INTERNATIONAL STANDARD

IEC 61523-1

First edition
2001-09

Delay and power calculation standards –

Part 1:
Integrated circuit delay and power calculation systems

Reference number
IEC 61523-1:2001(E)

Publication numbering

As from 1 January 1997 all IEC publications are issued with a designation in the 60000 series. For example, IEC 34-1 is now referred to as IEC 60034-1.

Consolidated editions

The IEC is now publishing consolidated versions of its publications. For example, edition numbers 1.0, 1.1 and 1.2 refer, respectively, to the base publication, the base publication incorporating amendment 1 and the base publication incorporating amendments 1 and 2.

Further information on IEC publications

The technical content of IEC publications is kept under constant review by the IEC, thus ensuring that the content reflects current technology. Information relating to this publication, including its validity, is available in the IEC Catalogue of publications (see below) in addition to new editions, amendments and corrigenda. Information on the subjects under consideration and work in progress undertaken by the technical committee which has prepared this publication, as well as the list of publications issued, is also available from the following:

- **IEC Web Site (www.iec.ch)**

- **Catalogue of IEC publications**

  The on-line catalogue on the IEC web site (www.iec.ch/catlg-e.htm) enables you to search by a variety of criteria including text searches, technical committees and date of publication. On-line information is also available on recently issued publications, withdrawn and replaced publications, as well as corrigenda.

- **IEC Just Published**

  This summary of recently issued publications (www.iec.ch/JP.htm) is also available by email. Please contact the Customer Service Centre (see below) for further information.

- **Customer Service Centre**

  If you have any questions regarding this publication or need further assistance, please contact the Customer Service Centre:

  Email: custserv@iec.ch
  Tel: +41 22 919 02 11
  Fax: +41 22 919 03 00

INTERNATIONAL STANDARD

IEC 61523-1

First edition
2001-09

Delay and power calculation standards –

Part 1:
Integrated circuit delay and power calculation systems

International Electrotechnical Commission 3, rue de Varembé Geneva, Switzerland
Telefax: +41 22 919 0300 e-mail: inmail@iec.ch IEC web site http://www.iec.ch

Commission Electrotechnique Internationale
International Electrotechnical Commission
Международная Электротехническая Комиссия

PRICE CODE XH

For price, see current catalogue

Contents

SECTION PAGE

List of Figures

List of Tables

INTERNATIONAL ELECTROTECHNICAL COMMISSION

DELAY AND POWER CALCULATION STANDARDS –

Part 1: Integrated circuit delay and power calculation systems

FOREWORD

1) The IEC (International Electrotechnical Commission) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of the 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, the IEC publishes International Standards. 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. The IEC collaborates closely with the International Organization for Standardization (ISO) in accordance with conditions determined by agreement between the two organizations.
2) The formal decisions or agreements of the 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 National Committees.
3) The documents produced have the form of recommendations for international use and are published in the form of standards, technical specifications, technical reports or guides and they are accepted by the National Committees in that sense.
4) In order to promote international unification, IEC National Committees undertake to apply IEC International Standards transparently to the maximum extent possible in their national and regional standards. Any divergence between the IEC Standard and the corresponding national or regional standard shall be clearly indicated in the latter.
5) The IEC provides no marking procedure to indicate its approval and cannot be rendered responsible for any equipment declared to be in conformity with one of its standards.
6) Attention is drawn to the possibility that some of the elements of this International Standard may be the subject of patent rights. The IEC shall not be held responsible for identifying any or all such patent rights.

International Standard IEC 61523-1 has been prepared by IEC technical committee 93: Design automation.

This standard is based on IEEE Std P1481 (1999): *IEEE Standard for delay and power calculation systems*.

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

FDIS	Report on voting
93/143/FDIS	93/144/RVD

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

This standard does not follow the rules for the structure of international standards given in Part 3 of the ISO/IEC Directives.

IEC 61523 consists of the following parts, under the general title: *Delay and calculation standards:*

IEC 61523-1, Part 1: *Integrated circuit delay and power calculation systems*

IEC 61523-2, Part 2: *Prelayout delay calculation model specification of CMOS ASIC libraries* (to be published)

The committee has decided that the contents of this publication will remain unchanged until 2006. At this date, the publication will be

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

Introduction

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 which 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.

A major integrated circuit design problem is to ensure the chip meets the designer's timing and power requirements. At this time in the EDA industry, there are numerous methodologies — many incompatible or inconsistent with each other — for representing timing and power information. The DPCS responds to the urgency to improve chip design methods.

The DPCS includes all aspects and uses of delay and power calculation (including synthesis, floorplanning, and simulation) during both pre-layout and post-layout phases of chip design. A fundamental requirement placed on the DPCS is that it work equally well with Verilog HDL and VHDL. In this draft of the standard, the scope of the DPCS is limited to integrated circuit designs.

The DPCS gains its leverage out of an effort by Silicon Integration Initiative (Si2) to standardize on contributed technology from IBM in the form of a Delay Calculation Language (DCL) and its companion procedural interface (PI), along with an effort by Open Verilog International (OVI) to standardize on file formats to exchange chip parasitic and cluster information in the form of a Physical Design Exchange Format (PDEF) and a Standard Parasitic Exchange Format (SPEF). SPEF is based on the Standard Parasitic Format (SPF) technology supplied by Cadence Design Systems. PDEF is based on technology provided by Synopsys.

Since each of these specifications is based on existing technology which has proven itself in industry, the time to introduce this standard into industry is greatly reduced over what would have been required for a newly created standard. The industry owes a great deal of gratitude to these companies for contributing these technologies.

The P1481 Working Group is composed of four subgroups: Architecture, Language, Parasitics and Clustering, and Power. The members of each of these groups are members of the P1481 Working Group. The P1481 Working Group has a vice-chair liaison with Japan to coordinate development and review of the specification there. The IEC TC93 WG 2 also participates in the review and development of this specification.

In addition to numerous individual contributors, both OVI, VHDL International (VI), and Si2 have played significant roles in the development of this standard and funding of the technical documentation.

DELAY AND POWER CALCULATION STANDARDS –

Part 1: Integrated circuit delay and power calculation systems

1. Overview

The Delay and Power Calculation System (DPCS) standard is a coordinated set of standards which 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. The four distinct standard specifications covered in this document include:

- A description language for timing and power modeling (DCL).
- A software procedural interface (PI) for communications between EDA applications and compiled libraries of DCL descriptions.
- A standard file exchange format for parasitic information about the chip design (SPEF).
- A standard file exchange format for floorplan cluster information relative to the chip design (PDEF).

1.1 Scope

As stated in the introduction, the scope of the DPCS standard is to make 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 these standards is integrated circuit timing and power. The standard may be applied to both unit logic cells supplied by the integrated circuit vendor and logical macros defined by the integrated circuit designer. Although this specification is written towards 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 do not *guarantee* it. They provide for a single executable software program which computes delay and power based on integrated circuit vendor-supplied algorithms (or designer-supplied algorithms for macros), but does not guarantee EDA applications correctly communicate the design-specific information required for these algorithms. By specifying standard exchange formats for parasitic data and floorplanning information, the 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. These specifications also do not detail how the information contained within the standard exchange files shall be obtained.

1.2 Purpose

As feature sizes for chips shrink below 0.5µm, interconnect delay effects 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) now need 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 delay and power equations. The designer now needs EDA applications which can be directed to specific timing constraints, and provide consistent and accurate characterization of a chip's timing and power before it is manufactured. Plus, due to the complexities of the delay and power equations, the integrated circuit vendor needs to have control of application calculations and not be restricted by unique characteristics of the broad set of applications demanded by the customers (the designers).

Over the past few years it has become increasingly apparent modern 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 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 then 0.25µm) designs. Now, EDA tools can accurately calculate and model power by using this DPCS standard.

1.3 Contents of this standard

The organization of the remainder of this standard is:

- Clause 2 (References) provides references to other applicable standards that are assumed or required for DPCS.
- Clause 3 (Definitions) defines terms used throughout the different specifications contained in this standard.
- Clause 4 (Acronyms and abbreviations) defines the acronyms used in this standard.
- Clause 5-1 (High-level DPCS architecture) gives an overview of how all of the elements of this standard fit together to achieve the DPCS goals.
- Clause 6 (Delay Calculation Language (DCL)) provides the specification for the language used to model timing and power.
- Clause 7 (Power modeling and calculation) provides the specification for modeling power and the interaction between an application and a DPCM for power calculations.
- Clause 8 (Procedural Interface (PI)) provides the specification of the functions used to communicate between EDA applications and the DCL based libraries as well as their software interface.
- Clause 9 (Parasitics) provides the specification for the exchange of parasitics which are required to calculate accurate delays.
- Clause 10 (Physical design exchange) provides the specification for the exchange of floorplan clustering information which is used to provide more accurate timing calculations based on approximate layout.
- Annexes. Following Clause 10 are a series of normative and informative appendices.

2. References

This standard shall be used in conjunction with the following publications. When the following standards are superseded by an approved revision, the revision shall apply.

ISO/IEC 9899:1990, Programming Languages — C

3. Definitions

For the purposes of this recommended practice, the following terms and definitions apply. IEEE Std 100-1992, The New IEEE Standard Dictionary of Electrical and Electronics Terms, should be referenced for terms not defined in this standard.

3.1 **application**, **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, etc. *See also:* **delay calculation module**, **procedural interface**.

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

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

3.4 **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.

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

3.6 **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**.

3.7 **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.

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

3.9 **bus**: In PDEF, a physical collection of *net*s and/or *pnet*s, or of *pin*s and/or *node*s. 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.

3.10 **C-effective**: A capacitance value, often computed as an approximation to an Resistor/Inductor/Capacitor (RLC) network or a π-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 which assumes a pure capacitive output load. *Syn:* **effective capacitance**.

3.11 **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 *DPCM* without reference back to the application for the internal structure of the cell. This primitiveness typically is 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. For PDEF, **cell** refers to a logical **instance** and a library primitive is called a **gate**.

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**.

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

3.13 **cluster**: A grouping of cell instances and/or clusters which 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. *Syn:* **partition**, **region**.

3.14 **column**: In a PDEF *datapath cluster*, a *cluster* of *cell*, *spare_cell*, and/or *cluster instance*s placed or constrained to be placed in the vertical (Y-axis) direction. *See also:* **row, datapath**.

3.15 **constraint**: A timing property of a design which is supplied as a goal or objective to an 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**.

3.16 **datapath**: A type of PDEF *cluster* which contains *row*s and/or *column*s of *cluster*, *cell*, *skip*, and/or *spare_cell instance*s. A PDEF datapath typically corresponds to structured logic. *See also:* **row, column**.

3.17 **delay**: The time taken for a digital signal to propagate between two points;

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

3.19 **delay calculation language (DCL)**: The programming language used to calculate instance-specific timing data. DCL contains high-level constructs which 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.

3.20 **delay calculation language 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**.

3.21 **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 EDA application. The DPCM is loaded into memory at run-time and linked to the application via the PI. A DPCM typically is created from DCL subrules compiled by the DCL compiler and linked together with run-time support modules.

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

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

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

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

3.26 **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 wireload models.

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

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

3.29 **function**, **PI function**: One of the C functions that comprise the DPCS procedural interface.

3.30 **gap**: In PDEF, spacing between *row*s and/or *column*s in a *datapath*.

3.31 **gate**: In PDEF, the physical abstraction of an library primitive.

3.32 **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.

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

3.34 **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*.

3.35 **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**.

3.36 **hold time**: *See:* **setup/hold timing check**, **nochange timing check**.

3.37 **software 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.

3.38 **software implementation limits**: Restrictions imposed by an implementation.

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

3.40 **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 PDEF, this is a physical *cluster* or a logical *cell*. *See also:* **cell**, **cell type**, **cluster**, **hierarchical instance**.

3.41 **interconnect**: A collective term for structures (in an integrated circuit) which 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, etc. But, interconnect shall not include such structures if they occur as part of the fixed layout of a cell.

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

3.43 **layer**: In PDEF, a particular level of interconnect on which a logical or physical pin is located.

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

3.45 **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.

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

3.47 **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.

3.48 **mesh table**: A multi-dimension table which 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.

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

3.50 **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.

3.51 **modeling statements**: DCL statements which map cell configurations to modeling procedures.

3.52 **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 an unique interconnect structure in the implementation.

3.53 **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*, i.e., 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.

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

3.55 **nugget**: A data structure used in the PI for rapid switching between technologies.

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

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

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

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

3.60 **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.

3.61 **π-model**: A simplification of a general RLC network that represents the driving point admittance for an interconnect.

3.62 **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.

3.63 **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.

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

3.65 **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 RLC circuit, since poles above this frequency can be ignored in calculations without significant loss of accuracy.

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

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

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

3.69 **procedural interface**, **PI**: The set of C functions used by an application and a DPCM to exchange information and determine the timing calculation for a design.

3.70 **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.

3.71 **RC, 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. *Syn:* **Elmore delay**.

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

3.73 **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.

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

3.75 **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**.

3.76 **region**: A region pertains to a particular physical section or block of a floorplan. *See:* **cluster**.

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

3.78 **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**.

3.79 **residue**: The value of $\lim_{s \to s_0} (s - s_0) \times F(s)$, where $F(s)$ has the complex pole s_0. *See also:* **pole**.

3.80 **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.

3.81 **route, global route**: In PDEF, the physical description of interconnect routing between logical and physical pins of *cell*, *spare_cell*, and/or *cluster instance*s.

3.82 **row**: In a 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**.

3.83 **scalar**: An integer constant.

3.84 **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.)

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

3.86 **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.

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, the reliability of the resulting state of the flip-flop or latch is unknown.

3.87 **setup time**: *See:* **setup/hold timing check**, **nochange timing check**.

3.88 **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**.

3.89 **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.

3.90 **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 interconnect, the sink is the receiver pin.

3.91 **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 wireload models. *See also:* **wireload model**.

3.92 **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.

3.93 **skip**: In PDEF, spacing between the ordered *cell*, *spare_cell*, and/or *cluster instance*s in *row*s and/or *column*s of a *datapath*.

3.94 **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 DPCS allows either interpretation if used consistently.

3.95 **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.

3.96 **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.

3.97 **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.

3.98 **soft region**: A cluster which 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.

3.99 **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.

3.100 **spare_cell**: A *cell instance* which 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.

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

3.102 **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.

3.103 **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 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.

3.104 **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 or., 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.

3.105 **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 off-line 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. Syn: **back annotation**.

3.106 **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.

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

3.108 **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.

3.109 **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 …").

3.110 **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. *Syn:* **intrinsic delay**.

3.111 **undefined behavior**: Behavior for which the standard imposes no requirements (e.g., use of an erroneous program construct). Permissible undefined behavior ranges from

— ignoring a situation completely with unpredictable results;
— behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message);
— 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.

3.112 **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.

3.113 **via**: In PDEF, a physical connection between two different levels of interconnect, or between a level of interconnect and a physical or logical pin.

3.114 **wireload 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 section lists the acronyms and abbreviations used in this standard.

API	Application Programming Interface (see also PI)
ASIC	Application Specific Integrated Circuit
AWE	Asymptotic Waveform Evaluation
CAE	Computer-Aided Engineering (the term EDA is preferred)
CFI	CAD Framework Initiative
DCL	Delay Calculation Language
DPCM	Delay and Power Calculation Module
DPCS	Delay and Power Calculation System
D-SPF	Detailed Standard Parasitic Format
EDA	Electronic Design Automation
EDIF	Electronic Design Interchange Format
HDL	Hardware Description Language
OID	Object Identifier
OVI	Open Verilog International
PDEF	Physical Design Exchange Format
PI	Procedural Interface
PVT	Process/Voltage/Temperature
RC	Resistance times Capacitance
RICE	Rapid Interconnect Circuit Evaluator
R-SPF	Reduced Standard Parasitic Format
SDF	Standard Delay Format
Si2	Silicon Integration Initiative
SPEF	Standard Parasitic Exchange Format
SPF	Standard Parasitic Format
SPICE	Simulation Program with Integrated Circuit Emphasis
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VI	VHDL International
VITAL	VHDL Initiative Towards ASIC Libraries
VLSI	Very Large Scale Integration

5. Delay and power calculation system architecture

5.1 Overview

The goals of the Delay and Power Calculation System (DPCS) architecture are to make it possible in a multi-vendor, multi-tool environment, for integrated circuit designers to calculate timing and power *consistently*, for integrated circuit vendors to express timing and power information *once* for a given technology, while enabling sufficient EDA application accuracy.

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 delay and power calculation to their customers independent of the release of design tools by EDA vendors.

Figure 5-1 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.

