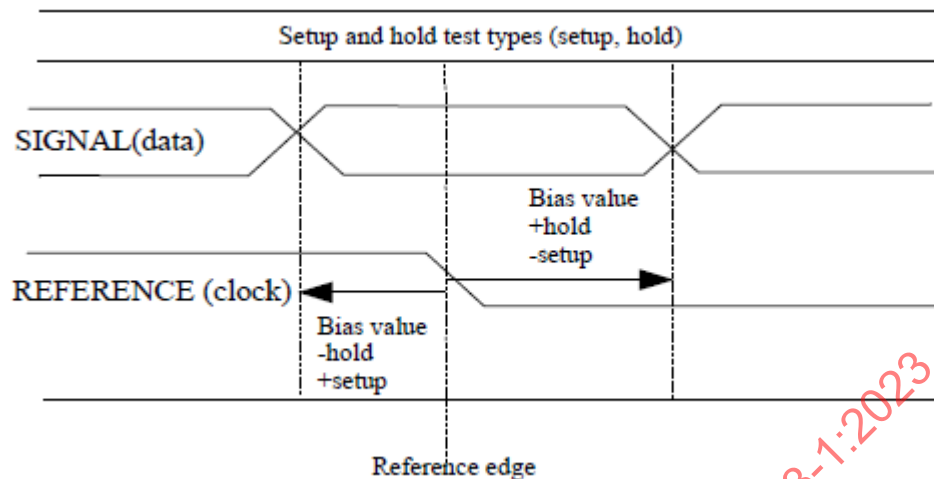


Figure 17—Bias calculation

Clock pulse width implies the signal and reference are the same clock but different edges (see Figure 18). For clock pulse width checking, where the checking is a function of the rising and falling slews, the rising slew is passed within the *EARLY_SLEW* and the falling slew is passed within the *LATE_SLEW*.

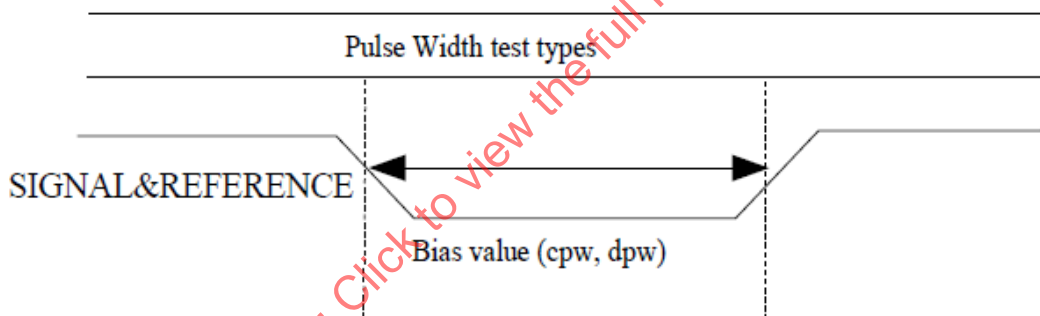


Figure 18—Clock pulse width

10.27 Modeling functions

The application initiates the modeling process with a call to *modelSearch*. This call results in multiple callbacks to the application, which convey the model's structure from DPCM to the application. These functions allow the application to remember the characteristics of each cell and not have to recompute them each time a particular cell is encountered during processing. All these modeling callback functions shall be implemented by any standards-compliant application. The modeling functions return the number 0 (zero) on success and nonzero on error or failure.

10.27.1 modelSearch

Table 441 provides information on *modelSearch*.

Table 441—modelSearch

Function name	modelSearch
Arguments	Standard Structure pointer
Result	None
Standard Structure fields	block, CellName, inputPins, outputPins, inputPinCount, outputPinCount, nodes, nodeCount
C syntax	<code>int modelSearch(DCM_STD_STRUCT *std_struct);</code>

Called by the application for each instance of a cell that has to be modeled. Given the flexibility of the DCL language, models may depend on instance-specific data. If a cell's model does not depend on instance-specific data, an application can elaborate that model once and share the results among all instances of that cell. If a cell's model does depend on instance-specific data, the application shall elaborate that model for each instance. An application can safely use models using either of the following two approaches:

- Choose to always elaborate (i.e., call *modelSearch*) models for every instance.
- Instrument all “callback” PI functions (those application functions the DPCM can call) so the application can determine—after *modelSearch* returns control to the application—which functions (if any) were called during the computations that are part of *modelSearch* processing. Given that information, the application can then decide whether the particular model elaboration can be shared with other instances of the same cell.

modelSearch causes callbacks to the application that describe the behavior for the specified cell. The internal timing arcs are constructed and stored within the application through the DPCM callbacks of necessary modeling functions for paths and their propagation properties (see 10.27). The application shall save this information because there is no other mechanism for conveying model structure to the application.

modelSearch enables the DPCM to “instruct” the application to model the requirements of the technology cell. The DPCM translates DCL mode operators into enumerations passed to the application through calls to *newDelayMatrixRow*. Each call to *newDelayMatrixRow* transfers the edge at the start of the timing arc, the corresponding edge at the output, the mode of propagation at the start of the timing arc (for that edge), and the mode of propagation at the end of the timing arc for its corresponding edge.

The DPCM may generate multiple calls to *newDelayMatrixRow* and shall make enough calls to enumerate all of the edges and modes propagated. After enumerating all edge propagations, the DPCM begins enumerating the timing arcs that have these edge propagations. The three different types of timing arcs that can be generated are as follows:

- Within in a cell
- From a cell's output to all receivers it drives
- From all sources to a cell's input

Timing arcs that are within a cell, or with a known start and end point, are identified through calls to *newPropagateSegment*. Timing arcs that have only a known starting point are identified by calls to *newNetSourcePropagateSegments*. Timing arcs with only a known ending point are identified by calls to *newNetSinkPropagateSegments* to support the calls for delay, slew, or check.

For *newPropagateSegment* calls, the application generates one segment that spans the known start and end points. For *newNetSourcePropagateSegment* calls, the application is expected to connect the cell's output pin to all receivers on the interconnect. For *newNetSinkPropagateSegments* calls, the application is expected to generate timing arcs from all drivers on the interconnect.

The general sequence of events is initiated by calling *modelSearch* on a particular cell. The DPCM then calls the application back via sequences of *newDelayMatrixRow*, followed by sequences of one of the propagate segment calls, depending on the type of function. DCL *PATH* and *BUS* statements call *newPropagateSegment*. DCL *OUTPUT* statements call *newNetSourcePropagateSegments*. As the application fulfills the requests from the DPCM, it creates a complete timing graph for both the intercell and intracell timing arcs.

The DPCM describes test segments in a manner similar to timing arcs. First, the DPCM calls back the application through *newTestMatrixRow* to establish test properties. These properties include the edge of the reference, the edge of the signal, the test mode for the signal, and the test mode for the reference. There may be more than one of these calls in a sequence before the DPCM has completely described all the test properties. After completing the test property description, the DPCM continues calling the application back, indicating the pins that shall be tested with these properties. The DPCM calls back the application through *newAltTestSegment* for each signal and reference point to be tested.

This function is not called explicitly by name but is accessed via a pointer supplied in the DCMTransmittedInfo structure as a result of the first call to the subrule returned by demBT_BindRule.

Unless the application can define internal timing points understood by the *MODELPROC*, the Standard Structure fields *NODES* and *NODE_COUNT* shall be set to 0 (zero).

The *inputPins* field of the *Standard Structure* shall contain handles for all input and bidirectional pins that are used by the block identified in the *Standard Structure*.

The *outputPins* field of the *Standard Structure* shall contain handles for all output and bidirectional pins that are used by the block identified in the *Standard Structure*.

The application shall be responsible for keeping the input and output parts of a bidirectional pin separate and distinguishable. The two parts shall have their own unique pin handles, so they can be viewed as separate pins by the library. The library shall be able to associate a unique *pathData* pointer with each of these parts, and the application shall store such a pointer with the appropriate part when requested to do so by the library.

10.27.2 Mode operators

Mode operators describe propagation and test properties. For calls to *newDelayMatrixRow* and *newTestMatrixRow*, the mode operator is not passed to the application. Instead, the mode operators are split into two enumerations, each representing the starting and ending point of the timing arc shown in Table 442. *newTestMatrixRow* decomposes the mode operator <-> into two calls, one for the late mode operator (<-) and one for the early mode operator (->). *newTestMatrixRow* shall not support the operators <-X-> and ->X<-.

Table 442—Mode propagation operators

Mode operator	Timing arc starting point	Timing arc end point	Description
<-	DCM_LateMode	DCM_SameMode	Propagate only late mode times.
->	DCM_EarlyMode	DCM_SameMode	Propagate only early times.
<->	DCM_BothModes	DCM_SameMode	Propagate both early and late times.
<-X->	DCM_Late	DCM_Early	Propagate earliest arriving late mode edge
->X<-	DCM_Early	DCM_Late	Propagate latest arriving early mode edge

The application shall communicate the mode of operation for delay, slew, and check calculations by putting an enumeration representing the operations requested into the Standard Structure. The mode enumerations are presented to the DPCM as the mode for the early mode evaluation, the mode for the late mode

evaluation, and the mode for the check evaluation. The enumeration values are the same for the application communication to the DPCM, but the legal combinations are different (see 10.27.4).

The mode operators for delay and slew are shown in Table 443.

Table 443—Mode computation operators for delay and slew

Mode operator	sourceMode/EARLY_MODE	sinkMode/LATE_MODE	Description
<-	DCM_LateMode	DCM_LateMode	Compute late mode values; short circuiting the early mode calculations is permitted.
->	DCM_EarlyMode	DCM_EarlyMode	Compute early values; the late mode value may be short circuited.
<->	DCM_EarlyModes	DCM_LateMode	Compute both early and late values.
<-X->	DCM_EarlyMode	DCM_EarlyMode	Compute earliest arriving late mode edge and the earliest arriving early mode edge
->X<-	DCM_LateMode	DCM_LateMode	Compute latest arriving early mode edge and the latest arriving late mode edge

The application shall only be required to communicate one value of mode for calls to check. The legal combinations of mode for test are in Table 444.