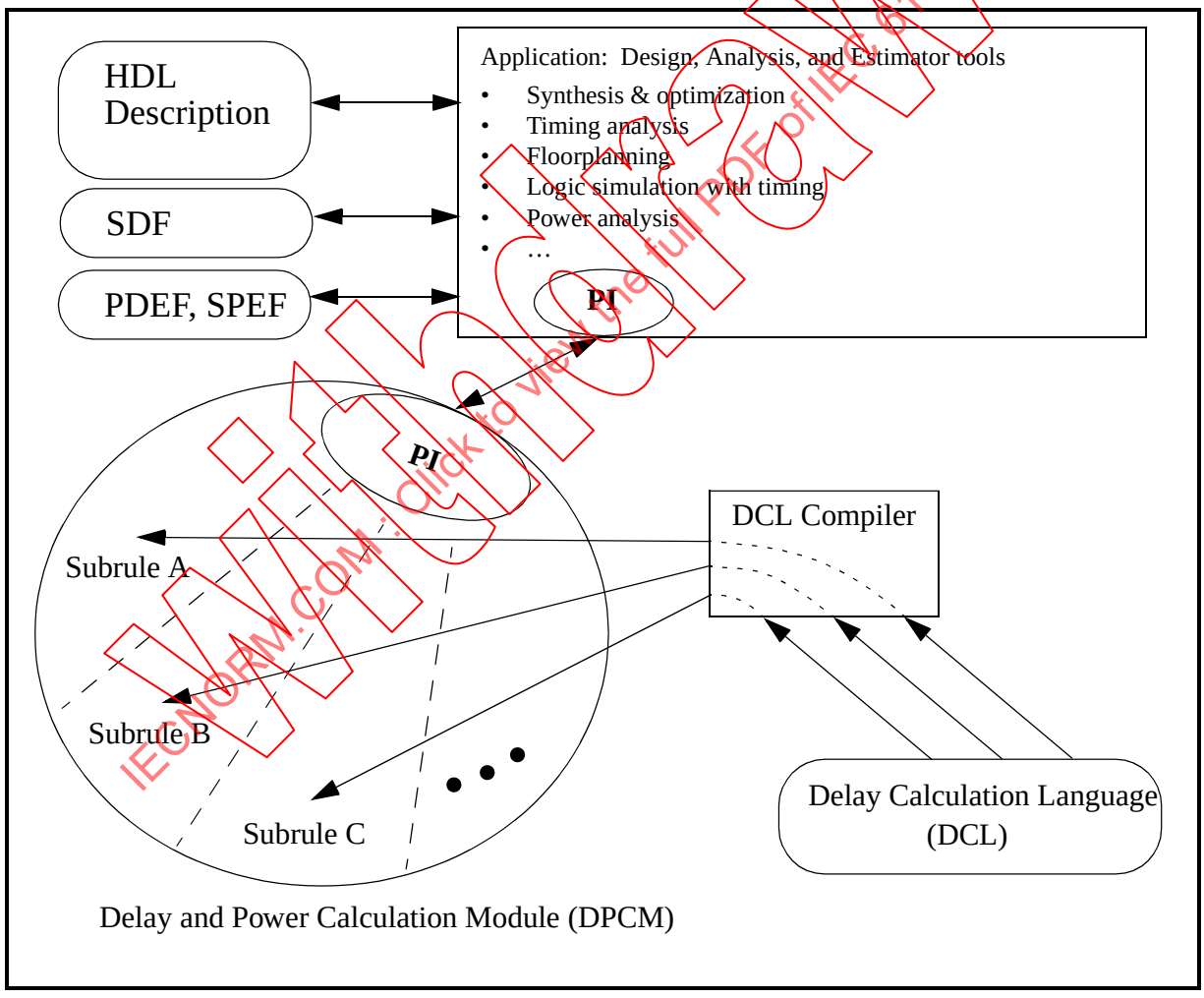


Figure 5-1 High-level DPCS architecture

The Delay and Power Calculation Module (DPCM) contains code (compiled from DCL source) which enables an application to compute power, timing delays and timing constraints efficiently.

- The procedural interface (PI) is used by both an application and the DPCM to control delay and power calculations.
- 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 post-layout data and write out the results in SDF.
- 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 front-end applications (synthesis, timing analysis, and partitioning) to those on the back-end.
- SPEF contains constructs to convey information between parasitic extractors and applications that utilize the extracted circuit information.

Figure 5-1 also illustrates several key points:

a) Applications shall be written (or modified) to use the DPCS-defined PI for all delay and power calculations.
b) The application is fully responsible for mapping between design instances and delay models.
c) The application is fully responsible for the state of the design. The DPCM knows everything about calculating delays and power, but nothing about a specific design. The DPCM uses the PI to acquire *all* design-specific information, including electrical modeling of parasitic effects and other data (often contained in SDF, PDEF, or SPEF files).
d) The DPCM is composed of one or more *subrules*. Subrules are compiled C code constructed from an ASCII description following the rules of the Delay Calculation Language (DCL). The DPCM is dynamically linked with the application during execution.
e) A subrule may implement one or more timing and power models.
f) One or more timing models may be incrementally patched through the release (and use) of multiple subrules.

5.2 Procedural interface

A crucial part of the DPCS standard is the procedural interface (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 8.

To enable an application to be independent of implementation-specific characteristics of a DPCM, code which implements the calls `dcmSetNewStorageManager`, `dcmSetMessageIntercept`, and/or `dcmBindRule` shall not be statically linked with the application. Figure 5-2 illustrates the required linkage structure. The code for these three functions shall reside in a shared object-code library named `libdcmlr`, dynamically linked to the application at run-time. There may be implementation-dependent procedure calls between `libdcmlr` and the DPCM, but this architecture isolates the application from such details.

NOTE — 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 `libdcmlr`.

5.2.1 Global policies and conventions

The DPCM assumes a single locus of control and data understanding exists for callbacks. 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, since the DPCM cannot know when any such data might become invalid. The DPCM *can* cache within a single application call.

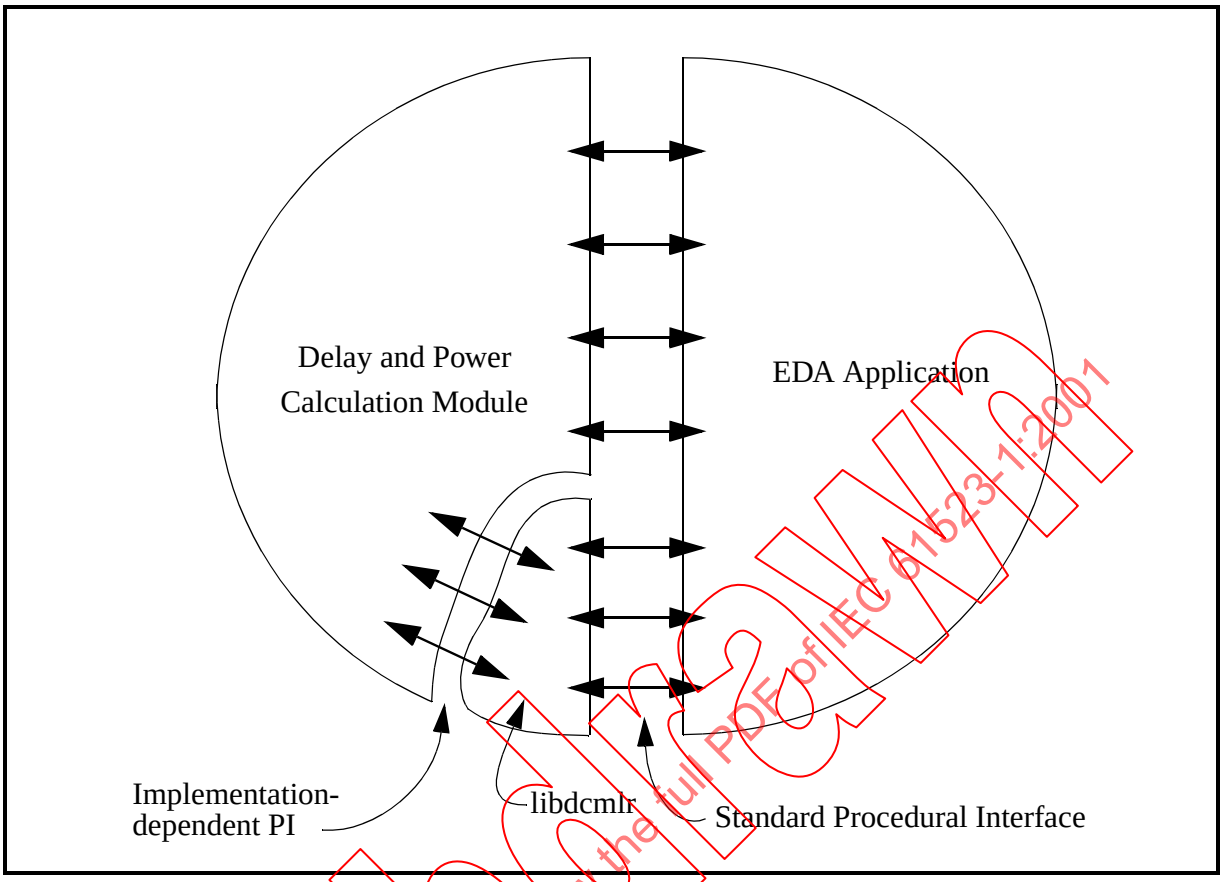


Figure 5-2 High-level DPCS architecture linkage structure

5.2.2 Flow of control

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

a) There is no "abnormal" flow of control across its PI, i.e., all function calls return control through the normal procedure-return mechanism.
b) A DPCM never terminates the execution of the application to which it is linked.

5.3 DPCM - application relationships

The DPCM has been designed to allow easy integration of dynamically linked executable modules for the purpose of delay and power calculation. The library developer decides how the components of the technology library should be implemented, then organizes the statements into logical groups called technology families and physical units called subrules. The run-time system assembles these units into a unified DPCM. The application then views the DPCM as a single sub-program from which it can request services.

5.3.1 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. Subrules may be segmented into technology families, which alters the way they are made available to the application (see 6.14).

5.3.2 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 independent components:

— C preprocessor directives
— Embedded C code
— DCL directives
— DCL atomic, model procedure, table manipulation, and library control statements

5.4 Inter-operability

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

In particular, an application can concurrently access subrules constructed in multiple implementations by

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`, and
c) requiring the application to
   1) track which technology families use which implementation,
   2) use the appropriately-named PI calls, and
   3) maintain separate implementation-specific standard structures.

6. Delay Calculation Language (DCL)

This section describes the delay modeling and calculation used in this standard. The formal syntax is described using the following Backus-Naur Form (BNF). For this clause only following conventions are used:

a) Lowercase words, some containing embedded underscores, are used to denote syntactic categories (terminals and non-terminals), e.g.,

literal_character_sequence

b) Boldface words are used to denote reserved keywords, operators, and punctuation marks as a required part of the syntax. These words appear in a larger font for distinction, e.g.,

MODEL SLEW => ;

c) The **::=** operator separates the two parts of a 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 (d) shows five options for a *timing_mode_operator*.

d) A vertical bar separates alternative items (use one only) unless it appears in boldface, in which case it stands for itself, e.g.,

timing_mode_operator ::= **->** | **<-** | **<->** | **<-X->** | **->X<-**

e) Square brackets enclose optional items unless it appears in boldface, in which case it stands for itself. For example:

result_definition ::= **RESULT (** [result_sequence] **)**

indicates *result_sequence* is an optional syntax item for *result_definition*, whereas

pin_range ::= pin_name **[** range_expression **]** pin_name

indicates square brackets are part of the syntax for *pin_range*.

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. Thus, the following two rules are equivalent:

comma_expression_list ::= expression { **,** expression }
comma_expression_list ::= expression | comma_expression_list **,** expression

g) Parenthesis enclose items within a group (use one only) unless it appears in boldface, in which case it stands for itself. In the following example:

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

the first set of parenthesis 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_first_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:

identifier_first_letter ::= **a-z** | **A-Z**

indicates the first letter of the identifier can be a lowercase letter (from a to z) or an uppercase letter (from A to Z).

j) If the name of any category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic information. For example, *msb*_constant_expression and *lsb*_constant_expression are equivalent to constant_expression.

The main text uses *italicized* font when a term is being defined, and `monospace` font for examples, file names, and while referring to constants such as `0`, `1`, or `x` values.

6.1 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, Programming Languages — C, excluding tri-graph 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, Programming Languages — C.

6.2 Lexical elements

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

6.2.1 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*.

6.2.2 Comments

A comment shall be either

— a character sequence which 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.
— a character sequence which starts with `//` and ends with the first occurrence of either the *newline* or *carriage_return* characters.

6.2.3 Tokens

The syntax for tokens in DCL is given in Syntax 6-1.

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

Syntax 6-1—Syntax for DCL token

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

6.2.3.1 Keyword

DCL reserves many tokens as *keywords* (see Table 6-1). These keywords shall be used only in the context as defined by DCL. No keyword shall be used as an identifier (see 6.2.3.2).

All keywords shall be case sensitive. Each keyword has two valid syntactic forms: one using only upper-case letters, and one using only lower-case letters. Table 6-1— shows only the upper-case syntactic form.

6.2.3.2 Identifier

An *identifier* is the name which represents a value, except within the `PATH`, `FROM`, `TO`, `BUS`, `TEST`, `INPUT`, NODE and `OUTPUT` clauses, 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.

The syntax for identifiers in DCL is given in Syntax 6-2.

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

Syntax 6-2—Syntax for DCL 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 the ISO C standard)
- any character sequence beginning with the letters `DCM` in any mixture of case

NOTE — It may be desirable to ensure that some *identifier*s are universally unique, so as to not collide when disparate library pieces from different companies are combined. Examples of such *identifier*s are `METHOD` names (see 6.12.7) and `TECH_FAMILY` names (see 6.14.1.1).

6.2.3.3 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 6-3.

```
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 6-3—Syntax for DCL double quoted character

Table 6-1—Keywords

ADD_ROW	FILE	REFERENCE
ANYIN	FILE_PATH	REFERENCE_EDGE
ANYOUT	FILTER	REFERENCE_EDGE_SCALAR
ASSIGN	FLOAT	REFERENCE_MODE
BIAS	FORWARD	REFERENCE_MODE_SCALAR
BLOCK	FROM	REFERENCE_POINT
BOTH	FROM_POINT	REFERENCE_SLEW
BUS	HOLD	REMOVAL
BY	IMPORT	REPLACE
CALC	IMPURE	REPEAT
CALC_MODE	INCONSISTENT	RESULT
CALC_MODE_SCALAR	INPUT	RISE
CALL	INPUT_PIN_COUNT	RULE_PATH
CELL	INTEGER	SETUP
CELL_DATA	INTERNAL	SETVAR
CELL_QUAL	KEY	SIGNAL
CGHT	LATE	SIGNAL_EDGE
CGPW	LATE_MODE	SIGNAL_EDGE_SCALAR
CGST	LATE_MODE_SCALAR	SIGNAL_MODE
CHECK	LATE_SLEW	SIGNAL_MODE_SCALAR
CHECKS	LOAD_TABLE	SIGNAL_POINT
CKTTYPE	METHOD	SIGNAL_SLEW
CLKFLG	METHODS	SINK_EDGE
COMPARE	MODEL	SINK_EDGE_SCALAR
COMPILATION_TIME_STAMP	MODEL_DOMAIN	SINK_MODE
CONSISTENT	MODEL_NAME	SINK_MODE_SCALAR
CONTROL_PARM	MODELPROC	SKEW
CPW	MONOLITHIC	SLEW
CST	NEW	SOURCE_EDGE
CYCLEADJ	NIL	SOURCE_EDGE_SCALAR
DATA	NOCHANGE	SOURCE_MODE
DEFAULT	NODE	SOURCE_MODE_SCALAR
DEFINES	NODE_COUNT	STATEMENTS
DELAY	NUMBER	STORE
DELETE_ROW	ONE_TO_Z	STRING
DHT	OPTIONAL	SUBMODEL
DO	OTHERWISE	SUBRULE
DOUBLE	OUTPUT	SUBRULES
DPW	OUTPUT_PIN_COUNT	SUPPRESS
DST	OVERRIDE	TABLE
DYNAMIC	PASSED	TABLEDEF
EARLY	PATH	TABLE_PATH
EARLY_MODE	PATH_DATA	TECH_FAMILY
EARLY_MODE_SCALAR	PATH_SEPARATOR	TERM
EARLY_SLEW	PHASE	TEST
EDGES	PIN	TEST_TYPE
END	PINLIST	TO
EXPORT	PROPAGATE	TO_POINT
EXPOSE	PROPERTIES	UNLOAD_TABLE
EXTERNAL	PROTOTYPE_RECORD	UNTIL
FALL	PURE	VAR
	QUALIFIERS	VOID
	RECOVERY	WHEN
		Z_TO_ONE
		Z_TO_ZERO
		ZERO_TO_Z

6.2.3.4 Predefined references to Standard Structure fields

DCL defines a set of identifiers to reference fields in the *Standard Structure* (see 8.13). These identifiers shall be visible in all scopes. Table 6-2 lists the predefined identifiers.

Table 6-2—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: bestCase, worstCase, or nominal.
CALC_MODE_SCALAR	INTEGER	Passed in calculation.	Indicates the type of calculation associated with the current propagate segment calculation: bestCase, worstCase, 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 period).
CELL_DATA	VOID	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	INTEGER	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.

Table 6-2—DCL predefined references to Standard Structure fields *(continued)*

Predefined Identifier	Data Type	Set by DPCM or passed in by application	Description
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.
INPUT_PIN_COUNT	INTEGER	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	INTEGER	Passed in during model search.	Holds the number of nodes connected to the circuit currently being modeled or under calculation.
OUTPUT_PIN_COUNT	INTEGER	Passed in during model search.	Holds the number of output pins connected to the circuit currently under calculation or model build.
PATH	STRING	Set during model search, Passed in calculation.	Holds the path or test statement name.
PATH_DATA	VOID	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`.

Table 6-2—DCL predefined references to Standard Structure fields *(continued)*

Predefined Identifier	Data Type	Set by DPCM or passed in by application	Description
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	INTEGER	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	INTEGER	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_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 which 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	INTEGER	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	INTEGER	Set during model search, Passed in calculation.	Indicates whether the calculation represents early, mode, late mode, or both.

Table 6-2—DCL predefined references to Standard Structure fields *(continued)*

Predefined Identifier	Data Type	Set by DPCM or passed in by application	Description
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.
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	INTEGER	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	INTEGER	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	INTEGER	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.

Table 6-2—DCL predefined references to Standard Structure fields *(continued)*

Predefined Identifier	Data Type	Set by DPCM or passed in by application	Description
SOURCE_MODE_SCALAR	INTEGER	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.

6.2.3.5 Compiler generated predefined identifiers

The DCL compiler generates the values for the predefined identifiers shown in Table 6-3.

Table 6-3—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 which loaded this rule.
RULE_PATH	STRING	Gives access to the `RULE_PATH` value specified in the `SUBRULE` statement which loaded this rule.
TABLE_PATH	STRING	Gives access to the `TABLE_PATH` value specified in the `SUBRULE` statement which loaded this rule.

6.2.3.6 Constant

A *constant* is defined as either a floating point constant number or an integer constant number.

The syntax for constants in DCL is given in Syntax 6-4.