Table 444—Mode operator enumerators for check

Mode operator	sourceMode/TEST_MODE	Description
<-	DCM_LateMode	Compute late mode bias value a.
->	DCM_EarlyMode	Compute early mode bias value.

10.27.3 Arrival time merging

Static timing dictates when two signals converge at a point, a data reduction can occur. To achieve this, at each convergence point converging late mode edges, the application retains the information associated with the latest arriving edge. An analogous case exists for the early mode edges; in this situation, the application retains the information associated with the earliest edge. In the situation where both modes are propagated, but in the complement mode (such as <-X->), the beginning mode propagation indicates the propagation to be altered during convergence.

The *lateMode*, *complementMode* (<-X->) instructs the application to do the following:

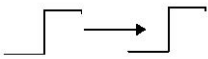
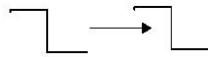
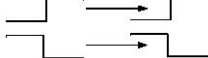
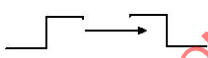
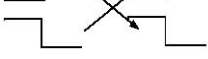
- Propagate both mode edges
- Reduce the early mode edges by keeping the earliest arriving early mode edges of the same type
- Reduce the late mode edges by keeping the earliest of the late mode edges of the same type.
- The *earlyMode*, *complementMode* (->X<-) instructs the application to do the following:
- Propagate both mode edges
- Reduce the late mode edges by keeping the latest arriving late mode edges of the same type
- Reduce the early mode edges by keeping the latest arriving early mode edges of the same type

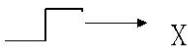
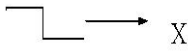
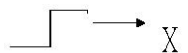
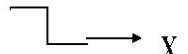
10.27.4 Edge propagation communication to the application

The DPCM represents edge propagation as a pair of enumerations. Table 445 is a representative sampling of enumeration pairs that can exist. The enumeration pairs are the same for test segments as they are for propagation segments. Also, the following is true:

- Whenever the ending edge specification is identical to the starting edge and the edge type is either *RISE* or *FALL*, *DCM_SameEdge* shall be passed to the application for the ending edge.
- Whenever the ending edge specification is identical to the starting edge and that edge is *BOTH*, *DCM_BothEdges* shall be passed to the application for the ending edge.
- The ending edge scalar shall always be passed as *DCM_SameMode* for mode operators *->*, *<-*, and *<->*.

Table 445—Edge propagation enumeration pairs

Propagation	Starting enumeration/ Reference enumeration	Ending enumeration/ Signal enumeration	Meaning
	DCM_RisingEdge	DCM_SameEdge	Rise to Rise
	DCM_FallingEdge	DCM_SameEdge	Fall to Fall
	DCM_BothEdge	DCM_SameEdge	Rise to Rise and Fall to Fall
	DCM_RisingEdge	DCM_FallingEdge	Rise to Fall
	DCM_FallingEdge	DCM_RisingEdge	Fall to Rise
	DCM_BothEdge	DCM_ComplementEdge	Fall to Rise and Rise to Fall
	DCM_BothEdges	DCM_BothEdges	Both to Both

Propagation	Starting enumeration/ Reference enumeration	Ending enumeration/ Signal enumeration	Meaning
 X	DCM_RisingEdge	DCM_Terminate	Rise to Terminate
 X	DCM_FallingEdge	DCM_Terminate	Fall to Terminate
 X  X	DCM_RisingEdge DCM_FallingEdge	DCM_Terminate DCM_Terminate	Rise to Terminate and Fall to Terminate
 one to Z	DCM_RisingEdge	DCM_OneToZ	Rise to One_To_Z
 Zero to Z	DCM_FallingEdge	DCM_ZeroToZ	Fall to Zero_To_Z
 Zero to Z	DCM_RisingEdge	DCM_ZeroToZ	Rise to Zero_To_Z
 One to Z	DCM_FallingEdge	DCM_OneToZ	Fall to One_To_Z
Z to One 	DCM_ZtoOne	DCM_RisingEdge	Z_to_One to Rise
Z to One 	DCM_ZtoOne	DCM_FallingEdge	Z_to_One to Fall
Z to Zero 	DCM_ZtoZero	DCM_RisingEdge	Z_to_Zero to Rise
Z to Zero 	DCM_ZtoZero	DCM_FallingEdge	Z_to_Zero to Fall

Propagation	Starting enumeration/ Reference enumeration	Ending enumeration/ Signal enumeration	Meaning
	DCM_BothEdges	DCM_RisingEdge	Both to Rise
	DCM_RisingEdge	DCM_BothEdges	Rise to Both
	DCM_FallingEdge	DCM_BothEdges	Fall to Both
Term	DCM_Terminate	DCM_RisingEdge	Terminate to Rise
Term	DCM_Terminate	DCM_FallingEdge	Terminate to Fall
Term	DCM_Terminate	DCM_BothEdges	Terminate to Both
Term	DCM_BothEdges	DCM_Terminate	Both to Terminate
Term	DCM_RisingEdge	DCM_Terminate	Rise to Terminate
Term	DCM_FallingEdge	DCM_Terminate	Fall to Terminate

10.27.5 Edge propagation communication to the DPCM

Although the DPCM passes edge enumerations that represent more than one edge pair, the application shall limit itself to single edge combinations. The application shall break up complex edges (such as *BOTH*) into component parts. Table 446 is a representative sampling of the enumeration pairs the are allowed.

Table 446—Edge propagation communication with DPCM

Edge pair	Source edge/ reference edge	Sink edge/ signal edge
Rising Input Rising Output	DCM_RisingEdge	DCM_RisingEdge
Rising Input Falling Output	DCM_RisingEdge	DCM_FallingEdge
Falling Input Falling Output	DCM_FallingEdge	DCM_FallingEdge
Falling Input Rising Output	DCM_FallingEdge	DCM_RisingEdge
Falling Input ZeroToZ Output	DCM_FallingEdge	DCM_ZeroToZ
Falling Input OneToZ Output	DCM_FallingEdge	DCM_OneToZ
Rising Input ZeroToZ Output	DCM_RisingEdge	DCM_ZeroToZ
Rising Input OneToZ Output	DCM_RisingEdge	DCM_OneToZ
Falling Input ZtoZero Output	DCM_FallingEdge	DCM_ZtoZero
Rising Input ZtoZero Output	DCM_RisingEdge	DCM_ZtoZero
Falling Input ZtoOne Output	DCM_FallingEdge	DCM_ZtoOne
Rising Input ZtoOne Output	DCM_RisingEdge	DCM_ZtoOne

For edge propagation enumeration passed into the DPCM, the application shall use the exact edge (same and complementEdge(s) are not supported).

10.27.6 newTimingPin

Table 447 provides information on newTimingPin.

Table 447—newTimingPin

Function name	newTimingPin
Arguments	Standard Structure pointer, Pointer to node name
Result	Pointer to application's node structure
Standard Structure fields	pathData (timing)
C syntax	int newTimingPin (DCM_STD_STRUCT *std_struct, DCM_HANDLE *appstruct, char *nodename);

The DPCM calls *newTimingPin* to create an internal node for the cell being modeled for timing. A pointer to the name of the internal node (*nodename*) is passed to the application.

The application returns a pointer to its internal pin structure (*appstruct*) whose first field shall be a char * that points to a string representing the pin name.

Names for internal cell nodes are unique only within the life span of a single call to a modelSearch (see 10.27.1).

- If the *pathData* field in the Standard Structure has a value other than zero (0), then the application shall record that value and associate it with this internal node for subsequent use.

Multiple calls of this function for the same node shall constitute an error.

10.27.7 newDelayMatrixRow

Table 448 provides information on newDelayMatrixRow.

Table 448—newDelayMatrixRow

Function name	newDelayMatrixRow
Arguments	Standard Structure pointer, Signal edge type for beginning edge, Propagation mode for beginning edge, Signal edge type for ending edge, Propagation mode for ending edge
Result	Pointer to application structure for the delay matrix
Standard Structure fields	None
C syntax	<pre>int newDelayMatrixRow (DCM_STD_STRUCT *std_struct, DCM_HANDLE *delayMatrix, DCM_EdgeTypes edge1, DCM_PropagationTypes mode1, DCM_EdgeTypes edge2, DCM_PropagationTypes mode2);</pre>

The DPCM calls *newDelayMatrixRow* to describe how one of the edges of the signal shall propagate across the timing arc. A timing arc may propagate multiple edges. *newDelayMatrixRow* is called once for each edge.

Initially, *delayMatrix* is 0; it signals to the application that it needs to create the first delay matrix row. The application creates its appropriate structure and returns the pointer to the DPCM. Subsequent calls to *newDelayMatrixRow* shall reuse this pointer, which allows the application to add propagation information about the arc.

The resulting collection of propagation information is supplied as the *delayMatrix* parameter to the *newNetSinkPropagateSegments*, *newNetSourcePropagateSegments*, or *newPropagateSegments* functions. The same delay matrix may be reused for any number of arcs created by the same DCL modeling statement.

Example

Consider the following DCL *PATH* statement:
 PATH(*): FROM(A) TO(Y)
 PROPAGATE(RISE->RISE & FALL<-FALL)....;

This example results in the following:
 DCM_Handle dpcmDelayHandle = 0; ...
 newDelayMatrixRow(const std_struct,
 &dpcmDelayHandle,
 DCM_RisingEdge,
 DCM_EarlyMode,
 DCM_SameEdge,
 DCM_SameMode); ...
 newDelayMatrixRow(const std_struct,
 &dpcmDelayHandle,
 DCM_FallingEdge,
 DCM_LateMode,
 DCM_SameEdge,
 DCM_SameMode); ...

10.27.8 newNetSinkPropagateSegments

Table 449 provides information on *newNetSinkPropagateSegments*.

Table 449—newNetSinkPropagateSegments

Function name	newNetSinkPropagateSegments
Arguments	Standard Structure pointer, Node (sink) pin pointer, Import pin pointer, Delay matrix (created by newDelayMatrixRow)
Result	None
Standard Structure fields	pathData (power or timing pin-specific)
C syntax	<pre>int newNetSinkPropagateSegments (DCM_STD_STRUCT *std_struct, DCM_HANDLE importPin, DCM_HANDLE sinkPin, DCM_HANDLE delayMatrix);</pre>

The DPCM calls *newNetSinkPropagateSegments* to request the application find all sources on the net to which the passed import pin is connected and build propagation arcs from these sources to the passed node (sink) pin. That sink pin may or may not be a part of the physical interconnect to which the source pin is connected. Arcs are to be generated from every source pin *except* the import pin argument. The propagation characteristics are described by the delay matrix created by previous *newDelayMatrixRow* calls. The clkflg value is made available to the application at pathData->pcdb->clkflg.

Calls to this function result from one of two different *MODELPROC* functions, as follows:

- An *INPUT* function generates calls where the node (sink) pin and the import pin arguments are the same pointer.
- The *NODE* clause (in a *DO* function) generates calls with the import pin set to the pin designated by the associated *IMPORT* clause and the node (sink) pin set to the new *NODE* being defined.
- When the *sinkPin* and *importPin* arguments identify the same pin, the *pathData* value supplied in the *Standard Structure* shall be associated with this pin. If this *pathData* value is nonzero, the application shall record it for subsequent use. If, under these circumstances, the *delayMatrix* argument has a value of zero (0), the application shall perform no other actions in response to this call. This can result from an *INPUT* statement without a propagation sequence.
- When the *sinkPin* and *importPin* arguments are different, the *pathData* field in the *Standard Structure* shall be associated with the node identified by *sinkPin*. Under these circumstances, the *delayMatrix* argument shall have a nonzero value. The node so identified shall have already been created via a call by the library to *newTimingPin()* (see 10.27.6). If a *pathData* value for the node has been recorded by the application in response to that call and a nonzero *pathData* value is supplied with this call, then these two values shall be the same. If no *pathData* value for the node has been recorded previously and a nonzero value is supplied with this call, then the application shall record this value. If a *pathData* value of zero (0) is supplied with this call, then it shall be ignored by the application.
- It shall be an error for the library to provide different *pathData* values in multiple calls to this function made for the same pin or node.

NOTE—This situation can arise, for example, in ECL and other bipolar technologies where wired outputs are allowed to alter the state of a storage element.

Both of these are demonstrated in Figure 19 and Figure 20.

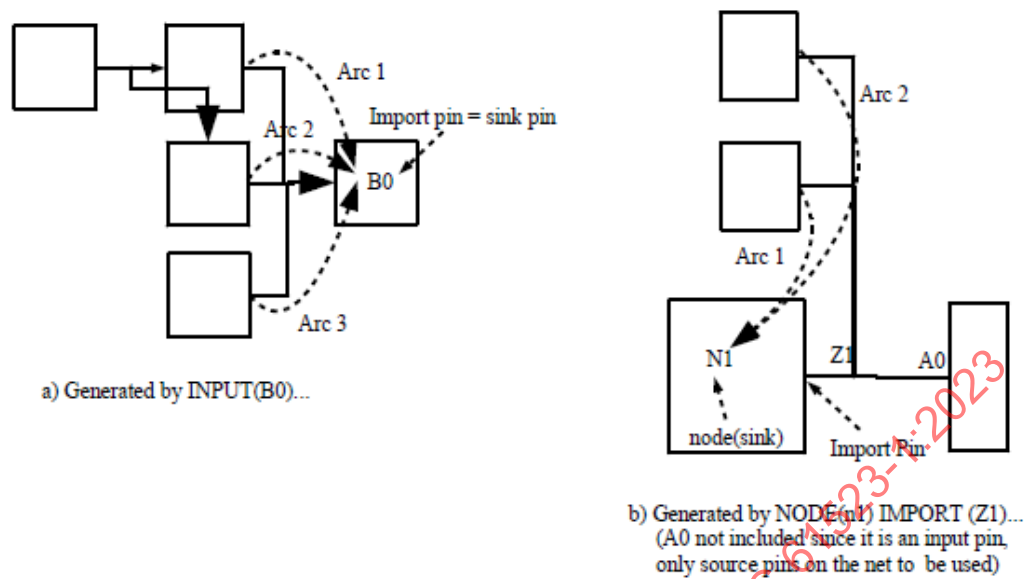


Figure 19—Sample MODELPROC results

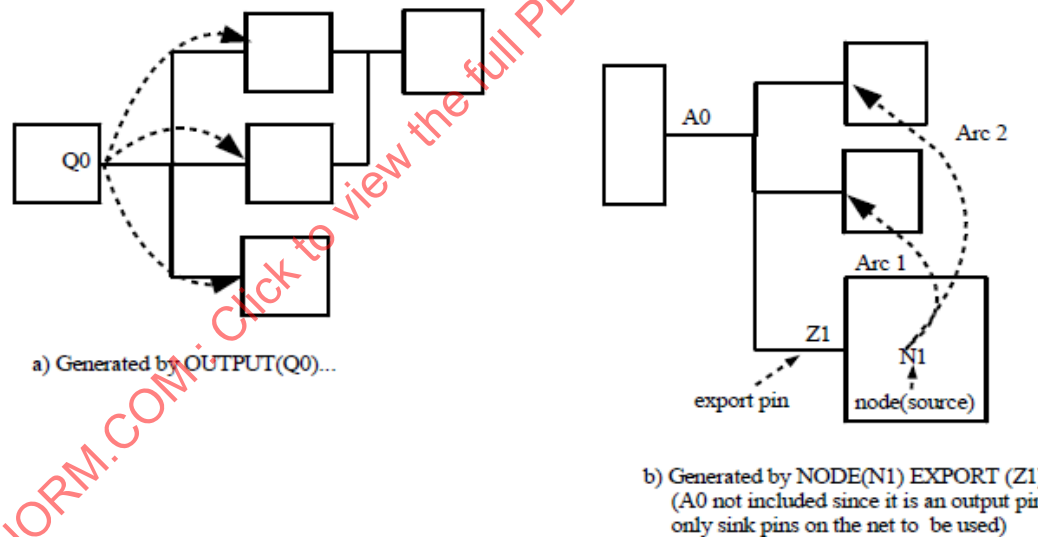


Figure 20—Additional MODELPROC results

10.27.9 newNetSourcePropagateSegments

Table 450 provides information on newNetSourcePropagateSegments.

Table 450—newNetSourcePropagateSegments

Function name	newNetSourcePropagateSegments
Arguments	Standard Structure pointer, node (source) pin pointer, export pin pointer, Delay matrix (created by newDelayMatrixRow)
Result	None
Standard Structure fields	pathData (power or timing pin-specific)
C syntax	<pre>int newNetSourcePropagateSegments (DCM_STD_STRUCT *std_struct, DCM_HANDLE sourcePin, DCM_HANDLE exportPin, DCM_HANDLE delayMatrix);</pre>

The DPCM calls *newNetSourcePropagateSegments* to request the application find all sinks on the net where the passed export pin is connected and build propagation arcs to these sinks from the passed node (source) pin. That source pin may or may not be a part of the physical interconnect to which the export pin is connected. Arcs are to be generated to every sink pin *except* the export pin argument. The propagation characteristics are described by the delay matrix created by previous *newDelayMatrixRow* calls.

Calls to this function result from one of two different *MODELPROC* functions, as follows:

- An *OUTPUT* function generates calls where the export pin and the node (source) pin arguments are the same pointer.
- The *NODE* clause (in a *DO* function) generates calls with the export pin set to the pin designated by the associated *EXPORT* clause and the node (source) pin set to the new *NODE* being defined.
- When the *sourcePin* and *exportPin* arguments identify the same pin, the *pathData* value supplied in the *Standard Structure* shall be associated with this pin. If this *pathData* value is nonzero, then the application shall record it for subsequent use. If, under these circumstances, the *delayMatrix* argument has a value of zero (0), the application shall perform no other actions in response to this call. This can result from an *OUTPUT* statement without a propagation sequence.
- When the *sourcePin* and *exportPin* arguments are different, the *pathData* field in the *Standard Structure* shall be associated with the node identified by *sourcePin*. Under these circumstances, the *delayMatrix* argument shall have a nonzero value. The node so identified shall have already been created via a call by the library to *newTimingPin()* (see 10.27.6). If a *pathData* value for the node has been recorded by the application in response to that call and a non-zero *pathData* value is supplied with this call, then these two values shall be the same. If no *pathData* value for the node has been recorded previously and a nonzero value is supplied with this call, then the application shall record this value. If a *pathData* value of zero (0) is supplied with this call, it shall be ignored by the application.
- It shall be an error for the library to provide different *pathData* values in multiple calls to this function made for the same pin or node.

10.27.10 newPropagateSegment

Table 451 provides information on newPropagateSegment.

Table 451—newPropagateSegment

Function name	newPropagateSegment
Arguments	Standard Structure pointer, Driver (source) pin, Receiver (sink) pin, Delay matrix (created by newDelayMatrixRow)
Result	New timing arc handle pointer
Standard Structure fields	pathData (timing-arc-specific)
C syntax	<pre>int newPropagateSegment (DCM_STD_STRUCT *std_struct, DCM_HANDLE *output, DCM_HANDLE sourcePin, DCM_HANDLE sinkPin, DCM_HANDLE delayMatrix);</pre>

DPCM calls this function to connect a timing arc between the two specified points in the model, the driver (source) pin and receiver (sink) pin. The propagation characteristics are described within the delay matrix created by previous *newDelayMatrixRow* function calls.

The application returns a pointer to the newly created timing arc through the function's *output* parameter.

The *PATH* function generates a call to *newPropagateSegment* for each segment modeled.

NOTE—DPCM expects the application to save the *PATH_DATA* field in the Standard Structure for later use when delay and slew calculations are desired for this timing arc.

10.27.11 newTestMatrixRow

Table 452 provides information on newTestMatrixRow.