```
constant ::= floating_constant | integer_constant
```

Syntax 6-4—Syntax for DCL constant

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

6.2.3.6.1 Pre-defined constant NIL

`NIL` is a pre-defined 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.

6.2.3.6.2 Edge type enumerations

The values for the SINK_EDGE and SOURCE_EDGE predefined identifiers are enumerations of the edge values as shown in Table 6-4. 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 6-4—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 since 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 since 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

NOTE — See 8.11.7 for DCM_EdgeTypes.

6.2.3.6.3 Propagation type enumerations

SINK_MODE and SOURCE_MODE are converted to enumerations as shown in Table 6-5.

Table 6-5—Propagation mode conversions

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

6.2.3.6.4 Calculation mode enumeration

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

Table 6-6—Calculation mode conversions

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

6.2.3.6.5 Test type enumeration

TEST_TYPE values are converted enumerations as shown in Table 6-7.

Table 6-7—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
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

6.2.3.7 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 6-5.

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

Syntax 6-5—Syntax for DCL string literal

*string_literal*s follow the same semantics as ISO C string literals (see the ISO C standard, Section 6.1.4).

6.2.3.8 Operators

The syntax for DCL operators is given in Syntax 6-6.

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

Syntax 6-6—Syntax for DCL operator

6.2.3.9 Punctuator

The syntax for punctuators is given in Syntax 6-7.

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

Syntax 6-7—Syntax for DCL punctuator

The punctuators `()`, `[]`, `%{}%`, and `{}` shall occur in balanced pairs.

6.2.3.10 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 clause.

6.2.4 Header names

Header name preprocessing tokens shall only appear with a `#include` preprocessing directive. The header name shall be defined the same as in section 6.1.7 ("Header Names") of the ISO C standard.

6.2.5 Preprocessing directives

DCL preprocessing directives are exactly the set defined in Section 6.8 of the ISO C standard and have the same semantics.

6.3 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, and
- all other identifiers.

6.4 Storage durations of objects

An object has a *storage duration* which determines its lifetime. There are two storage durations: *static* and *automatic*.

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 non-`array` objects whose identifiers are not defined by a `STORE` clause or an `ASSIGN` statement shall have *automatic storage duration*. For such objects, their storage shall be allocated when the defining statement is called and shall persist until the calling statement goes out of scope.

For all `array` objects, their storage shall be allocated by the `NEW` operator and shall persist (while in scope within the DPCM) at least until control reaches the application or while the application or the DPCM has locked the array(see 8.4.1).

6.5 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*. There are ten different types of scope:

- global
- `TECH_FAMILY`
- subrule
- statement-prototype
- statement
- modeling procedure
- `STATEMENTS`
- `RESULT`
- `PASSED`
- 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:

global
    TECH_FAMILY

```
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`.

6.6 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*. There are three subrule scope options affecting linkage: `EXPORT`, `IMPORT`, and `FORWARD`. Expose chaining also defines how `EXPOSE` identifiers are linked within a `TECH_FAMILY`'s scope. (See 8.8.1.1 for more details.)

6.6.1 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.

6.6.2 IMPORT

A statement *identifier* `EXPORT`ed 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.

6.6.3 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.

6.6.4 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).

6.7 DCL data types

DCL supports three categories of data types: *native*, *derived*, and *array*. The native data types are defined in terms of the ISO C data types. See 8.11.6 for details of the mapping between DCL and ISO C data types.

6.7.1 Native data types

The syntax for native data types in DCL is given in Syntax 6-8.

native_type ::= mathematical_type | pointer_data_type

Syntax 6-8—Syntax for DCL native data type

6.7.1.1 Mathematical calculation data types

The syntax for calculation data types in DCL is given in Syntax 6-9.

mathematical_type ::= **NUMBER** | **DOUBLE** | **FLOAT** | **INTEGER**

Syntax 6-9—Syntax for DCL calculation data type

6.7.1.2 Pointer data types

The syntax for pointer data types in DCL is given in Syntax 6-10.

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

Syntax 6-10—Syntax for DCL pointer data type

6.7.2 Array types

The syntax for array types in DCL is given in Syntax 6-11.

native_array_type ::= [mathematical_type | STRING | PIN | VOID] **[** dimension_list **]**
dimension_list ::= fixed_dimension_list | variable_dimension_list
fixed_dimension_list ::= scalar { **,** scalar }
variable_dimension_list ::= ***** { **,** ***** }

Syntax 6-11—Syntax for DCL array type

An array type describes a contiguously allocated nonempty set of objects with a particular member object type, 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 may have type *fixed* or *variable*. The number of elements in a *fixed* dimension shall be specified when the array is declared and shall not vary during program execution. 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 are true for all array types:

- 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.
- The length of each dimension in a multi-dimensional array shall not depend on the index used for any of the other dimensions of that array — “ragged” arrays shall not be supported.
- Array indices shall start with zero (`0`) in each dimension.
- Array types shall be passed and returned as pointers to a contiguous memory region. The array data shall be organized as defined by the ISO C standard.

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 VAR array modifier is defined in Table 6-8.

Table 6-8—VAR array modifier

Symbol	Definition
VAR, **var**	Indicates (to the compiler) the array field is modifiable by any variable that references the array. Without the `var` array modifier, arrays are constant and cannot be modified in any way.

Example

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

The VAR array 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.

6.7.3 Derived data types

The syntax for derived data types is given in Syntax 6-12.

```
derived_type ::= result_type | statement_type
result_type ::=
      assign_statement_name
    | calc_statement_name
    | expose_statement_name
    | external_statement_name
    | internal_statement_name
    | tabledef_statement_name
statement_type ::= result_type ( )
```

Syntax 6-12—Syntax for DCL derived data type

6.7.3.1 Result types

RESULT clauses in statements and DATA clauses in TABLEDEFs define derived data types which shall be analogous to structures in the ISO C standard. The name of the associated statement or TABLE can be used as the name of the derived data type. RESULT type field ordering is defined section 6.12.1.2.1 and 6.12.1.2.2.

Example

```
calc(resultType):     passed(integer: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)));
```

6.7.3.2 Statement types

A statement name followed by ( ) defines a derived data type. Any statement having the same PASSED arguments (number, order, and type) and RESULT structure matches this derived type. This enables the passing of statement pointers into other statements.

Example

exercise() passes statementType1() to example() as a derived type:

```
calc(statementType1):   passed(integer:x)
                        result(integer:resType(x)+ 3*x);
calc(statementType2):   passed(integer:z)
                        result(integer:resType(z)+6*z+ 1);
calc(example):          passed(statementType1(): y)
                        result(integer: y(5));
calc(exercise):         result(integer:example(statementType1)+
                                       example(statementType2));
```

6.8 Type conversions

Type conversions shall be performed based on

— changing the type of an argument to the type of the expected parameter or
— changing the type of an expression term to the type of other terms in the same expression.

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

6.8.1 Implicit conversions

The following implicit type conversions shall be performed:

— `INTEGER` to `DOUBLE` or `FLOAT`

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

— `FLOAT` to `DOUBLE`

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

— `PIN` to `STRING`

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

— `native_array_type, pointer_data_type or derived_type` to `VOID`

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

6.8.2 Explicit conversions

The following explicit type conversions shall be performed:

— `FIXED`-dimension array to `VARIABLE`-dimension array

A `FIXED`-dimension array shall be converted to a `VARIABLE`-dimension array when a fixed dimension array is assigned to a variable dimension array of the same dimensionality and data type.

— `DERIVED` data type to `DERIVED` data type

A `DERIVED` data type shall be converted to an equivalent `DERIVED` data type when the expected parameter has the same structure (order, number, and native types) as the argument.

Example:

```
CALC(resType): passed(integer:x)
               result(double:fp = 2*x+3 & integer:x+1);
CALC(resType2):passed(integer:x)
               result(double:y=2**x & integer: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)));
```

6.9 Operators

This section details the operators used in DCL.

6.9.1 String prefix operator

This subsection details the use of string prefix operators.

6.9.1.1 Explicit string prefix operator

The *explicit string prefix operator* is a unary operator which 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 which precedes a string literal or an *identifier* having a string value.

6.9.1.2 Embedded string prefix operator

The *embedded string prefix operator* is a unary operator which 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 (\).

6.9.1.3 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.

6.9.2 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.

6.9.3 New operator

The new operator (NEW) shall be used to create memory space for arrays. `NEW` is used when a new instance of an array is needed. After the `NEW` operator creates the space for an array, the values of the individual members shall be undefined.

6.9.4 SCOPE operator

The scoping prefix `PATH_DATA::` or `CELL_DATA::` determines what *action statement* is associated with a method (see 6.12.7), and what the value of a store variable is, based on the value of the corresponding predefined identifier `PATH_DATA` or `CELL_DATA` (see Syntax 6-16).

6.9.5 Purity operator

A reference to an impure statement shall be treated as PURE if the reference is preceded by the purity operator (^) (see 6.12.2.1). The purity operator is defined in Table 6-9.

Table 6-9—Purity operator

Symbol	Definition
^	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.

6.9.6 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, defined in "Timing resolution modes" on page 56 and used in PROPAGATE, EDGE and TEST clauses.

Table 6-10—Timing resolution modes

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

6.9.7 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 typeSee "Test type enumeration" on page 47., test mode operator See "Test mode operators" on page 57., and EDGE clause.

DCL supports three ways to perform this comparison, defined in "Test mode operators" on page 57.

Table 6-11—Test mode operators

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

6.9.8 Test mode operators

This subsection details the use of test mode operators.

6.9.8.1 CGHT

The Clock Gating Hold test mode operator (`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.

6.9.8.2 CGPW

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

6.9.8.3 CGST

The Clock Gating Setup test mode operator (CGST) 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.

6.9.8.4 CPW

The Clock Pulse Width test mode operator (`CPW`) 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.

6.9.8.5 CST

The Clock Separation test mode operator (`CST`) 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.

6.9.8.6 DHT

The Data Hold test mode operator (DHT) 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 shall ensure the SIGNAL_EDGE comes after the REFERENCE_EDGE. The difference between this test mode operator and DST (see 6.9.8.8) 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.

6.9.8.7 DPW

The Data Pulse Width test mode operator (DPW) 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.

6.9.8.8 DST

The Data Setup test mode operator (DST) is used to determine the offset between two data signals. This separation is established by the CHECKS clause. 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.

6.9.8.9 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 following the both modes operator. The reference edge shall be from a clock pin.

6.9.8.10 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.

6.9.8.11 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.

6.9.8.12 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.

6.9.8.13 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.

6.9.8.14 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.

6.10 Expressions

An expression is a sequence of operators and operands which:

— specifies the computation of a value,
— designates an object or a function,
— generates side effects, or
— 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.

6.10.1 Array subscripting

The syntax for array subscripting is given in Syntax 6-13.

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

Syntax 6-13—Syntax for array subscripting

A reference to an array element shall be made using the array name followed by a non-empty list of subscript expressions, 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.

6.10.2 Statement calls

This section describes statement calls in DCL.

6.10.2.1 General syntax

The syntax for statement calls is given in Syntax 6-14.

```
statement_call ::= statement_name ( [comma_expression_list ] )
statement_reference ::= statement_call [ array_index ] . field_reference
```

Syntax 6-14—Syntax for statement call

Arguments passed to a *statement* shall be read-only within that statement, with the sole exception of VAR arrays (see Table 6-8), which shall also be writable by the *statement*.

6.10.2.2 Built-in function calls

The syntax for built-in function calls is given in Syntax 6-15.

```
built-in_function_call ::= built-in_name ( [comma_expression_list ] )
```

Syntax 6-15—Syntax for built-in function call

6.10.2.3 Method statement calls

The syntax for method statement calls is given in Syntax 6-16.

```
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 6-16—Syntax for 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.

6.10.3 Assign variable reference

The syntax for assign variable references is given in Syntax 6-17.

```
assign_variable_reference ::=
    assign_statement_name [ array_index | . field_reference ]
field_reference ::= result_field_name [ array_index ]
```

Syntax 6-17—Syntax for assign variable reference

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

6.10.4 Store variable reference

The syntax for store variable references is given in Syntax 6-18.

```
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 6-18—Syntax for 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, use of `PATH_DATA::` shall be assumed.

6.10.5 Mathematical expressions

This section details the mathematical expressions allowed in DCL and gives their syntax (see Syntax 6-19).

```
expression ::=
      constant
    | identifier
    | string_literal
    | 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 )
variable_reference ::= expr_variable[ array_index | . field_reference ]
expr_variable ::=
      assign_statement_name
    | passed_argument_name
    | result_field_name
    | predefined_variable_name
```

Syntax 6-19—Syntax for expression

6.10.5.1 Mathematical operators

The mathematical operators are defined in Table 6-12.

Table 6-12—Mathematical operators

Symbol	Definition
*	multiply
/	divide
+	add, unary plus
-	subtract, unary minus
**	exponentiation

6.10.5.2 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 6-20.

```
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 ::= INTEGER : loop_variable_name = integer_expression
discrete_end ::= TO integer_expression
discrete_operator ::= + | *
discrete_expression_body ::= expression
```

Syntax 6-20—Syntax for discrete math expression

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

* indicates a product operation on the *loop_body* expression and

+ 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.

6.10.5.2.1 INTEGER discrete

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

6.10.5.2.2 PINLIST discrete

The pinlist discrete expression indicates the loop variable has type `PINLIST` and is stepped though the list of `PIN`s in the `PINLIST`.

6.10.6 Logical expressions and operators

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

The syntax for logical expressions is given in Syntax 6-21.

```
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 6-21—Syntax for logical expression

The DCL logical operators are defined in Table 6-13.

Table 6-13—Logical operators

Symbol	Definition
||	or
&&	and
!	not
==	equal
!=	not equal
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

6.10.7 Pin range

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

6.10.7.1 Pin range syntax

The syntax for a pin range is given in Syntax 6-22.

```
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 6-22—Syntax for pin range expression

NOTE — The use of angle brackets (<>) or square brackets ([]) affects the expansion of the Pin Range (see 6.10.7.2).

6.10.7.2 Pin range semantics

a) *pin_name*
represents a single pin with the specified name.

b) `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.

c) `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.

d) *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`)

1) *Constraints*:

The specified pin names shall have the same number of characters.
The specified pin names shall contain only characters within `A-Z`, `a-z`, `0-9`.

2) *Name incrementation rule*:

Names are incremented by lexically incrementing the right-most character of the name according to the *Character incrementation rule*s below. When a character being incremented in a name cycles back, then the character to its left (if any) is incremented.

3) *Character incrementation rules*:

A character in a name is lexically incremented through a specific range of characters. This range depends on the initial character.

If the initial character is in the range `A-Z`, then this character shall be incremented through the range of characters `A-Z`. When the character `Z` is incremented, it becomes `A`.

If the initial character is in the range `a-z`, then this character shall be incremented through the range of characters `a-z`. When the character `z` is incremented, it becomes `a`.

If the initial character is in the range `0-9`, then this character shall be incremented through the range of characters `0-9`. When the character `9` is incremented, it becomes `0`.

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
```

e) [*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 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
```

f) [*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 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
```

6.10.8 Embedded C code expressions

The syntax for an embedded C code expression is given in Syntax 6-23.

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

Syntax 6-23—Syntax for embedded C code expression

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.

6.11 Computation order

This section details the computation order used in DCL.

6.11.1 Mathematical expressions

Mathematical expressions shall be evaluated according to the operator precedence shown in Table 6-14.

Table 6-14—Mathematical operator precedence (high to low)

Operators	Associativity
() [] .	left to right
+ - (unary operators)	right to left
**	left to right
* /	left to right
+ -	left to right

6.11.2 Logical expressions

Logical expressions shall be evaluated according to the operator precedence shown in Table 6-15.

Table 6-15—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 guarantee left-to-right evaluation; 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 guarantee left-to-right evaluation; 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.

6.11.3 Passed parameters

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

6.11.4 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.

6.11.5 REPEAT - UNTIL clause

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

6.12 DCL statements

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

6.12.1 Clauses

This subsection defines the PASSED and RESULT clauses.

6.12.1.1 PASSED clause

The PASSED clause is an ampersand delimited list which declares the quantity, types, and names of the required formal input parameters for a DCL statement. Its syntax is given in Syntax 6-24.

passed_clause ::= **PASSED (** passed_argument_list **)**
passed_argument_list ::= passed_type_list { **&** passed_type_list }
passed_type_list ::= passed_type **:** variable_name_list
passed_type ::= native_array_type | native_type | derived_type
variable_name_list ::= name { **,** name }

Syntax 6-24—Syntax for PASSED *clause*

6.12.1.2 RESULT clause

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

6.12.1.2.1 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 6-25.

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

Syntax 6-25—Syntax for RESULT *prototype*

6.12.1.2.2 Conditional logic

The conditional RESULT clause defines the types, names (except the unnamed RESULT variable), and the 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 6-26.