Table 452—newTestMatrixRow

Function name	newTestMatrixRow
Arguments	Standard Structure pointer, Signal edge type for beginning edge, Propagation mode for beginning edge, Signal edge type for ending edge, Propagation mode for ending edge, Test type
Result	Pointer to the application's test matrix structure
Standard Structure fields	None
C syntax	<pre>int newTestMatrixRow (DCM_STD_STRUCT *std_struct, DCM_HANDLE *testMatrix, DCM_EdgeTypes edge1, DCM_PropagationTypes model1, DCM_EdgeTypes edge2, DCM_PropagationTypes mode2, DCM_TestTypes testType);</pre>

DPCM calls this function to describe the propagation characteristics for timing arcs created by DCL *TEST* statements. Initially, *testMatrix* is 0, indicating a test matrix row needs to be created. The application shall create its appropriate structure and return a pointer to that structure to the DPCM (**testMatrix*). Subsequent calls to this function from the DPCM shall reuse this pointer, which allows the application to add test information.

The resulting collection of test information is supplied as the *testMatrix* parameter to the *newAltTestSegment* function. The same test matrix may be reused for any number of test arcs created by the same DCL modeling statement.

The values of testType are enumerated in Table 7.

10.27.12 newAltTestSegment

Table 453 provides information on newAltTestSegment.

Table 453—newAltTestSegment

Function name	newAltTestSegment
Arguments	Standard Structure pointer, From pin, To pin, Test matrix (created by newTestMatrixRow)
Result	Pointer to the test arc handle
Standard Structure fields	pathData (timing-arc-specific)
C syntax	<pre>int newAltTestSegment (DCM_STD_STRUCT *std_struct, DCM_HANDLE *output, DCM_HANDLE clock, DCM_HANDLE data, DCM_HANDLE testMatrix);</pre>

The DPCM calls this function when a specific test arc is to be created between the two specified pins. The From pin is described by the clock parameter and the To pin is described by the data parameter. The propagation characteristics are described by the test matrix, which was created by previous *newTestMatrix* calls.

The *TEST* function generates a call to *newAltTestSegment* for each segment tested.

NOTE—The DPCM expects the application to save the pathData field in the Standard Structure for later use when test calculations are desired for this arc.

10.27.13 Interactions between interconnect modeling and modeling functions

The two strategies for modeling interconnect in DCL are as follows:

- Use OUTPUT and/or INPUT statements in the model.
- Use the DEFAULT delay and slew calculation statement.

The application shall determine whether the DPCM used either an *INPUT* or *OUTPUT* function with a propagation sequence by examining the passed argument named delayMatrix from callback functions *newNetSourcePropagationSegment* and *newNetSinkPropagationSegment*. When the argument *delayMatrix* has a value of zero (0), it shall mean NO propagation sequence was present with the associated *INPUT* or *OUTPUT* function.

An application shall perform the following in order to handle interconnect calculations correctly:

- At *modelSearch* time (when model structure is conveyed to the application by the DPCM):
The application shall remember the *pathData* pointer for any *OUTPUT* or *INPUT* functions in the model. Conceptually, the application shall associate such *pathData* pointer values with the appropriate nodes in the design.
- At interconnect delay/slew calculation time:
For situations when nonzero values of the argument *delayMatrix* to the call *newNetSourcePropagationSegment* are received, the application shall use the *pathData* pointer associated with this call when calling for delays and slews on those nets. For situations when non-

zero values for the argument *delayMatrix* to the call *newNetSinkPropagationSegment* are received, the application shall use the *pathData* pointer associated with this call when calling for delays and slews on those nets. For situations where only zero (0) values for *delayMatrix* are received for callbacks associated with both ends of the net, the application shall use the *pathData* pointer associated with the *newNetSourcePropagationSegment* if called; otherwise, the value of zero (0) shall be used for the *pathData* pointer.

- For situations where no callbacks were made on pins associated with the net, the application shall use a value of zero (0) for the *pathData* pointer when calling for delays and slews. The application shall further assume the net propagation properties are a rise edge at the source causes a rise at the sink and a fall edge at the source causes a fall at the sink.
- The application shall help to ensure all the fields required by the call to delay and slew are filled, and for interconnect calculation, the application shall also help to ensure that fields such as *cell*, *cellQual*, and *modelDomain* represent the cell driving the net.

Calculation functions are not called explicitly by name, but are accessed with pointers supplied in DCMTransmittedInfo as a result of the first call to the dcmRT_BindRule (see 10.25.4.2).

10.28 Deprecated functions

The function definitions contained in this subclause are here for compatibility with older versions of this specification. The use of these functions for new applications and library is strongly discouraged. These functions have a high probability of being removed from future revisions of this standard.

10.28.1 Parasitic handling

The following functions were for processing RLC networks. These functions have newer replacements.

10.28.1.1 appGetPiModel

Table 454 provides information on appGetPiModel.

Table 454—appGetPiModel

Function name	appGetPiModel
Arguments	Pin pointer
Result	estFlag, Capacitance value (nearest the driver), Capacitance value (nearest the load), Resistance value
Standard Structure fields	None
DCL syntax	EXTERNAL (appGetPiModel) : passed(pin: outputPin) result(int: estFlag & double: capNear, capFar, Resistance) ;
C syntax	typedef struct {INTEGER estFlag ; DCM_DOUBLE capNear, capFar, Resistance} T_capResValue; int appGetPiModel (DCM_STD_STRUCT *std_struct, T_capResValue *rtn, DCM_PIN outputPin);

This returns the capacitance and resistance values for the π model of the interconnect to which the passed pin is connected. This is meant for use in computation of the load-dependent delay portion of an interconnect delay.

capNear represents the capacitance value nearest the *sourcePin* and *capFar* represents the capacitance value farthest from the *sourcePin*. An example is shown in Figure 21.

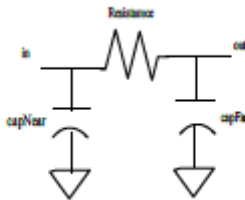


Figure 21—Capacitance value example

estFlag indicates whether the π model's values were computed accurately or merely estimated. It is valid for an application, which cannot compute (or otherwise access) π values, to return an approximation of those values; in which case, *estFlag* shall be set to a nonzero value. Because calculation of *C-effective* from this model may be expensive, if the input values are merely an approximation or estimate, the DPCM may be coded to approximate its calculation for *C-effective*. An *estFlag* value of 0 indicates that the π model is accurate.

In the case of an error, or if the application cannot answer the request at all, it shall set the function return code to a nonzero value in accordance with the rules described in 10.10.1. In this error situation, the DPCM (or DCL subrule) is expected to use the appropriate default model.

10.28.1.2 appGetPolesAndResidues

Table 455 provides information on *appGetPolesAndResidues*.

Table 455—*appGetPolesAndResidues*

Function name	<i>appGetPolesAndResidues</i>
Arguments	Driver (source) pin pointer, Receiver (sink) pin pointer, Order number
Result	An array of real and imaginary poles and an array of real and imaginary residues
Standard Structure fields	None
DCL syntax	<pre>EXTERNAL (appGetPolesAndResidues) : passed (pin: sourcePin, receiverPin; int: orderNum) result (double[*]: rel_poles, img_poles, rel_residues, img_residues);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE_ARRAY *rel_poles, *img_poles, *rel_residues, *img_residues; } T_polesResid; int appGetPolesAndResidues (DCM_STD_STRUCT *std_struct, T_polesResid *rtn, DCM_PIN sourcePin, DCM_PIN receiverPin, INTEGER orderNum);</pre>

This returns the step response poles and residues for the transient response function of the receiver (sink) pin as seen by the passed driver (source) pin. The transient response of the sink can then be expressed as shown in Figure 22.

$$v_{step}(t) = v_{dc} + \sum_{i=1}^q k_i e^{P_i * t}$$

NOTE— q is the number of poles, k_i are the residues, and P_i are the poles (rad/s).

Figure 22—Equation for poles and residues

The *orderNum* argument is the maximum number of pole/residue pairs to be returned.

The real and imaginary components of the poles and the residues are returned in the resulting arguments *rel_poles*, *img_poles* and *rel_residues*, *img_residues*.

NOTE—If the application cannot compute or access the requested pole-residue information, it may call *dpcmCalcPolesAndResidues* to request these values. The DPCM may then (dependent on the library code) call *appGetRLCnetworkByPin* to get the complete RLC configuration of the network (which it can reduce to an equivalent circuit and compute the poles and residues).

10.28.1.3 appGetCeffective

Table 456 provides information on *appGetCeffective*.

Table 456—appGetCeffective

Function name	<i>appGetCeffective</i>
Arguments	delay calculation function pointer, slew calculation function pointer, PI model
Result	late Ceffective, early Ceffective
Standard Structure fields	None
DCL syntax	<pre> forward_calc(stdDelaySlewEq): passed(double: loadCap, inputTransition) result(double); forward_calc(getPiModel): passed(pin: outputPin) result(int: estFlag ; double:nearCap, farCap, resistance); EXTERNAL(appGetCeffective): passed(stdDelaySlewEq(): delayEq; stdDelaySlewEq(): slewEq; getPiModel: piModel) result(double: lateCeфф, earlyCeфф); </pre>

C syntax	<pre> typedef struct { INTEGER estFlag; DCM_DOUBLE nearCap, farCap, resistance; } getPIModelStruct; typedef struct { DCM_DOUBLE lateCeffective, earlyCeffective; } T_Ceffective; int appGetCeffective (DCM_STD_STRUCT *std_struct, T_Ceffective *rtn, DCM_GeneralFunction delayEq, DCM_GeneralFunction slewEq, getPIModelStruct *piModel); </pre>

This returns an equivalent value for the effective capacitance (*C-effective*) seen by the passed driver (source) pin. *Effective capacitance* is an equivalent capacitance value (as seen by the driving circuit) that correctly models the slew and delay values taking into account resistance and capacitance on the interconnect. Because of the resistance within the network, the driving output may reach a sufficient voltage to have switched before the driven interconnect reaches this voltage. *Effective capacitance* is the value of a lumped capacitance that alone (with no resistance counterpart) would cause the cell to have taken the same amount of time to switch. This allows the library developer to model the library with simple capacitors. This function passes through the application the necessary parameters for the calculation of *C-effective*. The application shall pass these parameters to *dcpmCalcCeffective* without altering them.

10.28.1.4 appGetRLCnetworkByPin

Table 457 provides information on AppGetRLCnetworkByPin.

Table 457—appGetRLCnetworkByPin

Function name	appGetRLCnetworkByPin
Arguments	Pin pointer, RLC network handle
Result	newRLCnetwork handle
Standard Structure fields	None
DCL syntax	<pre> EXTERNAL (appGetRLCnetworkByPin) : passed(pin: inputPin, void:RLCnetwork) result(void: newRLCnetwork); </pre>
C syntax	<pre> typedef struct { VOID newRLCnetwork; } T_RLCnetworkByPin; int appGetRLCnetworkByPin (DCM_STD_STRUCT *std_struct, T_RLCnetworkByPin *rtn, DCM_PIN inputPin, VOID RLCnetwork); </pre>

Returns the *RLCnetwork* handle for the interconnect specified by the *PASSED* pin pointer argument. If the *PASSED RLCnetwork* argument is 0, then an existing *RLCnetwork* handle may be returned. If an appropriate *RLCnetwork* does not yet exist, then the application is expected to request the DPCM to build one.

If the *PASSED RLCnetwork* argument is nonzero, it signifies a nested call as a result of an *dcpmAppendPinAdmittance* call (on a pass-through device, for example) while the application was constructing an *RLCnetwork* from an earlier *appGetRLCnetworkByPin* call.

10.28.1.5 appGetRLCnetworkByName

Table 458 provides information on appGetRLCnetworkByName.

Table 458—appGetRLCnetworkByName

Function name	appGetRLCnetworkByName
Arguments	Pin name, RLC network handle
Result	Address of the structure that contains the RLC network in the DPCM (see pcmSetRLCmember 10.28.1.9)
Standard Structure fields	block
DCL syntax	EXTERNAL (appGetRLCnetworkByName) : passed(string: pinName, void:RLCnetwork) result(void:newRLCnetwork);
C syntax	typedef struct { VOID newRLCnetwork; } T_RLCnetworkByName; int appGetRLCnetworkByName (DCM_STD_STRUCT *std_struct, T_RLCnetworkByName *rtn, STRING pinName, VOID RLCnetwork);

This returns the *RLCnetwork* handle for the interconnect specified by the *PASSED* pin name argument. The behavior and semantics of this function are the same as those for the *appGetRLCnetworkByPin* in Table 454.

10.28.1.6 dpcmCalcPiModel

Table 459 provides information on dpcmCalcPiModel.

Table 459—dpcmCalcPiModel

Function name	dpcmCalcPiModel
Arguments	Driver (source), pin pointer, RLC network pointer
Result	estFlag, Capacitance value (nearest the driver), Capacitance value (nearest the load), Resistance value
Standard Structure fields	block, CellName, pathData (timing-arc-specific), cellData (timing)
DCL syntax	(dpcmCalcPiModel) : passed(pin: sourcePin; void: RLCnetwork) result(int: estFlag; double: capNear, capFar, Resistance);
C syntax	typedef struct { INTEGER estFlag; DCM_DOUBLE capNear, capFar, Resistance; } T_calcCapRes; int dpcmCalcPiModel (const DCM_STD_STRUCT *std_struct, T_calcCapRes *rtn, DCM_PIN sourcePin, VOID RLCnetwork);

This requests the DPCM to compute the capacitance and resistance values for the π model of the interconnect to which the passed driver (source) pin is connected, for use in computation of the load dependent delay or slew portion of an arc. A zero value for *RLCnetwork* passed by the application indicates this call shall create a new *RLC network* rather than reuse one generated by another function call.

capNear represents the capacitance value nearest the *sourcePin* and *capFar* represents the capacitance value farthest from the *sourcePin*. For an example, see Figure 22.

See 10.27.13 for details about the interaction between the DPCM and the application during the calculation of π values.

estFlag indicates whether or not the π model's values were computed accurately or merely estimated. It is valid for the DPCM, which cannot compute (or otherwise access) π values, to return an approximation of those values; in which case, *estFlag* shall be set to a nonzero value.

10.28.1.7 dpcmCalcPolesAndResidues

Table 460 provides information on dpcmCalcPolesAndResidueus.

Table 460—dpcmCalcPolesAndResidues

Function name	dpcmCalcPolesAndResidues
Arguments	Driver (source) pin pointer, Receiver (sink) pin pointer, Order number, RLC network pointer
Result	An array of real and imaginary poles and an array of real and imaginary residues
Standard Structure fields	CellName, block, pathData, (timing-pin-specific), cellData (timing)
DCL syntax	<pre>EXPOSE(dpcmCalcPolesAndResidues): passed(pin: sourcePin, loadPin; int: orderNum; void:RLCnetwork) result(double[*]: rel_poles, img_poles, rel_residues, img_residues);</pre>
C syntax	<pre>typedef struct { DCM_DOUBLE_ARRAY *rel_poles, *img_poles; DCM_DOUBLE_ARRAY *rel_residues, *img_residues; } T_calcPolesRes; int dpcmCalcPolesAndResidues (const DCM_STD_STRUCT *std_struct, T_calcPolesRes *rtn, DCM_PIN sourcePin, DCM_PIN sinkPin, INTEGER orderNum, VOID RLCnetwork);]</pre>

This requests the DPCM to compute the step response poles and residues for the transient response function of the receiver (sink) pin specified by driver (source) pin as seen by the driver (source) pin. For the relevant formula, see Figure 22.

The *orderNum* argument is the maximum number of pole/residue pairs to be returned.

The real and imaginary components of the poles and the residues are returned in the result arguments *rel_poles*, *img_poles* and *rel_residues*, *img_residues*. Each of these resulting arguments is a pointer to an array of floats containing the real and imaginary components.

When the application calls this function, a zero value for the *RLCnetwork* indicates that this function shall call back the application to build another RLC network.

NOTE—The DPCM may then (dependent on the library code) call back the application, through appGetRLCnetworkByPin to get the complete RLC configuration of the network (which it can reduce to an equivalent circuit and compute the poles and residues).

10.28.1.8 dpcmCalcCeffective

Table 461 provides information on dpcmCalcCeffective.

Table 461—dpcmCalcCeffective

Function name	dpcmCalcCeffective
Arguments	delay calculation function pointer, slew calculation function pointer, PI model
Result	late Ceffective, early Ceffective
Standard Structure fields	CellName, block, sourceEdge, sinkEdge, pathData (timing-arc-specific), cellData (timing), toPoint, fromPoint, earlySlew, lateSlew
DCL syntax	<pre>forward calc(stdDelaySlewEq): passed(double: loadCap, inputTransition) result(double); forward calc(getPiModel): passed(pin: outputPin) result(int: estFlag; double:capNear, capFar, Resistance); EXPOSE(dpcmCalcCeffective): passed(stdDelaySlewEq(): delayEq; stdDelaySlewEq(): slewEq; getPiModel: piModel) result(double: lateCeff, earlyCeff);</pre>
C syntax	<pre>typedef struct { INTEGER estFlag; DCM_DOUBLE capNear, capFar, resistance; } getPiModelStruct; typedef struct { DCM_DOUBLE lateCeffective, earlyCeffective; } T_Ceffective; int dpcmCalcCeffective (const DCM_STD_STRUCT *std_struct, T_Ceffective *rtn, DCM_GeneralFunction DelayEq, DCM_GeneralFunction slewEq, getPiModelStruct *piModel);</pre>

This returns an equivalent “effective” capacitance as seen by the passed toPoint.

10.28.1.9 dpcmSetRLCmember

Table 462 provides information on dpcmSetRLCmember.

Table 462—dpcmSetRLCmember

Function name	dpcmSetRLCmember
Arguments	RLC network to add this member to, Driver (source) pin pointer, Receiver (sink) pin pointer, Element type (R, L, C, or M), Element name, Element value (capacitance, resistance, inductance or mutual inductance), Terminal 1 type, Terminal 2 type
Result	newRLCnetwork
Standard Structure fields	None

DCL syntax	<pre> EXPOSE(dpcmSetRLCmember): passed(void: RLCnetwork; pin: terminal1, terminal2; string: elementType, elementName; double: elementValue; int: terminal1Type, terminal2Type) result(void: newRLCnetwork); </pre>
C syntax	<pre> typedef struct { VOID newRLCnetwork; } T_SetRLCmember; int dpcmSetRLCmember (const DCM_STD_STRUCT *std_struct, T_SetRLCmember *rtn, VOID RLCnetwork, DCM_PIN terminal1, DCM_PIN terminal2, STRING elementType, STRING elementName, DCM_DOUBLE elementValue, INTEGER terminal1Type, INTEGER terminal2Type); </pre>

This sends an *R*, *L*, *C*, or *M* element value for an arc of the interconnect identified from the last call of *appGetRLCnetworkByPin* or *appGetRLCnetworkByName*.

The *elementType* field is set to *R* if the value is for a resistor, *L* if the value is an inductor, *C* if the value is for a capacitor, and *M* if the value is mutual inductance. A terminal type of 1 indicates a port and terminal type of 0 indicates an internal node of the interconnect network.

If the *RLCnetwork PASSED* parameter is the handle 0, the DPCM shall create a new RLC network. If the *RLCnetwork PASSED* parameter is nonzero, the DPCM shall assume it is a handle to an RLC network created previously by the DPCM. In either case, the DPCM shall add the *PASSED* member information to the network and return the current network handle in the *newRLCnetwork RESULT* parameter. The handle shall be a pointer to a structure whose first element is itself a pointer to a function with the same argument signature as *dpcmSetRLCmember*. This latter function shall be called for all additions to the RLC network.

Subsequent calls made by the application to add members to any RLC network shall always use the latest *newRLCnetwork* handle returned from the last call to *dpcmSetRLCmember* as the *PASSED RLCnetwork* argument.