```
conditional_result ::=
      RESULT ( result_sequence )
    | REPEAT RESULT ( result_sequence ) UNTIL ( logical_expression )
    | outter_when_sequence
result_sequence ::=
      result_expression
    | condition_logic
    | result_sequence & result_expression
    | result_sequence & condition_logic
result_expression ::=
      named_result_expression
    | default_result_expression
    | named_result_expression & default_result_expression
named_result_expression ::=
    result_type : assignment { & result_type : assignment }
assignment ::=
    variable_name = expression { , variable_name = expression }
default_result_expression ::= result_type : expression
conditional_logic ::=
      REPEAT result_definition UNTIL ( logical_expression )
    | when_list , OTHERWISE opt_result_definition
when_list ::= when_logic { , when_logic }
when_logic ::= WHEN ( logical_expression ) opt_result_definition
opt_result_definition ::= RESULT ( result_sequence ) | RESULT ( )
outter_when_sequence ::= outter_when_list , OTHERWISE result_definition
outter_when_list ::=
      WHEN ( logical_expression ) result_definition
    { , WHEN ( logical_expression ) result_definition }
result_definition ::= RESULT ( result_sequence )
```

Syntax 6-26—Syntax for conditional result

6.12.1.2.3 Default variable

A RESULT clause may have an unnamed, *default* variable. Such a variable shall

- 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.

Examples

```
RESULT(INTEGER: 5)
RESULT(DOUBLE: a = 4.32 & INTEGER: 5)
```

6.12.1.3 Default Clause

The syntax for the DEFAULT clause is given in Syntax 6-27.

default_clause ::= [**DEFAULT (** default_sequence **)**]
default_sequence ::= named_expression_list [**&** default_expression]
named_expression_list ::=
    variable_name **=** expression { **&** variable_name **=** expression }
default_expression ::= expression

Syntax 6-27—Syntax for DEFAULT *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.

6.12.2 Modifiers

This section describes how modifiers are used in DCL.

6.12.2.1 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:

a) ASSIGN, EXPOSE, LOAD_TABLE, UNLOAD_TABLE, ADD_ROW, and DELETE_ROW statements shall be considered to be IMPURE.
b) Statements executed by means of a statement pointer shall be considered to be impure.
c) 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.
d) Variables created with the NEW operator shall be considered to be impure.

NOTES

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

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.

6.12.2.2 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:

a) The following statements are considered to be `INCONSISTENT`:
   1) `TABLEDEF` statements with the `DYNAMIC` option.
   2) `LOAD_TABLE` statements.
   3) `UNLOAD_TABLE` statements.
   4) `ADD_ROW` statements.
   5) `DELETE_ROW` statements.
   6) `EXTERNAL` statements.
   7) `ASSIGN` statements.
   8) `INTERNAL` statements.
   9) Statements with the `IMPORT` prototype modifier.

b) All other statements are considered to be `CONSISTENT` unless the statements:
   1) are impure statements.
   2) reference an inconsistent statement.
   3) reference embedded C code.
   4) Executed by means of a statement pointer.

c) 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`. `IMPURE CONSISTENT` is self contradictory.

`IMPORT PURE` without `CONSISTENT` shall be understood as `IMPORT PURE INCONSISTENT`. `IMPORT CONSISTENT` without `PURE` shall be understood as `IMPORT PURE CONSISTENT`. `FORWARD PURE` without `CONSISTENT` shall be understood as `FORWARD PURE CONSISTENT`. `FORWARD CONSISTENT` without `PURE` shall be understood as `FORWARD PURE CONSISTENT`.

NOTES

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

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.

6.12.3 Prototypes

This section describes how prototypes are used in DCL.

6.12.3.1 Prototype modifiers

The syntax for prototype modifiers is given in Syntax 6-28.

```
prototype_modifier ::= IMPORT | FORWARD
std_postfix_modifier ::=
    [ optimize_ctl_postfix_modifier ] [ model_ctl_postfix_modifier ]
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
```

Syntax 6-28—Syntax for prototype modifier

6.12.3.2 Common prototypes for ASSIGN, CALC, EXPOSE, EXTERNAL, and INTERNAL statements

The syntax for common prototype modifiers is given in Syntax 6-29.

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

Syntax 6-29—Syntax for common prototype modifier

6.12.3.3 TABLEDEF prototype

The syntax for TABLEDEF prototype modifiers is given in Syntax 6-30.

```
tabledef_prototype ::=
      IMPORT TABLEDEF (name) std_postfix_modifier [ OVERRIDE ] :
      [ passed_clause ] qualifiers_clause data_clause [ key_clause ] ;
    | IMPORT TABLEDEF (name) std_postfix_modifier [ OVERRIDE ] :
      [  passed_clause ] qualifiers_clause [ key_clause ] data_clause ;
    | IMPORT TABLEDEF (name) std_postfix_modifier [ OVERRIDE ] :
      [ passed_clause ] [key_clause ] qualifiers_clause data_clause ;
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 6-30—Syntax for `TABLEDEF` *prototype modifier*

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 `PURE CONSISTENT`.

6.12.3.4 Load and Unload table prototypes

The syntax for load and unload table prototype modifiers is given in Syntax 6-31.

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

Syntax 6-31—Syntax for `LOAD_TABLE` *and* `UNLOAD_TABLE` *prototype modifier*

6.12.3.5 ADD_ROW and DELETE_ROW prototypes

The syntax for the add and delete row prototype modifiers is given in Syntax 6-32.

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

Syntax 6-32—Syntax for `ADD_ROW` *and* `DELETE_ROW` *prototype modifier*

6.12.3.6 DELAY and SLEW prototypes

The syntax for the `DELAY` and `SLEW` prototype modifiers is given in Syntax 6-33.

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

Syntax 6-33—Syntax for `DELAY` *and* `SLEW` *prototype modifier*

6.12.3.7 CHECK prototype

The syntax for the `CHECK` prototype modifier is given in Syntax 6-34.

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

Syntax 6-34—Syntax for `CHECK` *prototype modifier*

6.12.4 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, 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, a non-zero ("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 (see 8.10).

NOTE — 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, that clause is evaluated. Otherwise, statement S1 fails.

6.12.5 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 Table 8-8 and Table 8-9.

6.12.5.1 EXPOSE statement

The syntax for the EXPOSE statement is given in Syntax 6-35.

```
expose_statement ::=
    [ EXPORT ] EXPOSE ( name ) method_postfix_modifier :
    [ passed_clause ] conditional_result ;
```

Syntax 6-35—Syntax for 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:

— imported using the IMPORT reserved word.
— exported (by default or by the EXPORT reserved word explicitly).
— an EXPOSE statement 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 (see 8.8.1.1).

6.12.5.2 EXTERNAL statement

The syntax for the EXTERNAL statement is given in Syntax 6-36.

```
external_statement ::=
    [ EXPORT ] EXTERNAL ( name ) external_postfix_modifier :
    [ passed_clause ] result_prototype [default_clause ] ;
external_postfix_modifier ::= std_postfix_modifier [ optional_postfix_modifier ]
optional_postfix_modifier ::=
      DEFAULT ( method_name_list )
    | OPTIONAL
```

Syntax 6-36—Syntax for 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_FAMILYs/

The OPTIONAL modifier specifies the application is not required to define the referenced name. If an OPTIONAL EXTERNAL is referenced by the DPCM but was not defined by the application, the DPCM shall receive a return code of severity 2 from the reference.

6.12.5.3 INTERNAL statement

The syntax for the INTERNAL statement is given in Syntax 6-37.

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

Syntax 6-37—Syntax for INTERNAL *statement*

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

6.12.5.4 DCL to C communication

The passed argument list to C code shall consist of:

- The *Standard Structure* pointer as the first argument and named *std_struct*.
- 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*.
- 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 (see 8.11.1).

6.12.6 Calculation statements

The syntax for the calculation is given in Syntax 6-38.

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

Syntax 6-38—Syntax for calculation

6.12.6.1 CALC statement

The syntax for the CALC statement is given in Syntax 6-39.

```
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 6-39—Syntax for `CALC` *statement*

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

6.12.6.2 ASSIGN statement

The syntax for the `ASSIGN` statement is given in Syntax 6-40.

```
assign_statement ::=
    [ EXPORT ] ASSIGN ( name ) method_postfix_modifier :
    calculation_body ;
```

Syntax 6-40—Syntax for `ASSIGN` *statement*

An `ASSIGN` statement, 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 re-entrant.

6.12.6.3 DELAY statement

The `DELAY` statement calculates segment delays for a path. The syntax for the `DELAY` statement is given in Syntax 6-41.

```
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 6-41—Syntax for DELAY *statement*

6.12.6.3.1 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)
```

6.12.6.3.2 DEFAULT modifier

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 DEFAULT DELAY statement shall not have passed parameters. Since the DPCM does not model the segment, it shall not recognize the proper parameters to pass the statement.
— The DEFAULT DELAY statement shall not reference STORE variables.

6.12.6.4 SLEW statement

The SLEW statement calculates transition times for a path. The syntax for the SLEW statement is given in Syntax 6-42.

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

Syntax 6-42—Syntax for SLEW *statement*

6.12.6.4.1 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)
```

6.12.6.4.2 DEFAULT modifier

The DEFAULT modifier identifies the SLEW statement to be used in situations where 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 DEFAULT SLEW statement shall not have passed parameters. Since the DPCM does not model the segment, it shall not recognize the proper parameters to pass the statement.
— The DEFAULT SLEW statement shall not reference STORE variables.

6.12.6.5 CHECK

The syntax for the CHECK statement is given in Syntax 6-43.

```
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 6-43—Syntax for 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), that 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.

6.12.7 METHOD statement

The syntax for the METHOD statement is given in Syntax 6-44.

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

Syntax 6-44—Syntax for METHOD *statement*

DCL supports access to multiple statements, 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_FAMILYs.

6.12.7.1 Default action statement

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

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

6.12.7.2 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* (see 8.13).

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

a) The associated CELL, PIN, or PATH is first determined:
If the METHOD reference uses the scope operator CELL_DATA::, 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), the action statement is assumed to be associated with the PIN or PATH identified by the pathData field of the *Standard Structure*.

b) If an action statement was associated with the identified CELL, PIN, or PATH, that action statement shall be executed.

c) If no such action statement was found or the supplied pathData or cellData handle is zero, and a DEFAULT action statement was declared, that DEFAULT statement shall be executed.

d) If no action statement was found and no default action statement was declared, an error shall be propagated back to the calling statement.

6.13 Tables

A DCL table is a one-dimensional vector of data. Each data vector (called a *row*) shall have the same structure (i.e., the same number of data values, 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 defined at compile-time (*static* tables), or at run-time (*dynamic* tables).

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 affect on the searching for a matching row (see 6.13.4.4.2).

6.13.1 TABLEDEF statement

The syntax for the TABLEDEF statement is given in Syntax 6-45.

```
tabledef_statement ::=
      [ EXPORT ] TABLEDEF ( name )
      std_postfix_modifier tabledef_postfix_modifier :
      [ passed_clause ] qualifiers_clause
      data_clause default_clause [ key_clause ] ;
    | [ EXPORT ] TABLEDEF ( name )
      std_postfix_modifier :
      passed_clause qualifiers_clause
      data_clause key_clause default_clause ;
    | [ EXPORT ] TABLEDEF ( name )
      std_postfix_modifier :
      [ passed_clause ] qualifiers_clause [ key_clause ]
      data_clause default_clause ;
    | [ EXPORT ] TABLEDEF ( name )
      std_postfix_modifier :
      [ passed_clause ] [ key_clause ] qualifiers_clause
      data_clause default_clause ;
    | [ EXPORT ] TABLEDEF ( name )
      std_postfix_modifier DYNAMIC :
      [ passed_clause ] qualifiers_clause
      data_clause default_clause;
```

Syntax 6-45—Syntax for 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.

6.13.1.1 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 6.13.4.4.2).

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

a) Variables of type PIN shall be converted to the type STRING using DCL's implicit conversions (see 6.8.1).
b) Variables of type INTEGER shall be converted to type STRING as generated by *sprintf()* using a format of %d (section 7.9.6.1 (the sprintf function), the ISO C standard).
c) 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, ISO C standard).
d) PINLIST, VOID, derived types, and array types shall not be used in the QUALIFIERS clause.

6.13.1.2 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 (`INTEGER`, `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 `PIN`, `PINLIST`, and `VOID` shall not be allowed in the `DATA` clause of a `TABLEDEF` statement.

6.13.1.3 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.

6.13.1.4 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 is only allowed across compilation units if the `OVERRIDE` modifier is specified on the `IMPORT`ed `TABLEDEF` statement prototype (see 6.13.4.6).

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

6.13.1.5 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 6.13.4.2) in the `TABLE` procedure.

6.13.2 Table visibility rules

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

The following rules also apply to all `TABLEDEF`s:

a) A table defined by a static `TABLEDEF` shall be visible only within its compilation unit.

b) A table defined by an `EXPORT TABLEDEF` shall be visible within its compilation unit and within any other compilation unit the same `TECH_FAMILY` and an `IMPORT` of that `TABLEDEF`.

c) A compilation unit which contains a static `TABLEDEF`, an `EXPORT TABLEDEF`, or an `IMPORT`ed `TABLEDEF` that specifies the `OVERRIDE` modifier shall contain at least one `TABLE` statement with the same name as the `TABLEDEF`.

d) A compilation unit which `IMPORT`s a `TABLEDEF` and does not specify the `OVERRIDE` modifier shall not contain any `TABLE` statements with the same name as the `TABLEDEF`.

6.13.3 TABLE statement

The syntax for the TABLE statement is given in Syntax 6-46.

```
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_field { [ , ] table_data_field }
table_data_field ::= table_array_element | table_data_element
table_array_element ::= [ [ table_data_fields ] ]
table_data_element ::= string_literal | statement_name | 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 6-46—Syntax for 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.

6.13.4 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:

a) The DEFAULT record (if present) shall appear in the first table in the source and follows the rules for default rows.
b) The PROTOTYPE_RECORD shall appear in the first table encountered and shall follow all the rules for `prototype_records`.
c) 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

d) 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 6.13.1.2). 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).

6.13.4.1 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, 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.

6.13.4.2 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, 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.

6.13.4.3 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.

6.13.4.4 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.

6.13.4.4.1 String prefix operator

The string prefix operator * (see 6.9.1) can prefix a non-empty string used as a qualifier component in either a table row or a table reference.

6.13.4.4.2 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 of the 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.

6.13.4.5 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.

6.13.4.6 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 which 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 6.13.1.4). When duplicate table rows are found in separately compiled table statements the latter used the OVERRIDE modifier, the TABLE statement's data which is loaded later shall supersede the existing table data.

6.13.5 Dynamic tables

Dynamic tables are loaded into memory at run-time (using the LOAD_TABLE statement) upon 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) upon request of the DPCM.

A table shall be designated as dynamic with the DYNAMIC modifier on the appropriate TABLEDEF statement.

6.13.5.1 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:

a) 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 semi-colon (;) at the end of the table description.
b) There shall be no other data in the file except the table row data.
c) The default operator (*) shall not be used as a table row qualifier.
d) The table shall not contain a PROTOTYPE_RECORD.

6.13.5.2 Limitations

The OVERRIDE modifier and the KEY clause shall be considered illegal in a dynamic TABLEDEF statement.

6.13.6 Dynamic table manipulation

This section details dynamic table manipulation within DCL.

6.13.6.1 LOAD_TABLE statement

The syntax for the LOAD_TABLE statement is given in Syntax 6-47.

```
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 ) ( string_exp )
opt_integer_default ::= DEFAULT ( integer_expression )
```

Syntax 6-47—Syntax for 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 is received by the statement, it shall conform to the QUALIFIER and DATA clauses of the associated TABLEDEF statement (see 6.13.6.1.2).

LOAD_TABLE shall not be declared CONSISTENT.

6.13.6.1.1 Restrictions

There shall be one FILE, or FILTER. At most there shall be one instance of each of the, FILE, FILTER and PATH clauses. 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.

6.13.6.1.2 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.

6.13.6.1.3 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 non-zero value which refers to an error code.

6.13.6.1.4 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, an empty table shall be created. If the FILTER clause is not present, the FILE shall be read in as is.

6.13.6.1.5 FILTER clause

The FILTER clause shall contain a STRING which 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.

6.13.6.1.6 PATH clause

The PATH clause shall contain a STRING which designates the name of an environment variable (UNIX) or 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 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 6.14.2.4).

6.13.6.1.7 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 6.13.1.5) shall be executed and the result of that clause shall be returned to the DPCM.

6.13.6.1.8 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.

6.13.6.2 UNLOAD_TABLE statement

The syntax for the UNLOAD_TABLE statement is given in Syntax 6-48.

```
unload_table_statement ::=
    [ EXPORT ] UNLOAD_TABLE ( name )
    std_postfix_modifier : [ passed_clause ]
    TABLEDEF ( name_of_tabledef ) [ opt_file_filter_path ]
    [ DEFAULT ( expression ) ] ;
```

Syntax 6-48—Syntax for 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 6.13.6.1.7). 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 6.13.6.1), 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 non-zero 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.

6.13.6.3 ADD_ROW statement

The syntax for the ADD_ROW statement is given in Syntax 6-49.

```
add_row_statement ::=
    [ EXPORT ] ADD_ROW ( name )
    std_postfix_modifier [ opt_replace ] :
    TABLEDEF ( name_of_tabledef ) [ DEFAULT ( expression ) ];
```

Syntax 6-49—Syntax for ADD_ROW *statement*

The ADD_ROW *statement* declares *name* as a function which can be called to add a single row to a dynamic table. The ADD_ROW statement shall not be declare CONSISTENT.