The DPCM shall cache as many *RLCnetworks* as requested by an application until they are explicitly removed via *dpcmDeleteRLCnetwork*. See 10.28.1.11. If the *RLCnetwork PASSED* parameter is not a valid handle returned by a prior call to *dpcmSetRLCmember*, then the behavior is undefined. The passed driver (source) and receiver (sink) pin pointers may refer to instance pins or internal nodes on the network (note node1 and node2 in Figure 3).

If the *elementType* field is *R*, *L*, or *M* then both the driver (source) and receiver (sink) pins are required (resistance and inductance are always assumed between two nodes or ports). If *elementType* is *C* and is grounded, then both *terminal1* and *terminal2* shall be the non-grounded node.

In Figure 23, *Pin1* and *Pin2* are terminal type 1 indicating ports, all others are terminal type 0 indicating internal nodes.

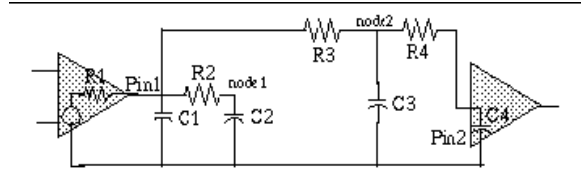


Figure 23—Example RC network

10.28.1.10 dpcmAppendPinAdmittance

Table 463 provides information on dpcmAppendPinAdmittance.

Table 463—dpcmAppendPinAdmittance

Function name	dpcmAppendPinAdmittance
Arguments	Sink pin handle, RLC network handle
Result	RLCnetwork handle
Standard Structure fields	block, CellName, pathData (timing-pin-specific), cellData (timing)
DCL syntax	<pre>EXPOSE(dpcmAppendPinAdmittance): passed(pin: sinkPin; void: RLCnetwork) result(void: newRLCnetwork);</pre>
C syntax	<pre>typedef struct { VOID newRLCnetwork; } T_AppendPinAdmittance; int dpcmAppendPinAdmittance (const DCM_STD_STRUCT *std_struct, T_AppendPinAdmittance *rtn, DCM_PIN inputPin, VOID RLCnetwork);</pre>

This function causes the DPCM to add the admittance for the specified pin to the specified *RLCnetwork*. If the *RLCnetwork PASSED* parameter is the handle 0, the DPCM shall create a new RLC network. If the *RLCnetwork PASSED* parameter is nonzero, the DPCM shall assume it is a handle to an RLC network created previously by the DPCM. In either case, the DPCM shall add the *PASSED* member information to the network and return the current network handle in the *newRLCnetwork RESULT* parameter. The handle shall be a pointer to a structure whose first element is itself a pointer to a function with the same argument signature as dpcmSetRLCmember. This latter function shall be called for all additions to the RLC network.

It is possible that the admittance for the specified pin is equivalent to the *RLCnetwork* for another pin, such as for the device output pin in the case of pass-through devices. In this case, the DPCM shall call *appGetRLCnetwork* for the pin specified in the original *dpcmAppendPinAdmittance* call.

10.28.1.11 dpcmDeleteRLCnetwork

Table 464 provides information on dpcmDeleteRLCnetwork.

Table 464—dpcmDeleteRLCnetwork

Function name	dpcmDeleteRLCnetwork
Arguments	RLC network handle
Result	return code
Standard Structure fields	None
DCL syntax	<pre>EXPOSE(dpcmDeleteRLCnetwork): passed(void: RLCnetwork) result(int: rc);</pre>
C syntax	<pre>typedef struct { INTEGER rc; } T_DeleteRLCnetwork; int dpcmDeleteRLCnetwork (const DCM_STD_STRUCT *std_struct, T_DeleteRLCnetwork *rtn, VOID RLCnetwork);</pre>

A DPCM is required to cache an *RLCnetwork* until the application calls *dpcmDeleteRLCnetwork* with that handle; however, the DPCM is not obligated to do anything (such as free any storage) as a result of such a call. The behavior of the DPCM as a result of subsequent use by the application of a deleted handle is undefined.

The application shall specify the same technology when calling *dpcmDeleteRLCnetwork* as was used when the network was created.

10.28.2 Array manipulation functions

These functions allow the application to manipulate array data that is returned by the DPCM.

10.28.2.1 dcm_copy_DCM_ARRAY

Table 465 provides information on *dcm_copy_DCM_ARRAY*.

Table 465—dcm_copy_DCM_ARRAY

Function name	dcm_copy_DCM_ARRAY
Arguments	DCM_ARRAY, DCM_AATTS
Result	DCM_ARRAY
Standard Structure fields	None
C syntax	<pre>DCM_ARRAY *dcm_copy_DCM_ARRAY (DCM_ARRAY *originalArray, DCM_AATTS attributes);</pre>

The application service *dcm_copy_DCM_ARRAY* allocates a new *DCM_ARRAY* and copies the contents of the original array into the newly allocated one. The *attributes* argument shall have the value 0xFF.

10.28.2.2 dcm_new_DCM_ARRAY

Table 466 provides information on *dcm_new_DCM_ARRAY*.

Table 466—dcm_new_DCM_ARRAY

Function name	dcm_new_DCM_ARRAY
Arguments	number of dimensions, vector of elements per dimension, size of each element,
Result	DCM_ARRAY
Standard Structure fields	None
C syntax	<pre>DCM_ARRAY *dcm_new_DCM_ARRAY (int numDims, int *elementsPer, int elementSize, DCM_ATYPE elementType, DCM_AATTS attributes, DCM_AINIT initialize, DCM_ArrayInitUserFunction initializer);</pre>

The application service *dcm_new_DCM_ARRAY* allocates a new array according to the number of dimensions, the number of elements in each dimension, and the size of each element. There are options to control the how an array is initialized. The maximum value for *numDims* is 255. When the system does not allocate the required space, an error is generated. A newly created array is locked once.

10.28.2.3 dcm_sizeof_DCM_ARRAY

Table 467 provides information on *dcm_sizeof_DCM_ARRAY*.

Table 467—dcm_sizeof_DCM_ARRAY

Function name	dcm_sizeof_DCM_ARRAY
Arguments	DCM_ARRAY
Result	Size of the DCM array
Standard Structure fields	None
C syntax	<pre>int dcm_sizeof_DCM_ARRAY(DCM_ARRAY *array);</pre>

The application service *dcm_sizeof_DCM_ARRAY* returns the number of bytes the *DCM_ARRAY*’s data elements consume. The application shall pass in the *DCM_ARRAY* pointer to be evaluated. If there is an error, a value of –1 is returned.

NOTE—Zero is a valid size for an empty array.

10.28.2.4 dcm_lock_DCM_ARRAY

Table 468 provides information on *dcm_lock_DCM_ARRAY*.

Table 468—dcm_lock_DCM_ARRAY

Function name	dcm_lock_DCM_ARRAY
Arguments	DCM_ARRAY
Result	Return code
Standard Structure fields	None
C syntax	<pre>int dcm_lock_DCM_ARRAY(DCM_ARRAY *array);</pre>

dcm_lock_DCM_ARRAY locks the array. The array shall persist until it is unlocked. The array may be locked multiple times by both the application and the DPCM.

If for any reason the system encounters an error, a nonzero value is returned; otherwise, a successful return value of zero is returned.

10.28.2.5 dcm_unlock_DCM_ARRAY

Table 469 provides information on `dcm_unlock_DCM_ARRAY`.

Table 469—`dcm_unlock_DCM_ARRAY`

Function name	<code>dcm_unlock_DCM_ARRAY</code>
Arguments	<code>DCM_ARRAY</code>
Result	Return code
Standard Structure fields	None
C syntax	<code>int dcm_unlock_DCM_ARRAY(DCM_ARRAY *array);</code>

`dcm_unlock_DCM_ARRAY` unlocks the array. The array shall be deleted when it has been unlocked as many times as it was locked. Neither the application nor the *DPCM* shall unlock the array more times than the application or the *DPCM* respectively locked it.

If for any reason the system encounters an error, a nonzero value is returned; otherwise, a successful return value of zero is returned.

10.28.3 Memory management

A *DCM_STRUCT* is a library-specific collection of data that includes memory management support. Memory management within the *DPCM* shall conform to the following behavior:

- When the *DPCM* creates a new array or structure, it keeps a *count* (initialized to 1) indicating how many references to that object exist. During the application's next call to the *DPCM* at a primary entry point, such as *modelSearch*, *delay*, *slew*, or *check*, the count is decremented; this is only done once per object during the first call to the *DPCM* after the object is allocated.
- When the application locks the object or the library creates a reference to the object, the count is incremented. When the application unlocks the object or the library removes a reference to the object, the count is decremented.
- The application is responsible for unlocking the object as many times as it has locked it. The library is responsible for removing references to the object it no longer needs.
- Once the reference count becomes 0, the memory is returned to the system.

10.28.3.1 dcm_lock_DCM_STRUCT

Table 470 provides information on `dcm_lock_DCM_STRUCT`.

Table 470—`dcm_lock_DCM_STRUCT`

C syntax	<code>int dcm_lock_DCM_STRUCT(DCM_STRUCT *dcmStruct);</code>

The application shall call `dcm_lock_DCM_STRUCT` (prior to calling the library again) to increment the reference count in a *DCM_STRUCT* (passed to it by the library).

10.28.3.2 dcm_unlock_DCM_STRUCT

Table 471 provides information on `dcm_unlock_DCM_STRUCT`.

Table 471—`dcm_unlock_DCM_STRUCT`

C syntax	<code>int dcm_unlock_DCM_STRUCT(DCM_STRUCT *dcmStruct);</code>

The application shall call `dcm_unlock_DCM_STRUCT` to decrement the reference count of a `DCM_STRUCT`. The application shall not call `dcm_unlock_DCM_STRUCT` more times than it called `dcm_lock_DCM_STRUCT` on any `DCM_STRUCT`.

10.28.3.3 dcm_getNumDimensions

Table 472 provides information on `dcm_getNumDimensions`.

Table 472—`dcm_getNumDimensions`

Function name	<code>dcm_getNumDimensions</code>
Arguments	<code>DCM_ARRAY</code>
Result	Number of dimensions
Standard Structure fields	None
C syntax	<code>int dcm_getNumDimensions(DCM_ARRAY *array);</code>

The application service `dcm_getNumDimensions` returns the number of dimensions defined for the `DCM_ARRAY` passed in by the application.