6.13.6.3.1 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.

6.13.6.3.2 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. The number, order, and types of these arguments are exactly those specified by the concatenation of the variables in the QUALIFIERS and DATA clauses of the associated TABLEDEF statement.

6.13.6.3.3 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.

6.13.6.3.4 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.

6.13.6.3.5 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 non-zero value.

6.13.6.4 DELETE_ROW statement

The syntax for the DELETE_ROW statement is given in Syntax 6-50.

```
delete_row_statement ::=
    [ EXPORT ] DELETE_ROW ( name )
    std_postfix_modifier : TABLEDEF ( tabledef_statement_name)
    [ default_clause ] ;
```

Syntax 6-50—Syntax for 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.

6.13.6.4.1 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.

6.13.6.4.2 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 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.

6.13.6.4.3 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.

6.13.6.4.4 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 (see 6.13.1.5) defines the values to be returned when DELETE_ROW statement reference fails. A failure shall occur if a matching row can not 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.

6.13.7 Lookup table

The *lookup table* facilitates interpolation between distinct points. The lookup table contains a one, two, or three dimensional array of data points. For each dimension there shall exist a vector of indices. A query from a lookup table produces a lookup table row structure that matches the definition defined by the associated LUTABLEDEF. The interpolated value associated with the matrix can be computed by passing the lookup table structure to the built-in function EVALUATE.

6.13.7.1 LUTABLEDEF statement

The syntax for the LUTABLEDEF statement is given in Syntax 6-51.

lutabledef_statement ::=
[**IMPORT** | **EXPORT**] **LUTABLEDEF (** name **)**
external_postfix_modifiers [**OVERRIDE**] **:** [passed_clause]
qualifiers_clause evaluation_clause **END ;**
evaluation_clause ::=
EVALUATION (DOUBLE | **FLOAT :** evaluation_name_list **)**
evaluation_name_list ::= *evaluation*_name { **,** *evaluation*_name }

Syntax 6-51 — Syntax for LUTABLEDEF *statement*

The OVERRIDE modifier shall be valid only with the IMPORT option.

6.13.7.1.1 PASSED clause

The *passed clause* for LUTABLEDEF shall have the same syntax and semantics as the PASSED clause for the TABLEDEF statement.

6.13.7.1.2 QUALIFIERS clause

The *qualifiers clause* for LUTABLEDEF shall have the same syntax and semantics as the QUALIFIERS clause for the TABLEDEF statement.

6.13.7.1.3 Evaluation clause

The evaluation clause sets the data type and organization for the LUTABLE. The evaluation clause shall contain one, two, or three identifier names. Each of these names shall represent two associated clauses with the LUTABLE and EVALUATE built-in function corresponding to an LUTABLE with one, two or three dimensions.

a) A one dimensional array of limits for each evaluation identifier name, containing four elements: lower limit, lower warning, upper warning and upper limit.
b) A one dimensional array of indices for each evaluation identifier name. These indices represent values where the data evaluation matrix corresponding to this index is the precisely correct value.

The type, number and order of identifiers in the evaluation clause represent the type, number, and order of the passed arguments to the EVALUATE built-in function following the LUTABLE structure.

6.13.7.2 LUTABLE statement

The syntax for the LUTABLE statement is given in Syntax 6-52.

lutable_statement ::= **LUTABLE (** name **) :** lutable_statements **END;**
lutable_statements ::= lutable_statement_type { lutable_statement_type }
lutable_statement_type ::=
[limit_sequence_list] [indicies_sequence_list] data_sequence_list
limit_sequence_list ::= limit_sequence { limit_sequence }
limit_sequence ::= **LIMITS (** *evaluation_name* **) :** table_array_element **;**
indicies_sequence_list ::= indicies_sequence { indicies_sequence }
indicies_sequence ::= **INDICIES (** *evaluation_name* **) :** table_array_element **;**
data_sequence_list ::= ludata_sequence { ludata_sequence }
ludata_sequence ::= table_qualifier_list **:** table_array_element **;**

Syntax 6-52—Syntax for LUTABLE *statement*

The number of dimensions in the array shall be equal to the number of evaluation identifiers present in the evaluation clause. Each dimension of the array shall represent an identifier in the evaluation clause, in the order of the names in `evaluation_name_list`. `data_sequence_list` is a multidimensional array of values.

A standards-compliant DCL translator shall pre-process the coefficient data in LUTABLE and generate interpolation coefficients so the EVALUATE built-in function can perform its interpolation using the following rules:

a) 1-dimensional:
```
F(X) = a + bX
```
b) 2-dimensional:
```
F(X,Y) = a + bX + cY + dXY
```
c) 3-dimensional:
```
F(X,Y,Z) = a + bX + cY + dZ+ eXY + fYZ + gZX + hXYZ
```

where

`X, Y, Z` are the values supplied for `evaulation_name`(s)
`a, b, c, d, e, f, g, h` are constants (coefficients)

6.13.7.2.1 Limits sequence

In each LUTABLE there shall exist one or more limit sequences for each evaluation identifier. Each limits sequence shall remain in effect until it is superseded by the next limits sequence for that identifier. There shall exist a limits sequence for each evaluation identifier before the first data sequence is encountered. Each limits sequence shall consist of an array containing four elements. The four elements are values corresponding to `low limit`, `low warning`, `upper warning`, and `upper limit` in this order. The data element types shall be the same type as those defined in the evaluation clause of the associated LUTABLEDEF.

6.13.7.2.2 Indices sequence

There shall be one or more indices sequences for each evaluation identifier. The indices sequence shall remain in effect until superseded by another indices sequence for the same identifier. There shall exist a indices sequence for each evaluation identifier before the first data sequence. The data associated with the indices sequence is a one dimensional array which contains elements of the type defined in the evaluation clause of the associated LUTABLEDEF.

6.13.7.2.3 Data sequence

The data associated with data sequence shall be an array with dimensionality equal to the number of evaluation identifiers. The data sequence array shall have its dimensions correspond to the evaluation identifiers in the order they were seen in the evaluation clause of the associated LUTABLEDEF statement. Each dimension of the data array shall have the same number of elements as the corresponding indices array. The data elements in the array shall have the same type as defined in the associated LUTABLEDEF's evaluation clause. The qualifier list shall have the same number, type, and sequence as those in the evaluation clause LUTABLEDEF.

6.13.7.3 EVALUATE built-in function

The syntax for the EVALUATE built-in function is given in Syntax 6-53.

evaluate_builtin ::= **EVALUATE (** parameter_list **)**

Syntax 6-53—Syntax for EVALUATE *statement*

The EVALUATE *built-in function* shall compute an interpolated value from a LUTABLE row and a list of indices. The arguments to EVALUATE shall be the LUTABLE row and the list of points where the evaluation is to occur. Each value point represents its corresponding evaluation identifier in the order seen in the LUTABLEDEF. The result shall consist of an integer and a single value of the same type as the evaluation clause of the associated LUTABLEDEF.

The EVALUATE built-in function shall behave as if it had the following RESULT clause:

```
RESULT(integer: status & evaluation_clause_type)
```

where

`status` is one of the following integer value encoding whether interpolation limits were exceeded during the computation:

0 - neither warning nor limit values were exceeded
1 - at least one warning, but no limit, values was exceeded
2 - at least one limit value was exceeded

6.14 Library control statements

This section 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 *subrules 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. See 8.8.1 and 8.8.3 for details.

6.14.1 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.

6.14.1.1 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.

6.14.1.2 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.

6.14.1.3 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`.

6.14.2 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 6-54.

```
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 6-54—Syntax for SUBRULE *statement*

SUBRULE statements allow one subrule to load another subrule. Subrules are not referenced, rather they are loaded in the order they appear in a depth first order.

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.

6.14.2.1 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.

6.14.2.2 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 6.14.2.4) it encounters with that name.

By default the current working directory shall be searched if this clause is not present.

6.14.2.3 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 6.14.2.4).

By default the current working directory shall be searched if this clause is not present.

6.14.2.4 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
is 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
is 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 is 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.

6.14.3 SUBRULES statement

The syntax for the SUBRULES statement is given in Syntax 6-55.

subrules_statement ::= **SUBRULES (** name **) :** {file_or_path} **;**
file_or_path ::= [**FILE (** string_literal **)**] | [**FILE_PATH (** string_literal **)**]

Syntax 6-55—Syntax for SUBRULES statement

The SUBRULES statement shall instruct the rule loading subsystem that a separate ASCII file contains additional instructions for loading subrules (see 6.14.3.3).

6.14.3.1 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.

6.14.3.2 FILE clause

The FILE clause shall contain the name of the file that contains the instructions to control subrule loading (see 6.14.3.3). If the FILE_PATH clause is omitted, the FILE clause contain shall contain the file name, as well as the path.

6.14.3.3 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.

6.14.3.3.1 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*:

— *Comment records*

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*

Default records begin with the word `default` starting in the first character position of the line. The default record doesn't load any subrules but rather 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*

This record loads subrules, according to the rules defined in Clause 6.14.3.3.3 and Clause 6.14.3.3.4.

6.14.3.3.2 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:

a) *Rule name*

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.

b) *Table name*

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.

c) *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.

d) *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 must 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.

6.14.3.3.3 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`.

a) *File name*

The first field shall contain the file name of the rule to be loaded .

b) *Rule name*

This is the name of the environment variable (UNIX) or user variable (Windows NT) that shall contain the path for the subrule.

c) *Table name*

This is name of the environment or user variable that shall contain the path for the compiled tables associated with the subrule.

d) *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.

e) *Control parameter*

If this field contains a string, then the meta variable `CONTROL_PARM` shall be set to the value of this string.

6.14.3.3.4 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 two ways:

— Use the * default operator.
— 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.

6.14.4 TECH_FAMILY statement

The syntax for the `TECH_FAMILY` statement is given in Syntax 6-56.

tech_family_statement ::= **TECH_FAMILY (** identifier **) ;**

Syntax 6-56—Syntax for `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, etc. 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.

6.14.4.1 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.

6.14.4.2 Loading SUBRULE and SUBRULES statements

A SUBRULE (see 6.14.2) or SUBRULES (see 6.14.3) statement may only load other subrules which 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 which case the GENERIC subrule shall inherit the name of the technology-specific subrule that loaded it.

6.15 Modeling

DCL defines a flexible approach to modeling cells:

— MODELPROC (“model procedure”) statements describe the actions of a circuit with respect to timing or power. Commonly-used groups of statements can be gathered into a SUBMODEL procedure that can be called from different MODELPROCs.
— MODEL statements define which cells are described by a MODELPROC.

6.15.1 Model organization

This subsection details the MODEL statement and model name matching.

6.15.1.1 MODEL statement

The syntax for the MODEL statement is given in Syntax 6-57.

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 6-57—Syntax for `MODEL` *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 timing.
`timing`	is the timing model only.
`power`	is the power model only.

6.15.1.2 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 $\texttt{component}_{\texttt{app}}$ be the field value supplied by the application, and let $\texttt{component}_{\texttt{DPCM}}$ be the field as present in the DPCM.

It shall be an error for any $\texttt{component}_{\texttt{app}}$ to use the string prefix operator or be a literal `*`.

If the application does not have a specific value for a $\texttt{component}_{\texttt{app}}$, it shall use the empty string (`""`) for that $\texttt{component}_{\texttt{app}}$.

For each component, the following precedence rules apply to name matching:

— If an exactly-matching $\texttt{component}_{\texttt{DPCM}}$ exists, match it
— If a match exists with a $\texttt{component}_{\texttt{DPCM}}$ that uses the string prefix operator, match it
— If a $\texttt{component}_{\texttt{DPCM}}$ exists that consists of `*`, match it.

6.15.2 MODELPROC statement

The syntax for the `MODELPROC` statement is given in Syntax 6-58.

```
model_procedure ::=
    MODELPROC ( name ) modelproc_postfix_modifier :
    modelproc_statement_list END ;
modelproc_postfix_modifier ::=
     COMPLEX
    | MONOLITHIC
modelproc_statement_list ::=
    properties_statement | { submodel_statement }
submodel_statement ::=
     do_statement
    | bus_statement
    | path_statement
    | input_statement
    | output_statement
    | test_statement
    | setvar_statement
    | path_separator_stmt
```

Syntax 6-58—Syntax for `MODELPROC` *statement*

A *MODELPROC statement* ("model procedure") describes the actions of a circuit with respect to timing and/or power.

6.15.2.1 MODELPROC flow of control

`MODELPROC` statements, statement references, and embedded "C" code contained within a `MODELPROC` shall be executed in lexical order, except:

- 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 actually present in the design.
- `PATH`, `BUS`, and `TEST` statements shall skip any `PIN` pairs not found.
- `INPUT` and `OUTPUT` statements shall skip any `PIN`s not found.

6.15.2.2 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 a number of 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.

6.15.3 SUBMODEL statement

The syntax for the SUBMODEL statement is given in Syntax 6-59.

submodel_procedure ::=
SUBMODEL (name **) :** { submodel_statement }**END ;**

Syntax 6-59—Syntax for SUBMODEL *statement*

The *SUBMODEL procedure statement* enables grouping of statement sequences which may be common to multiple MODELPROCs. Submodels can only be called from a MODELPROC. SUBMODEL names shall be visible only within the compilation unit that contains the definition.

A SUBMODEL shall not contain a PROPERTIES statement.

6.15.4 Modeling statements

This subsection lists the modeling statements in DCL.

6.15.4.1 PATH_SEPARATOR statement

The syntax for the PATH_SEPARATOR statement is given in Syntax 6-60.

path_separator_stmt ::= **PATH_SEPARATOR (** string_literal **) ;**

Syntax 6-60—Syntax for PATH_SEPARATOR *statement*

The *PATH_SEPARATOR statement* defines a string which 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.

6.15.4.2 PATH statement

The syntax for the PATH statement is given in Syntax 6-61.

```
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
propagation_sequence ::=
    PROPAGATE ( edge_mode_expression ) pre_code
    [ delay_slew_methods_store_list ] post_code
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
    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 6-61—Syntax for PATH *statement*

There shall be either zero or one of each of delay, slew, store or methods clauses present in a *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:

- the statements to use for computing delay and slew values
- properties, such as the signal edges that propagate across the segment
- the 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.

6.15.4.2.1 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.

6.15.4.2.2 PATH clause

The syntax for the path_list statement is given in Syntax 6-62.

```
path_list ::= [ default_path_list ] | path_name_list
default_path_list ::= *
path_name_list ::= name {, name}
```

Syntax 6-62—Syntax for 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:

a) If no path name is specified, i.e., PATH(), the segment shall have the empty string ("") as its name.
b) If a single path name is specified, e.g., PATH(A), the segment shall be given that name.
c) If the default operator * is specified, e.g., 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.
d) If multiple path names are specified, e.g., PATH(A,B), then a separate segment shall be created for each name.

6.15.4.2.3 FROM clause

The *FROM clause* identifies the timing points where the PROPAGATE segments begin. These points can be an input pin, output pin, a node. These pins may be specified directly in the DCL source or they may 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, 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.

6.15.4.2.4 TO clause

The *TO clause* identifies the timing points where the PROPAGATE segments end. These points can be an input pin, output pin, a node. These pins may be specified directly in the DCL source or they may 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, 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.

6.15.4.2.5 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.

a) 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.

b) 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.

c) METHODS clause

The syntax for the METHODS clause is given in Syntax 6-76 (see 6.15.4.5.2).

d) STORE clause

The syntax for the STORE clause is given in Syntax 6-77 (see 6.15.4.5.3).

e) CLKFLG clause

The syntax for the CLKFLG clause is given in Syntax 6-63.

clkflg_clause ::= **CLKFLG (** string_literal **)**

Syntax 6-63—Syntax for 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 ("").

f) CKTTYPE clause

The syntax for the CKTTYPE clause is given in Syntax 6-64.

```
ckttype_clause ::= CKTTYPE ( string_literal )
```

Syntax 6-64—Syntax for 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 (" ").

6.15.4.3 BUS statement

The syntax for the BUS statement is given in Syntax 6-65.

```
bus_statement ::=
    BUS ( path_list ) : FROM ( pin_range_list ) TO ( pin_range_list )
    conditional_propagation_sequence ;
```

Syntax 6-65—Syntax for 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.

   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.

b) ANYIN and ANYOUT shall not be allowed in the Pin Range list.
c) The pin count in the FROM clause shall match the pin count in the TO clause.
d) 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.

6.15.4.4 TEST statement

The syntax for the TEST statement is given in Syntax 6-66.

```
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 6-66—Syntax for 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 the PATH statement (see 6.15.4.2).

6.15.4.4.1 compare_list clause

The syntax for the `compare_list` clause is given in Syntax 6-67.

```
compare_list ::=
      compare_clause> <edges_clause> <compare_sequence_list>
compare_sequence ::=
    <test_type_clause>
    | <checks_clause>
    | <clkflg_clause>
    | <ampersan_store_clause>
    | <ampersan_methods_clause>
    | <ckttype_clause>
    | <cycleadj_clause>
```

Syntax 6-67—Syntax for compare list

There shall be one *test_type* and *checks* clause for each *compare_list*. There may be zero or one of *clkflg*, *ckttype* 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*.

6.15.4.4.2 COMPARE clause

The syntax for the COMPARE clause is given in Syntax 6-68.

```
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 6-68—Syntax for `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.

6.15.4.4.3 EDGES clause

The syntax for the `EDGES` clause is given in Syntax 6-69.

```
edges_clause ::= EDGES ( multi_compare_edge_list )
multi_compare_edge_list ::= compare_edges_list { & compare_edges_list }
compare_edges_list ::=
    reference_signal_edget mode reference_signal_edge
reference_signal_edge ::=
      edge
    | REFERENCE ( edge )
    | SIGNAL ( edge )
```

Syntax 6-69—Syntax for `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.

6.15.4.4.4 TEST_TYPE clause

The syntax for the `TEST_TYPE` clause is given in Syntax 6-70.

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 6-70—Syntax for TEST_TYPE *clause*

The *TEST_TYPE clause* indicates to the application the type of test to be performed.

e) The TEST_TYPE argument expression designates one or two test types depending upon 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 then *test_type* shall be the same as the corresponding type of the *compare_pin_list*.

f) CYCLEADJ clause

The syntax for the CYCLEADJ clause is given in Syntax 6-71.

cycleadj_clause ::= **CYCLEADJ (** *integer*_expression **)**

Syntax 6-71—Syntax for 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.

6.15.4.4.5 CHECKS clause

The syntax for the CHECKS clause is given in Syntax 6-72.

```
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 6-72—Syntax for 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 upon 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.

6.15.4.4.6 Ampersand METHODS clause

The syntax for the ampersand METHODS clause is given in Syntax 6-73.

```
ampersand_methods_clause ::= METHODS ( methods_action_lists )
methods_action_lists ::=
      methods_action_stmt_list { & methods_action_stmt_list }
```

Syntax 6-73—Syntax for ampersand METHODS *clause*

The *ampersand 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).

6.15.4.4.7 Ampersand STORE clause

The syntax for the ampersand STORE clause is given in Syntax 6-74.

```
ampersand_store_clause ::= STORE ( ampersand_store_list )
ampersand_store_list ::= store_cache_list { & store_cache_list }
```

Syntax 6-74—Syntax for ampersand `STORE` *clause*

The *ampersand 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*.

6.15.4.5 INPUT statement

The syntax for the `INPUT` statement is given in Syntax 6-75.

```
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 6-75—Syntax for `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.

6.15.4.5.1 Propagation clause

DCL does not allow control over an individual net segment, but rather 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.

6.15.4.5.2 METHODS clause

The syntax for the `METHODS` clause is given in Syntax 6-76.

```
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 6-76—Syntax for `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.

6.15.4.5.3 STORE clause

The syntax for the `STORE` clause is given in Syntax 6-77.

store_clause ::= **STORE** **(** store_cache_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 **)** }

Syntax 6-77—Syntax for `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(coeffTbl):
   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. S*tatements* 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 6.2.3.4) to identify the information to be saved. Table 6-16 enumerates which predefined identifiers are valid, based on the type of statement containing the STORE clause.

Table 6-16—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.

6.15.4.6 OUTPUT statement

The syntax for the OUTPUT statement is given in Syntax 6-78.

output_statement ::=
OUTPUT (*pin_range*_list **)** [**:** opt_conditional_propagation_sequence] **;**

Syntax 6-78—Syntax for 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 6.10.7) 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.

6.15.4.6.1 METHODS clause

The syntax for the METHODS clause is given in Syntax 6-76 (see 6.15.4.5.2).

6.15.4.6.2 STORE clause

The syntax for the STORE clause is given in Syntax 6-77 (see 6.15.4.5.3).

6.15.4.7 DO statement

The syntax for the DO statement is given in Syntax 6-79.

```
do_statement ::=
      { DO : do_clause_list } ;
    | DO : do_when_sequence [ , OTHERWISE {do_clause_list} ] ;
do_clause_list ::=
      call_clause
    | pre_code
    | statements_clause
    | node_clause
do_when_sequence ::=
    WHEN ( logical_expression ) {do_clause_list}
    { , WHEN ( logical_expression ) {do_clause_list} }
```

Syntax 6-79—Syntax for DO *statement*

The DO statement enables the use of conditional clauses. Within these conditional clauses, the following can be specified:

a) Calls to submodel procedures
b) Nested modelproc statements
c) New timing points

A DO statement may contain zero or more occurrences of a NODE clause (see 6.15.4.7.3), atomic statement reference, embedded C code, CALL clause (see 6.15.4.7.1), and/or STATEMENTS clause (see 6.15.4.7.2).

6.15.4.7.1 CALL clause

The *CALL clause* allows the DO statements to invoke a SUBMODEL procedure statement (see 6.15.4.7). The syntax for the CALL clause is given in Syntax 6-80.

```
call_clause ::= CALL ( submodel_procedure_name )
```

Syntax 6-80—Syntax for CALL *clause*

6.15.4.7.2 STATEMENTS clause

The *STATEMENTS clause* allows the use of SETVAR, and nesting of DO statements, which in turn allows nesting of WHEN clauses. The syntax for the STATEMENTS clause is given in Syntax 6-81.

```
statements_clause ::= STATEMENTS ( modelproc_statement_list )
```

Syntax 6-81—Syntax for STATEMENTS *clause*

6.15.4.7.3 NODE clause

The syntax for the NODE clause is given in Syntax 6-82.

node_clause ::= **NODE (** name **)** [import_export_sequence]
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 6-82—Syntax for `NODE` *clause*

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. (see also "newTimingPin" on page 261, "newNetSinkPropagateSegments" on page 263, "Sample MODELPROC results" on page 264, "newNetSourcePropagateSegments" on page 264 and "Additional MODELPROC results" on page 265.

The `NODE` name, `IMPORT` clause, and `EXPORT` clause are:

— `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 which are sinks on the net associated with the argument, except the argument itself.

6.15.4.8 PROPERTIES statement

The syntax for the `PROPERTIES` statement is given in Syntax 6-83.

```
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 6-83—Syntax for 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.

6.15.4.9 SETVAR statement

The syntax for the SETVAR statement is given in Syntax 6-84.

```
setvar_statement ::= SETVAR ( name ) : conditional_result ;
```

Syntax 6-84—Syntax for 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:

— 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 6.10.3). 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 contained within an outer STATEMENTS clause.

6.16 Embedded C code

The syntax for the a C code statement is given in Syntax 6-85.

embedded_C_code ::= **%{** C_language statement **}%**

Syntax 6-85—Syntax for embedded C code statement

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 which references it is executed and the embedded C code reference is encountered.

6.17 Definition of a subrule

The syntax for a subrule is given in Syntax 6-86.

```
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
    | lutable_statement
    | lutabledef_statement
model_statement ::=
      model_statement
    | model_procedure
    | submodel_procedure
environment_control_statement ::=
      subrule_statement
    | subrules_statement
```

Syntax 6-86—Syntax for a subrule

7. Power modeling and calculation

This section describes the power modeling and calculation used in this standard. The formal syntax is described using Backus-Naur Form (BNF). The following conventions are used:

a) Lowercase words, some containing embedded underscores, are used to denote syntactic categories (terminals and non-terminals), e.g.,

literal_character_sequence

b) Boldface words are used to denote reserved keywords, operators, and punctuation marks as a required part of the syntax. These words appear in a larger font for distinction, e.g.,

MODEL SLEW => ;

c) The **::=** operator separates the two parts of a 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 (d) shows five options for a *timing_mode_operator*.

d) A vertical bar separates alternative items (use one only) unless it appears in boldface, in which case it stands for itself, e.g.,

timing_mode_operator ::= **->** | **<-** | **<->** | **<-X->** | **->X<-**

e) Square brackets enclose optional items unless it appears in boldface, in which case it stands for itself. For example:

result_definition ::= **RESULT (** [result_sequence] **)**

indicates *result_sequence* is an optional syntax item for *result_definition*, whereas

pin_range ::= pin_name **[** range_expression **]** pin_name

indicates square brackets are part of the syntax for *pin_range*.

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. Thus, the following two rules are equivalent:

comma_expression_list ::= expression { **,** expression }
comma_expression_list ::= expression | comma_expression_list **,** expression

g) Parenthesis enclose items within a group (use one only) unless it appears in boldface, in which case it stands for itself. In the following example:

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

the first set of parenthesis 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:

identifier_first_letter ::= **a-z** | **A-Z**

indicates the first letter of the identifier can be a lowercase letter (from a to z) or an uppercase letter (from A to Z).

j) If the name of any category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic information. For example, *msb*_constant_expression and *lsb*_constant_expression are equivalent to constant_expression.

The main text uses *italicized* font when a term is being defined and `monospace` font for examples, file names, and references to constants such as `0`, `1`, or `x` values.

7.1 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:

a) the AET or "All Events Trace" technique using `dpcmGetAETCellPowerWithSensitivity`.
b) the "Group" technique using `dpcmCellPowerWithState`.
c) 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 is limited or impossible.

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 which 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 7.2). 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 7.3.

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

  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.

— Static power is

  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.

7.2 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 (upon 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 which have at least one initial state modeled.

7.2.1 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.

7.2.2 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`.

7.3 Caching load and slew information

The load and slew cache is private to the DPCM. The DPCM is responsible for allocating the associated memory (upon 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.

Once the application initiates a power calculation request using:

- `dpcmGetAETCellPowerWithSensitivity`,
- `dpcmGetCellPowerWithState`, or
- `dpcmGetPinPower`,

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 which 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.

7.3.1 Loading the load and slew cache

The application shall take one of the following actions upon being called by the DPCM (via `appRegisterCellInfo`):

a) 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 a `0` 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 remains valid and `dpcmFillPinCache` only needs to be called for those pins where the data being requested is 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.

b) Return the handle of a cache which was filled during a previous power calculation request of the same cell instance or type.

7.3.2 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.

7.4 Simultaneous 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 which 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.

7.5 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.

7.6 Power calculation

The following lists 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 if 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 pin power calculation technique can be used for this instance. The application passes the pin which 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 which 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:

1) For AET and group power calculation techniques only:

i) determine whether the current pin changes are to be considered simultaneous (see 7.4). 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. (shown below in step e2).

ii) determine whether the current pin changes are to be considered a partial swing (see 7.5). 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 8.17.9.3 for the details of the data being passed into this call.

Chronological ordering of events is important in this technique since 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).

Chronological ordering of events in not required for this technique. The DPCM cannot use the chronological ordering of power calculation requests to keep a representation of previous states.

iii) `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 call back (`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.

7.7 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.

7.8 Group pin list syntax and semantics

The Group Pin List is an array of strings which 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*.

7.8.1 Syntax

The syntax for a *GroupPinString* is given in Syntax 7-1.

GroupPinString ::= ‘ group_pin_list ‘
group_pin_list ::= group_pin_name{**,** group_pin_name}
group_pin_name ::= PinName | **ALLIN** | **ALLOUT** | **ANYIN** | **ANYOUT**

Syntax 7-1—Syntax for GroupPinString

where *PinName* is a sequence of any ASCII, non-whitespace, characters except the following special characters, which shall be escaped with a proceeding backslash (\) if used (**), (’), (,).**

7.8.2 Semantics

The semantics for a *GroupPinString* are:

— A *PinName* shall not be duplicated within one *GroupPinString*.
— A *PinName* may be duplicated in other *GroupPinString*s.
— A *PinName* shall be an actual pin name, `ANYIN`, `ANYOUT`, `ALLIN`, or `ALLOUT`.
— A *PinRange* (see 6.10.7) is not allowed in a *PinName*.

7.8.2.1 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 change 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 change value, the associated condition expressions shall be evaluated.

7.8.2.2 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.

7.8.3 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'
```

7.9 Group condition list syntax and semantics

The Group Condition List is an array of strings which 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 is 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 7.11.

7.9.1 Syntax

The syntax for a *GroupConditionString* is given in Syntax 7-2.

```
GroupConditionString ::= ‘ condition_list ‘
condition_list ::= condition_expression{, condition_expression}
```

Syntax 7-2—Syntax for GroupConditionString

where *condition_expression* is defined in 7.11.

7.9.2 Semantics

The semantics for a *GroupConditionString* are:

— Each comma delimited *condition_expression* constitutes a `ConditionIndex`.
— The application shall evaluate all *condition_expression*s within the selected *GroupConditionString*. For each *condition_expression* which 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.

7.9.3 Example

The following is a sample Group Condition List.

```
GroupStateArray[0] = 'A&&B&&C&&Q=Q-1, A==A-1&&!B&&!C&&!Q'
GroupStateArray[1] = '*'
GroupStateArray[2] = '!X||!Y||Z!=Z-1,*'
```

7.10 Sensitivity list syntax and semantics

The Sensitivity List is an array of strings which 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 *PinName*s 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 8.17.9.3 for more information on how pin changes are communicated to the DPCM.

7.10.1 Syntax

The syntax for a *SensitivityPinString* is given in Syntax 7-3.

```
SensitivityPinString ::= ‘ sensitivity_pin_list ‘
sensitivity_pin_list ::= sensitivity_pin_name{, sensitivity_pin_name}
sensitivity_pin_name ::= PinName | ANYIN | ANYOUT
```

Syntax 7-3—Syntax for SensitivityPinString

where *PinName* is a sequence of any ASCII, non-whitespace, characters except the following special characters, which shall be escaped with a proceeding backslash (`\`) if used: ‘ `\` `,`

7.10.2 Semantics

The semantics for a *SensitivityPinString* are:

- A *PinName* shall not be duplicated within one *SensitivityPinString*.
- A *PinName* may be duplicated in other *SensitivityPinString*s.
- A *PinName* shall be an actual pin name, `ANYIN`, or `ANYOUT`.
- A *PinName* shall not be `ALLIN` or `ALLOUT`.
- A *PinRange* (see 6.10.7) 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.

7.10.3 Example

The following is a sample Sensitivity List:

```
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'
```

7.11 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*.

7.11.1 Syntax

The syntax for a *condition_expression* is given in Syntax 7-4.

```
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
```

Syntax 7-4—Syntax for condition_expression

where * 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* (see 7.8) and *PinNameId* is a sequence of any ASCII, non-whitespace, characters except the following special characters, which shall be escaped with a proceeding backslash (\) if used: **- ~ ” = ! ( ) . , “ \ @ & | ^**.

7.11.2 Semantics

The semantics for a *condition_expression* are detailed in the following subsections.

7.11.2.1 Semantic rules for *PinName* and *PinNameId*

The semantics for *PinName* and *PinNameId* are:

— A *PinRange* (see 6.10.7) is not allowed in a *PinName*.
— A *PinNameId* shall be a valid *PinName* for the cell being modeled.

7.11.2.1.1 Interpreting ANYIN and ANYOUT in a *condition_expression*

The *PinName* `ANYIN` has the implied `OR`ing 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)
```

7.11.2.1.2 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)
```

7.11.2.2 Semantic rules for *PinName_Identifier* (named P_{id})

The semantics for a *PinName_Identifier* are shown in Table 7-1.

Table 7-1—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.

7.11.2.3 Semantic rules for *PinName_Level* (named P_{level})

The semantics for a *PinName_Level* are shown in Table 7-2.

7.11.2.4 Semantic rules for *PinName_State* (shorthand operators)

The semantics for a *PinName_State* are shown in Table 7-3.

Table 7-2—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.

Table 7-3—PinName_State semantics

Operator	Example	Description
~	~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.
@	@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.

7.11.2.5 Condition expression labels

A *condition_expression* consisting of a double quoted string is considered a label or name which 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.

7.11.2.6 Condition expression operators

The *condition_expression* operators are detailed below in Table 7-4.

7.11.2.6.1 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`.

7.11.2.6.2 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`.

7.11.3 Condition expression operator precedence

Condition expressions are evaluated left to right. The precedence (from highest to lowest) is:

```
-1
===
~
@
```

Table 7-4—Condition expression operators

Operator	Example	Description
==	L1==L2	is TRUE when the logical expression L1 is equal to L2.
!=	L1!=L2	is TRUE when the logical expression L1 is not equal to L2.
&&	L1&&L2	is TRUE when both the logical expressions L1 and L2 are TRUE.
||	L1||L2	is TRUE when one or both the logical expressions L1 and L2 are TRUE.
^	L1^L2	is TRUE when one but not both the logical expressions L1 and L2 are TRUE.
!	!L1	is TRUE when the logical expression L1 is FALSE.