If for any reason there is an error in determining the number of dimensions, a value of `-1` is returned; otherwise, the number of dimensions is returned.

10.28.3.4 dcm_getNumElementsPer

Table 473 provides information on `dcm_getNumElementsPer`.

Table 473—`dcm_getNumElementsPer`

Function name	<code>dcm_getNumElementsPer</code>
Arguments	<code>DCM_ARRAY</code>
Result	Number of elements in each dimension
Standard Structure fields	None
C syntax	<code>int *dcm_getNumElementsPer(DCM_ARRAY *array, int *answer);</code>

The application service `dcm_getNumElementsPer` returns an array whose elements are the length of each dimension of the array argument.

The application shall supply a `DCM_ARRAY` and an integer array where the application service can place its results. `dcm_getNumElementsPer` places in each element of the answer array the number of elements in the corresponding `DCM_ARRAY`, where the zeroth index of the `DCM_ARRAY` corresponds to the zeroth element of the answer array. If the service detects an error, it returns `(int*) 0`; otherwise, it returns the answer.

10.28.3.5 dcm_getNumElements

Table 474 provides information on `dcm_getNumElements`.

Table 474—`dcm_getNumElements`

Function name	<code>dcm_getNumElements</code>
Arguments	<code>DCM_ARRAY</code>
Result	Number of dimensions
Standard Structure fields	None
C syntax	<code>int dcm_getNumElements(DCM_ARRAY *array, int dimension);</code>

The application service `dcm_getNumElements` returns the number of elements for the dimension specified. If an error is encountered, the value returned is `-1`. The dimension parameter passed in from the application shall be between 0 and the *number_of_dimensions* `-1`.

10.28.3.6 dcm_getElementType

Table 475 provides information on `dcm_getElementType`.

Table 475—`dcm_getElementType`

Function name	<code>dcm_getElementType</code>
Arguments	<code>DCM_ARRAY</code>
Result	Type of element
Standard Structure fields	None
C syntax	<code>DCM_ATYPE dcm_getElementType(DCM_ARRAY *array);</code>

The application service `dcm_getElementType` is passed a `DCM_ARRAY` and returns the type of elements stored.

If the application service detects an error, the element type `DCM_ATYPE_ERROR` is returned.

10.28.3.7 dcm_arraycmp

Table 476 provides information on `dcm_arraycmp`.

Table 476—`dcm_arraycmp`

Function name	<code>dcm_arraycmp</code>
Arguments	Two <code>DCM_ARRAY</code> s
Result	None
Standard Structure fields	None
C syntax	<code>int dcm_arraycmp(DCM_ARRAY *a1, DCM_ARRAY *a2);</code>

The application service `dcm_arraycmp` is passed to two `DCM_ARRAY`s and compares them for equality. If the two arrays contain (bit-by-bit) identical data, then the value of zero is returned; otherwise, a nonzero value is returned.

10.28.4 Initialization functions

Initialization functions are called by an application to load or unload a DPCM, or to set a universal storage manager or message handler. These functions are called as part of the process of preparing the system to accept a DPCM or to clean up after one has been terminated. They are available to the application because the dynamically loaded modules that make up a DPCM are not yet in memory and cannot perform these operations.

An application shall call *dcmSetNewStorageManager* to assert common storage management between the DPCM and the application.

10.28.4.1 dcmCellList

Table 477 provides information on *dcmCellList*.

Table 477—*dcmCellList*

Function name	<i>dcmCellList</i>
Arguments	Standard Structure pointer
Result	Array of cell names, Array of cell name qualifiers, Array of model domain
Standard Structure fields	None
C syntax	<pre>typedef struct dcm_T_dcmCellList { DCM_STRING_ARRAY *cellNameArray; DCM_STRING_ARRAY *cellQualArray; DCM_STRING_ARRAY *model_domainArray; } T_dcmCellList; int dcmCellList (const DCM_STD_STRUCT *std_struct, T_dcmCellList *rtn);</pre>

This returns three parallel arrays with cell names, cell name qualifiers, and model domains contained in the current DPCM. The cell name, cell name qualifier, and model domain fields at the same array index identify a cell modeled in this DPCM. If no cell name qualifier or model domain field is specified for a given *MODEL* in the DCL source, the corresponding array element contains an asterisk (*).

This function shall only be called from a DPCM. An application shall always call *dpcmGetCellList* to determine the *MODELS* in the DPCM.

10.28.4.2 dcmSetNewStorageManager

Table 478 provides information on *dcmSetNewStorageManager*.

Table 478—*dcmSetNewStorageManager*

Function name	<i>dcmSetNewStorageManager</i>
Arguments	malloc function pointer, free function pointer, realloc function pointer
Result	None
Standard Structure fields	None
C syntax	<pre>int dcmSetNewStorageManager (DCM_Malloc_Type malloc, DCM_Free_Type free, DCM_Realloc_Type realloc);</pre>

This sets the function pointers for memory allocation, free, and reallocation functions for the DPCM. The typedefs match the ISO C (ISO/IEC 9899:1990) *malloc()*, *free()* and *realloc()* functions, respectively.

dcmSetNewStorageManager may only be called once, and it shall be called before any DPCM is loaded or any application call to *dcm_new_DCM_ARRAY* or *dcm_new_DCM_STD_STRUCT*. If the call to this function is made after the call to *dcmBindRule*, it shall not perform the pointer instantiation and shall return a nonzero result. Zero as a return value indicates the pointer instantiation was successful.

Once *dcmSetNewStorageManager* has been called to designate memory management functions defined in the application, calls to *dcmMalloc*, *dcmFree*, or *dcmRealloc* result in the DPCM calling back the designated application functions.

10.28.4.3 dcmMalloc

Table 479 provides information on *dcmMalloc*.

Table 479—dcmMalloc

Function name	<i>dcmMalloc</i>
Arguments	Number of bytes to allocate
Result	None
Standard Structure fields	None
C syntax	<code>void *dcmMalloc(size_t numBytes);</code>

This returns a pointer to a block of memory at least *numBytes* long, using the storage management function currently in effect.

10.28.4.4 dcmFree

Table 480 provides information on *dcmFree*.

Table 480—dcmFree

Function name	<i>dcmFree</i>
Arguments	Pointer to memory block to be freed
Result	None
Standard Structure fields	None
C syntax	<code>void dcmFree(void *allocatedBlock);</code>

This frees memory allocated by *dcmMalloc* or *dcmRealloc*, using the storage management function currently in effect. The *allocatedBlock* shall first have been returned by *dcmMalloc* or *dcmRealloc*.

10.28.4.5 dcmRealloc

Table 481 provides information on *dcmRealloc*.

Table 481—dcmRealloc

Function name	<i>dcmRealloc</i>
Arguments	Number of bytes to reallocate
Result	None
Standard Structure fields	None
C syntax	<code>void *dcmRealloc(void *allocatedBlock, size_t numBytes);</code>

This returns a pointer to a block of memory at least *numBytes* long, using the storage management function currently in effect. This function call also copies the data from the allocated block to the newly reallocated space. The *allocatedBlock* shall first have been returned by *dcmMalloc* or *dcmRealloc*.

10.28.4.6 dcmBindRule

Table 482 provides information on *dcmBindRule*.

Table 482—dcmBindRule

Function name	<i>dcmBindRule</i>
Arguments	Rule name
Result	None
Standard Structure fields	None
C syntax	<code>void *dcmBindRule(const char *rootSubruleName);</code>

This loads and links the specified primary rule.

The passed argument is a pointer to a string containing the name of the primary (root) library rule to be loaded See 10.14.

dcmBindRule returns a pointer to the rule initialization entry point *dcm_rule_init* ee if the primary (root) library rule can be found and loaded. If the rule cannot be found or there was an error in loading, the pointer returned is zero. If this call is successful, then the application shall not call it again until a successful response from *dcmUnbindRule* occurs.

10.28.4.7 dcmAddRule

Table 483 provides information on *dcmAddRule*.

Table 483—dcmAddRule

Function name	<i>dcmAddRule</i>
Arguments	rule name
Result	return code
Standard Structure fields	None
C syntax	<code>void* dcmAddRule(const char subruleName, int *returnCode);</code>

This adds additional DCL subrules to the *DPCM* during execution. *dcmAddRule* does not alter subrules in the current *DPCM*. The passed parameter is a pointer to a string that contains the subrule name for the technology library to be added. See 10.14. *returnCode* is set to the integer return code for this function call (see 10.10.1).

This function may only be called following a successful call to *dcmBindRule* and preceding a successful call to *dcmUnbindRule*.

The *dcmAddRule* function returns a pointer to the rule initialization entry if the subrule is found and loaded. If the subrule cannot be found or a loading error occurs, the pointer returned is zero.

10.28.4.8 dcmUnbindRule

Table 484 provides information on *dcmUnbindRule*.

Table 484—dcmUnbindRule

Function name	dcmUnbindRule
Arguments	None
Result	None
Standard Structure fields	None
C syntax	<code>int dcmUnbindRule(void *initFunction);</code>

This unloads the DPCM from memory and releases any memory the DPCM may have consumed.

The passed argument is the void pointer returned from *dcmBindRule*. *dcmUnbindRule* returns an integer return code with a zero value when the function completes without error; otherwise, a nonzero value is returned.

10.28.4.9 dcmFindFunction

Table 485 provides information on dcmFindFunction.

Table 485—dcmFindFunction

Function name	dcmFindFunction
Arguments	EXPOSE function name, Function table
Result	None
Standard Structure fields	None
C syntax	<code>DCM_GeneralFunction dcmFindFunction (char *fcnName, DCM_FunctionTable exposes);</code>

This locates the passed *EXPOSE* function within the loaded DPCM and returns a pointer to the function.

The first passed argument is a pointer to the requested *EXPOSE* name (**fcnName*). The second passed argument (*exposes*) is the initialization table set up by the DPCM initialization function *dcm_rule_init*. See 10.28.4.16.