```
( )
!
&&
^
||
!=
==
```

The group operator ( ) can always be used to force groupings or expressions to override the default precedence order. See 7.11.2.1.1 and 7.11.2.1.2 for examples.

7.11.4 Condition expressions referencing pin states and transitions

A *condition_expression* can express references to states and to transitions. A *condition_expression* which 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* which 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.

7.11.5 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 which 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

```
A00 && A01 && A02 && A03
```

shall behave like

```
A00 && A01 && A02
```

if pin `A03` doesn't exist.

If both of the pins bound to a binary operator don't 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

`L&&L'`	means `L` and `L'`
`nonexistent_pin``&& L'`	means `L'`
`L &&` *`nonexistent_pin`*	means `L`
`nonexistent_pin``&&` *`nonexistent_pin`*	means nonexistent pin (except at the top, when it means `1`)
`L || L'`	means `L` or `L'`
`nonexistent_pin``|| L'`	means `L'`
`L ||` *`nonexistent_pin`*	means `L`
`nonexistent_pin``||` *`nonexistent_pin`*	means nonexistent pin (except at the top, when it means 0)
`!( L )`	means not `L`
`!(`*`nonexistent_pin`*`)`	means nonexistent pin (except at the top, where it means 0)

8. Procedural Interface (PI)

8.1 Overview

A standard Procedural Interface (PI) is used for communication between an application and a compiled Delay and Power Calculation Module (DPCM). The functions that make up a PI are defined in three different logical components in the DPCM, application, and `libdcmlr`. Each of these components may consist of more than one compiled object, which are dynamically linked at run-time, as needed.

8.1.1 DPCM

Three categories of functions result from compilation of the DCL subrules of a technology library:

— The main calculation entry points (see 8.16.3.3) 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 `dcm_rule_init`, the DPCM primary entry point (see 8.17.11.16). These entry points map directly to the `MODELPROC`, `DELAY`, `SLEW`, and `CHECK` statements coded in the DCL source.

— Explicitly named `EXPOSE` functions (see 8.16.1) defined in the DPCM are made available to the application in a name/function-pointer table after the call to `dcm_rule_init`, the DPCM entry point (see 8.17.11.16). 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 8.16.3.2) are implicitly available as a result of code generated automatically by the DCL compiler. There are no DCL statements that directly map to these functions. Each of these functions are available for *dynamic linking* to the application.

8.1.2 Application

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 `dcm_rule_init`, the DPCM primary entry point (see 8.17.11.16). 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 8.16.3.4) 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.

— Memory management (`dcmMalloc`, `dcmFree`, and `dcmRealloc`) and message printing functions (`dcmIssueMessage`) may be defined by the application to centralize handling of these tasks. These functions shall be used by both the application and the DPCM if they are defined, and if either the `dcmSetNewStorageManager` (see 8.17.11.2) or `dcmSetMessageIntercept` (see 8.17.11.14) functions are invoked by the application.

8.1.3 libdcmlr

The DPCM-independent set of initialization functions, `dcmBindRule`, `dcmSetNewStorageManager`, and `dcmSetMessageIntercept` (see 8.16.3.2), shall be available for *dynamic linking* to the application. These are the key functions that load and initialize a DPCM.

8.2 Control and data flow

A design goal of the Delay and Power Calculation System (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 8-1.

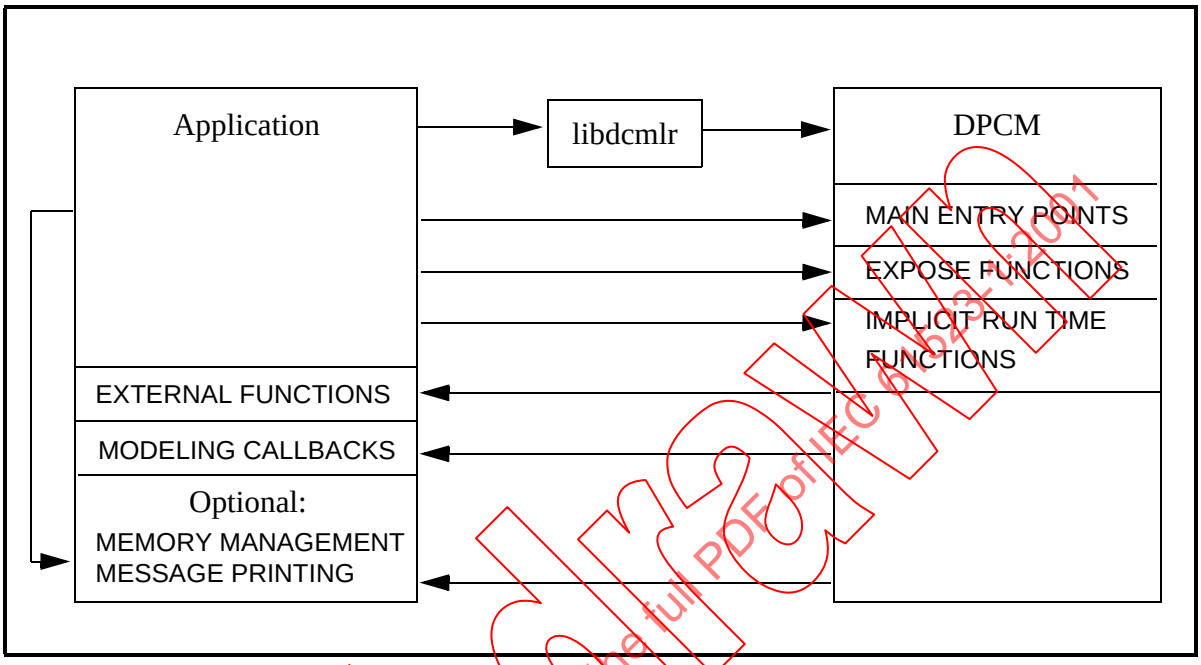


Figure 8-1 DPCM/application procedural interface

8.3 Architectural requirements

The interface requires integers to be 32 bits (or greater) to properly construct return codes (see 8.11.1).

8.4 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.

8.4.1 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 persists 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 arrays) returned by a DPCM function to an application, the DPCM assumes the data need 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 returned by a DPCM function to an application:

1) The array data is read-only.
2) If the application *locks* the array (see 8.17.10.4), the data persists until the application *unlocks* the array (see 8.17.10.5). When the array is locked, the application need not copy the array data and the DPCM shall not modify the array data.
3) If the application does not lock the array, the data only persists until the next call to any DPCM PI function. 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 locked by the application. For these calls, the application shall copy the array data if the application requires this array data to persist.
5) Both the application and the DPCM shall unlock an array only as many times as it locked it.

NOTE- see section 8.17.10.4 and section 8.17.10.5

8.4.2 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.

8.5 Application/DPCM interaction

This section 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 DCPM for power calculation (see Clause 7).

8.5.1 Application initializes message/memory handling

The DPCM allows the application to provide message handling functions (see 8.17.11.14). In addition the application is required to register memory management functions(see 8.17.11.2) which shall be used by both the DPCM and the application.

8.5.2 Application loads and initializes the DPCM

The application calls `dpcmBindRule` to dynamically load the main entry point within the DPCM. The application then calls this main entry point (see 8.8.1) to load and initialize the DPCM. 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 which implements the calls `dcmSetNewStorageManager`, `dcmSetMessageIntercept`, and/or `dcmBindRule` shall not be statically linked with the application.

8.5.3 Application requests timing models for cell instances

An application shall model a cell before calling any PI function which 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 8.17.13.1).

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.

8.5.4 Model domain issues

This subsection defines model issues across the PI.

8.5.4.1 Model domain selection

The DPCM may have one or two model domains which represent independent but related views of a technology, one for power and one for timing or one that contains both timing and power. To prevent 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 of the `EXPOSE` functions 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.

8.5.4.2 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.

8.5.5 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 8-14) and the input and output pins. Bidirectional pins, 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 section 8.17.13.6-section 8.17.13.12). 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 as well as the cell-specific value of the `cellData` pointer fields in the *Standard Structure* (see 8.13), since 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, since 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 re-use 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).

8.5.6 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 of the instances 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 8.17.12.1) and passes the following parameters using the DCL *Standard Structure*:

- a "handle" for the specific instance (`BLOCK`) (see 8.12)
- 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 8.17.13.6).
- the "to" pin (`TO_POINT`)
  This argument is a pointer to an application-created pin structure (see 8.17.13.6).
- 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 8.17.13.13)
  This argument was originally passed by the DPCM to the application during model elaboration (see 8.5.5).
- value for `cellData` pointer (`CELL_DATA`)
  This argument was originally passed by the DPCM to the application during model elaboration (see 8.5.5).

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 which originate or terminate on that pin (see 8.17.13.13 for information on interconnect delay calculation).

8.5.7 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 8.5.6).

8.6 Re-entry requirements

There are cases where 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 re-entry in order to cope with such situations. For example, a call to `appGetRLCnetworkByName` might result in a recursive call to the same function via the `dpcmAppendPinAdmittance` function.

8.7 Application responsibilities when using a DPCM

This section details the responsibilities an application shall have when using a DPCM.

8.7.1 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 8.11.2). Most PI functions require values to be initialized in one or more of the DCL *Standard Structure* fields (see 8.13).

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 which reference different cell instances.

NOTE — The number of extra *Standard Structures* required for this approach is small (generally one or two).

8.7.2 User object registration

The DPCM has a method (see 8.17.11.26) 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 which 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.

8.7.3 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 propa-

gated. 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.

8.7.3.1 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 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 is not feasible to guarantee *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.

8.7.3.2 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.

8.8 Application use of the DPCM

This section details how an application interacts with the DPCM.

8.8.1 Initialization of the DPCM

A running application first calls the function `dpcmBindRule`, which dynamically loads the `root` subrule and returns its entry point, `dcm_rule_init` (see 8.17.11.16), back to the application. Then, the application calls the `dcm_rule_init` entry point, which initiates several dynamic linking and run-time initialization processes.

Two parameters are passed to `dcm_rule_init`:

a) the address of a location for the DPCM to store a `DCMTransmittedInfo` structure pointer. The `DCMTransmittedInfo` (see 8.14) 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.

b) a pointer to a `DCM_FunctionTable` structure containing pairs of PI `EXTERNAL` names and pointers to the functions implementing them in the application. It is the application's responsibility to create this structure.

8.8.1.1 Dynamic linking

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. Undefined `EXTERNAL` entry points shall be automatically linked to a function which always returns a return code of severity `ERROR`.

`EXPOSE` entry points with the same name (originating in separate subrules) shall be linked together in a *chain*, called *expose chaining*. Expose chaining is a process which 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:

— the current EXPOSE function returns a zero (0) return code, or
— the current EXPOSE function returns a return code with severity 3 or greater, or
— the current EXPOSE function is the last function in the chain
(in which case it returns its return code back to the application).

8.8.1.2 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 TABLEs, if any, defined within the subrules are loaded into memory.
— Execution of the LATENT_EXPRESSION function.

A maximum of one (1) 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. 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:

a) The function name shall be LATENT_EXPRESSION.
b) Only one LATENT_EXPRESSION is allowed per subrule.
c) The function shall not have PASSED parameters.
d) The function shall be one of the following types: ASSIGN, CALC, INTERNAL, or EXPOSE.
e) The LATENT_EXPRESSION function may reference any other legal DCL statement.

8.8.2 Use of the DPCM

When dcm_rule_init is called, the DPCM loads the remaining subrules specified, 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 first argument to the dcm_rule_init call.

8.8.2.1 Application control

Control is returned to the application which then:

— 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; and
— 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.

8.8.2.2 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 Procedural Interface (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 8.17.13.1) prior to calling functions that require pathData or cellData *Standard Structure* fields (Table 8-14). The `MODELPROC` function (see 6.15.2) 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 8.17.13). 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.

8.8.3 Termination of DPCM

When an application is finished using the DPCM, it may call `dcmUnbindRule` (see 8.17.11.8). `dcmUnbindRule` invokes `TERMINATE_EXPRESSION`, 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.

In the case where the application comes to a normal termination (by calling `exit`) without calling `dcmUnbindRule`, 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.

8.9 DPCM library organization

This section highlights the DPCM library organization.

8.9.1 Multiple technologies

A subrule shall only load other subrules which are of the same `TECH_FAMILY` name, with the following exceptions:

- A `GENERIC` technology subrule (without a `TECH_FAMILY` name or with the name `GENERIC`) can load any other subrule, `GENERIC` or technology-specific.
- A technology-specific subrule may load a `GENERIC` subrule, in which case the loaded subrule inherits the name of the technology-specific subrule that loaded it, i.e., `GENERIC` subrules that inherit a `TECH_FAMILY` name become technology-specific.
- A `GENERIC` subrule may load a technology-specific subrule, in which case the loaded subrule retains its technology-specific characteristics.

Interactions between an application and a potentially multiple-technology DPCM are described by Table 8-1.

Table 8-1—Interaction between multiple technologies and application

Number of technologies	Technology names visible to application	Default DPCM visibility to application
1	loaded `TECH_FAMILY` name (or `GENERIC` if none specified)	everything
> 1	all loaded `TECH_FAMILY` names	`GENERIC`

8.9.2 Model names

A fully-qualified model name consists of the three strings `cell`, `cellQual`, and `modelDomain`.

8.10 DPCM error handling

When the DPCM detects errors of severity `0`, `1`, or `2` (see Table 8-2 on page 150), 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 8.8.1.1) 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)*).

8.11 C level language for EXPOSE and EXTERNAL functions

The following C language interface conventions shall be honored by applications interfacing with PI `EXTERNAL`s or `EXPOSE`s.

8.11.1 Integer return code

Functions return an integer code:

— Zero means the function completed successfully.
— Non-zero 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 8-2.

Table 8-2—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 8-3.

Table 8-3—Return code least significant bytes

Decimal range	Meaning
1 — 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.

8.11.2 The Standard Structure pointer

The DCL *Standard Structure* (see 8.13) 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 actually being used in that particular function.

8.11.3 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 8.11.1.

8.11.4 Passed arguments

The PASSED arguments (as listed in the function declaration) follow the results pointer.

Example

The following is an example of an EXTERNAL statement in DCL.

```
EXTERNAL (appGetPiModel): passed(pin: outputPin )
      result(integer: estFlag & double:capNear, capFar, Resistance );
```

The equivalent C language is:

```
typedef struct {INTEGER estFlag; DOUBLE capNear,capFar,Resistance}
                T_capResValue;