The result is a pointer to the *EXPOSE* function within the DPCM.

When the matching function cannot be found, an error message is issued and the returned pointer is zero.

10.28.4.10 dcmFindAppFunction

Table 486 provides information on dcmFindAppFunction.

Table 486—dcmFindAppFunction

Function name	dcmFindAppFunction
Arguments	EXTERNAL function name
Result	None
Standard Structure fields	None
C syntax	<code>int dcmFindAppFunction(char *fcnName);</code>

This determines whether the application defined the indicated *EXTERNAL* function. This function returns a nonzero value if the application did define the function; otherwise, a zero value is returned.

10.28.4.11 dcmQuietFindFunction

Table 487 provides information on dcmQuietFindFunction.

Table 487—dcmQuietFindFunction

Function name	dcmQuietFindFunction
Arguments	EXPOSE function name, Function table
Result	None
Standard Structure fields	None
C syntax	<pre>DCM_GeneralFunction dcmQuietFindFunction (char *fcnName, DCM_FunctionTable *exposes);</pre>

This function is the same as *dcmFindFunction*, except no error is issued if the function is not found (see 10.28.4.11).

10.28.4.12 dcmMakeRC

Table 488 provides information on dcmMakeRC.

Table 488—dcmMakeRC

Function name	dcmMakeRC
Arguments	Message number, Message severity, Error code address
Result	complete error code
Standard Structure fields	None
C syntax	<pre>int dcmMakeRC (int messageNumber, DCM_Message_Severities severity, int *errorCode);</pre>

This returns an error code constructed from the message number and severity arguments, which does not conflict with internal DCL reserved codes (such as those returned from *dcmHardErrorRC*).

The function returns as an integer value the constructed error code by taking the absolute value of messageNumber and adding 10 000 (the upper limit of the message numbers reserved for DCL system itself). The constructed code is also copied to the address specified by the third argument. If severity is zero or one, then the returned value shall be all zeros; otherwise, the severity byte is used as the most significant byte of the return value and the constructed error code is use as the least significant bytes.

If the message number contains any bits in the high-order byte, an informative message is issued.

10.28.4.13 dcmHardErrorRC

Table 489 provides information on dcmHardErrorRC.

Table 489—dcmHardErrorRC

Function name	dcmHardErrorRC
Arguments	Message severity
Result	None
Standard Structure fields	None
C syntax	<code>int dcmHardErrorRC(DCM_Message_Severities severity);</code>

This returns a return code constructed from the message severity argument. If the message severity is inform or warning (see Table 101), the return code is 0. Otherwise, the return code has the passed message severity with a message number of 0x00EEEEEE.

10.28.4.14 dcmSetMessageIntercept

Table 490 provides information on dcmSetMessageIntercept.

Table 490—dcmSetMessageIntercept

Function name	dcmSetMessageIntercept
Arguments	Application-defined function for printing messages
Result	None
Standard Structure fields	None
C syntax	<code>DCM_Message_Intercept_Type dcmSetMessageIntercept (DCM_Message_Intercept_Type msgfn);</code>

This sets the function pointer to be used to print messages generated by the DPCM or library. This function may be called at any time. This function is a pointer to the previous message intercept function or is NULL if there was no prior function.

For consistent message handling, the application shall set the message handler before it loads a DPCM. Message handlers can be changed at any time the application chooses; however, if a change is done after the DPCM is loaded, then only those messages occurring after the change are directed to the new handler. Messages prior to that shall be handled in a default manner as determined by DCL (if the message handler was not set) or as dictated by the previous call to *dcmSetMessageIntercept*.

10.28.4.15 dcmIssueMessage

Table 491 provides information on dcmIssueMessage.

Table 491—dcmIssueMessage

Function name	dcmIssueMessage
Arguments	Standard Structure pointer, Message number, Message severity, Message format string [format arguments]
Result	None
Standard Structure fields	None
C syntax	<code>int dcmIssueMessage (const DCM_STD_STRUCT *std_struct, int msgNum, DCM_Message_Severities msgSev, char* msgFormat [, ...]);</code>

This prints a message using the current message function in effect. This function assembles a complete DCL message from the severity, message number, format, and format arguments. *dcmIssueMessage* can be called by an application as well as by the DPCM (inside *INTERNAL* or in-line C code) to generate a DCL style message. Use this function instead of direct calls to *printf()* or *fprintf()* to help ensure proper ordering of both DPCM and application messages in the same output stream (see 10.28.4.14).

This function takes a minimum of four arguments. The first is the *DCM_STD_STRUCT* pointer. This argument is presented to keep this function consistent with the DCL standard function argument passing conventions. The message number and severity arguments shall follow the interface conventions defined in integer return code (see 10.10.1). The fourth argument is a format string that follows the conventions of the C *printf* function. The remaining arguments, if any, fulfill the conversion specifications identified in the format string.

This function returns the same integer value as *dcmMakeRC* would if it were passed the same severity and message number (see 10.28.4.12).

This function shall not buffer any messages.

10.28.4.16 dcm_rule_init

Table 492 provides information on *dcmrule_init*.

Table 492—*dcm_rule_init*

Function name	<i>dcm_rule_init</i>
Arguments	None
Result	None
Standard Structure fields	None
C syntax	<pre>int dcm_rule_init (DCMTransmittedInfo *xmitStruct, DCM_FunctionTable *externals);</pre>

This entry point called by the application after the *root* subrule of a DPCM was successfully loaded by *dcmBindRule*. *dcm_rule_init* causes the *root* subrule to load and link all other subrules and sets up the linkage for *EXPOSE* and *EXTERNAL* functions.

The call to *dcm_rule_init* takes two parameters: a pointer to a *DCMTransmittedInfo* and a pointer to a *DCM_FunctionTable*.

The *DCMTransmittedInfo* is a structure containing all the *EXPOSE* function pointer pairs along with a pointer to DPCM functions *modelSearch*, *delay*, *slew*, and *check*. Each *EXPOSE* function pointer pair consists of a string containing the name of the *EXPOSE* as it is seen in the subrule and a pointer to that function's entry point.

The second parameter is a pointer to a *DCM_FunctionTable* structure containing the application's *EXTERNAL* function pointer pairs. It is the application's responsibility to create this structure.

When *dcm_rule_init* is called, the DPCM loads the remaining subrules specified, cross-links all the *EXPORTs* and *IMPORTs*, and uses the *DCM_FunctionTable* to link the application *EXTERNAL* functions to the corresponding *EXTERNAL* functions listed in the DPCM. It then fills in the *DCMTransmittedInfo* with its *EXPOSEs* and modeling functions.

After the *root* subrule is loaded and its initialization function has been called, the application then does the following:

- Uses the *DCMTransmittedInfo* to initialize its pointers to the DPCM services required. *dcmFindFunction* and *dcmQuietFindFunction* are used to locate the function pointers associated with each *EXPOSE* desired.
- Initializes its modeling function pointer by the named field within the *DCMTransmittedInfo* for DPCM functions *modelSearch*, *delay*, *slew*, and *check*.

NOTE—This function is not called explicitly by name but is accessed a pointer supplied by the return value *dcmBindRule*.

10.28.4.17 DCM_new_DCM_STD_STRUCT

Table 493 provides information on *DCM_new_DCM_STD_STRUCT*.

Table 493—DCM_new_DCM_STD_STRUCT

Function name	DCM_new_DCM_STD_STRUCT
Arguments	None
Result	None
Standard Structure fields	None
C syntax	DCM_new_DCM_STD_STRUCT *DCM_new_STD_STRUCT(void);

This constructor function is used to allocate and properly initialize a Standard Structure.

10.28.4.18 DCM_delete_DCM_STD_STRUCT

Table 494 provides information on *DCM_delete_DCM_STD_STRUCT*.

Table 494—DCM_delete_DCM_STD_STRUCT

Function name	DCM_delete_DCM_STD_STRUCT
Arguments	None
Result	None
Standard Structure fields	None
C syntax	void DCM_delete_DCM_STD_STRUCT(DCM_STD_STRUCT *std_struct);

This destructor function is used to free a Standard Structure.

10.28.4.19 dcm_setTechnology

Table 495 provides information on *dcm_setTechnology*.

Table 495—dcm_setTechnology

Function name	dcm_setTechnology
Arguments	Standard Structure pointer, Pointer to technology name
Result	None
Standard Structure fields	None
C syntax	<pre>const char* dcm_setTechnology (DCM_STD_STRUCT *std_struct, const char *tech_name);</pre>

A DPCM can contain one or more technologies. If no technology was specified, then a DPCM contains the GENERIC technology. If a single technology was specified, then the DPCM contains that specified technology. If multiple technologies were specified, then the DPCM contains the GENERIC technology (at least for the root subrule), as well as the other specified technologies.

At any time, there is a current technology set in the Standard Structure; the DPCM as a whole has no notion of what technology is considered current. A newly created Standard Structure selects a technology according to the following rules:

- If the DPCM has no technology or has a single technology, that technology is selected.
- If the DPCM has multiple technologies, the GENERIC technology is selected.

An application can change the technology selected by a Standard Structure by calling either this function (dcm_setTechnology) or dcm_takeMappingOfNugget (see 10.28.4.25) to modify the passed Standard Structure to select the specified technology.

An application can switch between technologies by either using a single Standard Structure and calling dcm_setTechnology or dcm_takeMappingOfNugget, or maintaining multiple Standard Structures, each of which has been modified to select a different technology, and choosing the appropriate structure to pass across the PI.

10.28.4.20 dcm_getTechnology

Table 496 provides information on dcm_getTechnology.

Table 496—dcm_getTechnology

Function name	dcm_getTechnology
Arguments	Standard Structure pointer
Result	None
Standard Structure fields	None
C syntax	<pre>const char* dcm_getTechnology(const DCM_STD_STRUCT *std_struct);</pre>

This returns the technology name of the Standard Structure in use and returns a 0 value if completion is unsuccessful.

NOTE—Do not free the result string as it is constant.

10.28.4.21 dcm_getAllTechs

Table 497 provides information on dcm_getAllTechs.