int appGetPiModel( DCM_STD_STRUCT  *std_struct,
          T_capResValue            *rtn,
          PIN                      outputPin);
```

The types for the results structure and the passed parameters shall conform to data types as shown in Table 8-4. The definitions for these data types are available in the header files `dcmwords.h` and `std_stru.h`.

8.11.5 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`.

8.11.6 Conversion to C data types

The C data types for each of the function's result and passed type names are shown in Table 8-4. The `std_stru.h` header file (see 8.18) provides these definitions.

Table 8-4—Data types defined in DCL and C

Notation	ISO C data type
`INTEGER` or `integer`	`int`
`STRING` or `string`	`char *`
`NUMBER` or `number`	`float` or `double`[a]
`DOUBLE` or `double`	`double`
`FLOAT` or `float`	`float`
`PIN` or `pin` A pointer to an application-private data structure with a first field type `char *`.	`char **`
`PINLIST` or `pinlist` A `0`-terminated array of `PIN`s.	`char ***`
`VOID` or `void` A pointer to an indeterminate data structure.	`void *`

[a]This is an ISO C float in table data and the result of `DELAY`, `SLEW`, and `CHECK` computations. In all other contexts, a `NUMBER` is an ISO C double.

8.11.7 include files

An application shall include the header files shown in Table 8-5 (as needed to compile correctly) to obtain `typedef` and other necessary declarations.

Header files shall not be modified to ensure interoperability.

Table 8-5—Header files

Header file	Definition
`dcmdebug.h`	This header file (see 8.22) contains code necessary for uniform implementation of debugging facilities in the DPCM.
`dcmintf.h`	This header file (see 8.20) contains the `FindFunction` declarations, environment variable names (UNIX) or user variable names (Windows NT), and the definition for the `DCMTransmittedInfo` structure (see 8.14).
`dcmload.h`	This header file (see 8.21) defines how subrules are loaded.
`std_macs.h`	This header file (see 8.19) 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 (see 8.18) contains the *Standard Structure* definition (see 8.13) and the `typedef`s used by many of the PI functions.

`dcmgarray.h`
`dcmpltfm.h`
`dcmstate.h`
`dcmutab.h`
	Miscellaneous header files containing defines used by the other header files.

8.12 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 8-2 illustrates the relationship between `PINLIST`s, `PIN`s, and `STRING`s

The structure of a block is identical to a structure of a pin.

.

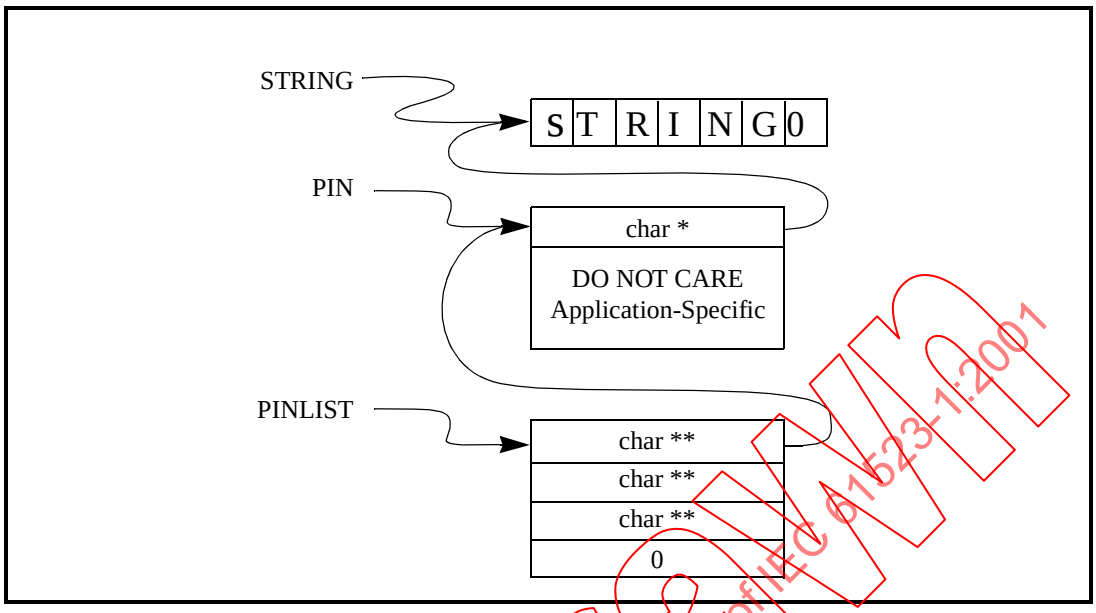


Figure 8-2 PIN and PINLIST

The application shall store enough private data in a `PIN`'s data structure to be able to answer `appGetHierPinName` (see 8.17.6.3), `appGetHierBlockName`, and `appGetHierNetName` (see 8.17.6.5).

8.13 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 8.16.4). 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 defines. 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 8-6, providing that the application names its standard structure variable "STD_STRUCT".

Table 8-6—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.

Table 8-6—Predefined macro names *(continued)*

#define (used within C source)	Description
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__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 pathDatacell 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.
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. 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. 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.

Table 8-6—Predefined macro names *(continued)*

#define (used within C source)	Description
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`. 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_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, 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, terminate, or both edges terminated.
DCM__TO_POINT	Gives access to `toPoint`. Represents the end point 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.
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.

Table 8-6—Predefined macro names *(continued)*

#define (used within C source)	Description
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`, since all the defines are of the form `(std_struct->`*XXX*`)`.

8.13.1 Alternate semantics for Standard Structure fields

Some of the 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_macs.h` header file specify the same *Standard Structure* field as `EARLY_SLEW` and `LATE_SLEW`. It is the application's responsibility to ensure the values set in the DCL *Standard Structure* are appropriate for the context of the function being called.

Table 8-7 lists the different names used for the same fields in the DCL *Standard Structure* under the corresponding context headings.

Table 8-7—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`
`toPoint`	`TO_POINT`	`SIGNAL_POINT`
`sourceMode`	`SOURCE_MODE`	`REFERENCE_MODE`
`sinkMode`	`SINK_MODE`	`SIGNAL_MODE`
`SourceEdge`	`SOURCE_EDGE`	`REFERENCE_EDGE`
`SinkEdge`	`SINK_EDGE`	`SIGNAL_EDGE`

8.13.2 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!

8.13.3 Standard Structure value restriction

Standard structure fields of type string shall not be set to the string "*" by the application.

8.14 DCMTransmittedInfo structure

This structure, as defined in the `dcmintf.h` header file (see 8.20), is used to support dynamic linking to the application of explicit `EXPOSE` functions (see 8.16.1) and implicit Calculation and Modeling functions (see 8.16.3.3 and 8.16.3.4) within the DPCM.

When the application first calls the DPCM entry point (which `dcmBindRule` 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 8.17.13.1), `delay` (see 8.17.12.1), `slew` (see 8.17.12.2), and `check` (see 8.17.12.3), followed by a table of pointers to functions, each paired with an `EXPOSE` name from the standard interface.

The application can use `dcmFindFunction` (see 8.17.11.9) or dcmQuiteFindFunction (see 8.17.11.11) by name the corresponding `EXPOSE` function pointer in the `DCMTransmittedInfo` structure. The four implicit functions can be called directly by the application.

8.15 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.

8.16 PI functions summary

This section 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`).

8.16.1 Expose functions

The following table 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 section defines the `EXPOSE` functions used in this standard (See Table 8-8.)

Table 8-8—EXPOSE functions

Section	EXPOSE function	Description
8.17.3.11	`dpcmAddWireLoadModel`	Add a custom wire load model (write into) DPCM.
8.17.9.5	`dpcmAETGetSettlingTime`	Returns settling time for a list of pins when using the AET power calculation method.
8.17.9.6	`dpcmAETGetSimultaneousSwitchTime`	Returns simultaneous switch times for a list of pins when using the AET power calculation method.

Table 8-8—EXPOSE functions *(continued)*

Section	EXPOSE function	Description
8.17.3.22	`dpcmAppendPinAdmittance`	Adds the admittance of a receiver pin to the RLC tree for the interconnect.
8.17.3.20	`dpcmCalcCeffective`	Returns the effective capacitance for the load seen by the specified driving pin.
8.17.9.9	`dpcmCalcPartialSwingEnergy`	Returns the energy of a partial logic swing for AET or group power calculation methods.
8.17.3.18	`dpcmCalcPiModel`	Calculates a Pi (π) model for the interconnect driven by the specified pin drives.
8.17.3.19	`dpcmCalcPolesAndResidues`	Calculates poles and residues for the specified load.
8.17.8.11	`dpcmDebug`	Allows the application to control the debug level setting if the DPCM is compiled with debugging enabled.
8.17.3.23	`dpcmDeleteRLCnetwork`	Deletes previously created RLC network.
8.17.8.14	`dpcmFillPinCache`	Supplies the load and slew of the specified pin.
8.17.8.15	`dpcmFreePinCache`	Frees the load and slew cache.
8.17.9.3	`dpcmGetAETCellPowerWithSensitivity`	Returns static power per rail, dynamic energy per rail, total energy, and total static power.
8.17.5.2	`dpcmGetBaseFunctionalMode`	Returns the index number of the default functional mode for the cell specified in the *Standard Structure*.
8.17.5.13	`dpcmGetBaseOpRange`	Returns the name of the base operating range modeled in the DPCM.
8.17.5.7	`dpcmGetBaseRailVoltage`	Returns a default voltage value for the specified rail.
8.17.5.12	`dpcmGetBaseTemperature`	Returns the base operating temperature for the library.
8.17.5.10	`dpcmGetBaseWireLoadModel`	Returns the index number of the default wire load model for the library.
8.17.4.1	`dpcmGetCapacitanceLimit`	Returns the capacitance limits for the specified pin.
8.17.6.1	`dpcmGetCellList`	Return a list of cells that exist in the library.
8.17.1.12	`dpcmGetCellIOlists`	Return inputs, outputs, and bidirectional pins for the given cell.
8.17.9.1	`dpcmGetCellPowerInfo`	Returns the power calculation methods supported by the DPCM.
8.17.9.2	`dpcmGetCellPowerWithState`	Returns power and energy for the group power calculation methodology.
8.17.5.4	`dpcmGetControlExistence`	Returns information which controls the existence of the segment identified by `pathData`.
8.17.1.7	`dpcmGetDefCellSize`	Returns cell's size metric for interconnect load estimation.
8.17.2.6	`dpcmGetDefPortCapacitance`	Returns default capacitance for a chip primary output.
8.17.2.5	`dpcmGetDefPortSlew`	Returns default slew for a chip primary input.
8.17.2.2	`dpcmGetDelayGradient`	Returns the rate of change of the delay at the specified point.
8.17.1.8	`dpcmGetEstLoadCapacitance`	Returns an estimated load capacitance for the specified pin.

Table 8-8—EXPOSE functions *(continued)*

Section	EXPOSE function	Description
8.17.2.4	`dpcmGetEstimateRC`	Returns an RC value for a specified pin pair. Called when the application does not know the RC value for the current pin pair.
8.17.1.9	`dpcmGetEstWireCapacitance`	Returns an estimated interconnect capacitance for the interconnect which the specified pin drives.
8.17.1.10	`dpcmGetEstWireResistance`	Returns the estimated wire resistance to which the specified pin is connected.
8.17.5.1	`dpcmGetFunctionalModeArray`	Returns the functional mode names for the cell specified in the *Standard Structure*.
8.17.9.7	`dpcmGroupGetSettlingTime`	Returns settling time for a list of pins when using the group power calculation method.
8.17.9.8	`dpcmGroupGetSimultaneousSwitchTime`	Returns simultaneous switch time for a list of pins when using the group power calculation method.
8.17.9.13	`dpcmGetNetEnergy`	Returns net energy.
8.17.5.14	`dpcmGetOpRangeArray`	Returns the operating range names that are modeled in the DPCM.
8.17.1.11	`dpcmGetPinCapacitance`	Returns the cell's pin capacitance for the specified pin.
8.17.9.4	`dpcmGetPinPower`	Returns static power per rail, dynamic energy per rail, total energy and total static power for a specific pin state change.
8.17.5.6	`dpcmGetRailVoltageArray`	Returns the voltage rail names that are modeled in the DPCM.
8.17.8.6	`dpcmGetRuleUnitToFarads`	Returns the basic units of capacitance assumed by the technology library.
8.17.8.7	`dpcmGetRuleUnitToHenries`	Returns the basic units of inductance assumed by the technology library.
8.17.8.9	`dpcmGetRuleUnitToJoules`	Returns the basic units of energy assumed by the technology library.
8.17.8.5	`dpcmGetRuleUnitToOhms`	Returns the basic units of resistance assumed by the technology library.
8.17.8.4	`dpcmGetRuleUnitToSeconds`	Returns the basic units of time assumed by the technology library.
8.17.8.8	`dpcmGetRuleUnitToWatts`	Returns the basic units of power assumed by the technology library.
8.17.2.3	`dpcmGetSlewGradient`	Returns the rate of change of the slew at the specified point.
8.17.4.2	`dpcmGetSlewLimit`	Returns the slew limit for the specified pin.
8.17.7.1	`dpcmGetThresholds`	Returns voltage and transition and delay points.
8.17.8.12	`dpcmGetVersionInfo`	Returns the version identification for the technology library and with which version of P1481 the library is compliant.
8.17.3.12	`dpcmGetWireLoadModel`	Returns wire load model to application.
8.17.9.11	`dpcmFreeStateCache`	Frees the state cache who's handle is passed in.
8.17.5.9	`dpcmGetWireLoadModelArray`	Returns the wire load model names modeled in the DPCM.

Table 8-8—EXPOSE functions *(continued)*

Section	EXPOSE function	Description
8.17.3.10	`dpcmGetWireLoadModelForBlockSize`	Returns the index number of the appropriate wire load model given a specified area.
8.17.4.3	`dpcmGetXovers`	Returns the capacitance limits for cell drive strengths of the cell to which the specified pin is connected.
8.17.8.13	`dpcmHoldControl`	Returns a signal from the DPCM that allows hold control to be performed.
8.17.8.10	`dpcmIsSlewTime`	Returns the units for calculated slews as absolute time or rate of change.
8.17.9.10	`dpcmSetInitialState`	Set the initial state for a cell.
8.17.5.5	`dpcmSetLevel`	Instructs the DPCM to perform calculations for performance (execution speed) or accuracy, and for the scope for derating supported by the application.
8.17.3.21	`dpcmSetRLCmember`	A function that is called to pass `R`, `L`, `C` and M elements within the specified interconnect.
8.17.5.17	`dpcmGetTimingStateArray`	Returns an array of strings which represent the valid states for the given segment.

8.16.2 External functions

These functions are defined in the application and declared by the DPCM via the `EXTERNAL` function, as shown in Table 8-9.

Table 8-9—EXTERNAL functions

Section	EXTERNAL function	Description
8.17.3.4	`appForEachParallelDriverByName`	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).
8.17.3.3	`appForEachParallelDriverByPin`	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).
8.17.3.14	`appGetCeffective`	Returns the value of C-effective of the load seen by the specified driver.
8.17.6.2	`appGetCellName`	Returns the cell name to which the specified pin is connected.
8.17.5.3	`appGetCurrentFunctionalMode`	Returns the current functional mode of the cell instance identified in the *Standard Structure*.
8.17.5.16	`appGetCurrentOpRange`	Returns the current operating range.
8.17.5.8	`appGetCurrentRailVoltage`	Returns a voltage value for the specified rail index number.
8.17.5.15	`appGetCurrentTemperature`	Returns the desired temperature value to be used for calculations.

Table 8-9—EXTERNAL functions *(continued)*

Section	EXTERNAL function	Description
8.17.5.18	`appGetCurrentTimingState`	Returns the current state or condition.
8.17.5.11	`appGetCurrentWireLoadModel`	Returns the current wire load model.
8.17.8.1	`appGetExternalStatus`	Returns whether and to what degree the application implemented an EXTERNAL
8.17.6.4	`appGetHierBlockName`	Returns the hierarchical instance name for the instance to which the specified pin is connected.
8.17.6.5	`appGetHierNetName`	Returns the hierarchical interconnect name for the interconnect to which the specified pin is connected.
8.17.6.3	`appGetHierPinName`	Returns the hierarchical pin name for the specified pin.
8.17.3.17	`appGetInstanceCount`	Returns the cell instance count for the interconnect region containing the driving the specified pin.
8.17.3.2	`appGetNumDriversByName`	Returns the number of source (driving) pins (including bidirectional pins) on the interconnect to which the named pin is connected.
8.17.3.1	`appGetNumDriversByPin`	Returns the number of source (driving) pins (including bidirectional pins) on the interconnect to which the specified pin is connected.
8.17.3.6	`appGetNumPinsByName`	Returns the total number of pins on the interconnect to which the named pin is connected.
8.17.3.5	`appGetNumPinsByPin`	Returns the total number of pins on the interconnect to which the specified pin is connected.
8.17.3.8	`appGetNumSinksByName`	Returns the number of `sinks` (including bidirectional pins) on the interconnect to which the named pin is connected.
8.17.3.7	`appGetNumSinksByPin`	Returns the number of `sinks` (including bidirectional pins) on the interconnect to which the specific pin is connected.
8.17.3.9	`appGetPiModel`	Returns the π model capacitance and resistance values for the interconnect to which the specified pin is connected.
8.17.3.13	`appGetPolesAndResidues`	Returns poles and residues for the specified load.
8.17.2.1	`appGetRC`	Returns the RC value for the specified pin pair.
8.17.8.3	`appGetResource`	Returns a user-supplied value for an arbitrary keyword passed from the DPCM.
8.17.3.15	`appGetRLCnetworkByPin`	Returns RLC elements representing the interconnect to which the specified pin is connected.
8.17.3.16	`appGetRLCnetworkByName`	Returns the RLC elements representing the interconnect which the named pin is connected.
8.17.1.6	`appGetSourcePinCapacitanceByName`	Returns the sum of capacitances on all `sourcePins` (including bidirectional pins) on the interconnect to which the named pin is connected.
8.17.1.5	`appGetSourcePinCapacitanceByPin`	Returns the sum of capacitances on all `sourcePins` (including bidirectional pins) on the interconnect to which the specified pin pointer is connected.
8.17.7.2	`appGetThresholds`	Returns voltage and transition and delay points.

Table 8-9—EXTERNAL functions *(continued)*

Section	EXTERNAL function	Description
8.17.9.12	`appGetStateCache`	Retrieves the cache handle associated with a cell instance and created with the call dpcmSetInitialState
8.17.1.2	`appGetTotalLoadCapacitanceByName`	Returns the total capacitance (sum of capacitance on all pins plus wire capacitance) for the interconnect to which the named pin is connected.
8.17.1.1	`appGetTotalLoadCapacitanceByPin`	Returns the total capacitance (sum of capacitance on all pins plus wire capacitance) for the interconnect to which the specified pin is connected.
8.17.1.4	`appGetTotalPinCapacitanceByName`	Returns the total pin capacitance (sum of capacitance on all pins on the interconnect) for the interconnect to which the named pin is connected.
8.17.1.5	`appGetTotalPinCapacitanceByPin`	Returns the total pin capacitance (sum of capacitance on all pins on the interconnect) for the interconnect to which the specified pin is connected.
8.17.8.2	`appGetVersionInfo`	Returns which version of P1481 the application is compliant.
8.17.8.16	`appRegisterCellInfo`	This call allows the application to supply load and slew information.

8.16.3 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.

8.16.3.1 libdcmlr

Table 8-10 lists the `libdcmlr` functions; calls to these functions shall precede the use of any other DPCM calls.

Table 8-10—libdcmlr functions

Section	libdcmlr function	Description
8.17.11.6	`dcmBindRule`	Loads and links the specified DCL subrule and returns the initialization entry point.
8.17.11.14	`dcmSetMessageIntercept`	Allows the application to supply a message intercepter, which controls the printing of DPCM messages.
8.17.11.2	`dcmSetNewStorageManager`	Allows the application to assert its storage management system on the DPCM.

8.16.3.2 run-time library

Once a DPCM is loaded, various information query and setup calls may be made. These functions shall be dynamically linkable to the application (see Table 8-11). Run-time functions shall be separate from the DPCM so multiple subrules can be linked without conflict.

Table 8-11—Run-time library

Section	Initialization function	Description
8.17.11.7	`dcmAddRule`	Adds additional DCL subrules to the DPCM after `dcmBindRule` has been called.
8.17.11.1	`dcmCellList`	Returns the list of cell names in the current library.
8.17.11.10	`dcmFindAppFunction`	Determines whether the application defined the indicated `EXTERNAL` function.
8.17.11.9	`dcmFindFunction`	Returns a pointer to the function matching the specified name. Issues an error message if the specified function name cannot be found.
8.17.11.4	`dcmFree`	Instructs the DPCM to free memory using the storage management function currently in effect.
8.17.11.15	`dcmIssueMessage`	Instructs the DPCM to print a message using the `dcmSetMessageIntercept` currently in effect.
8.17.11.12	`dcmMakeRC`	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`).
8.17.11.3	`dcmMalloc`	Instructs the DPCM to allocate memory using the storage management function currently in effect.
8.17.11.11	`dcmQuietFindFunction`	Returns a pointer to the function matching the specified name. No error message is issued if the specified function name cannot be found.
8.17.11.5	`dcmRealloc`	Instructs the DPCM to reallocate memory using the storage management function currently in effect.
8.17.11.8	`dcmUnbindRule`	Unloads the DPCM from memory and releases any memory it may have used.
8.17.11.22	`dcm_freeAllTechs`	Frees the technology list created by `dcm_getAllTechs`.
8.17.11.21	`dcm_getAllTechs`	Returns a list of all technologies loaded as part of the current DPCM.
8.17.11.20	`dcm_getTechnology`	Returns the technology name where the *Standard Structure* is mapped.
8.17.11.23	`dcm_isGeneric`	Indicates whether the current *Standard Structure* is pointing to a generic technology.
8.17.11.24	`dcm_mapNugget`	Returns the current technology mapping structure.
8.17.11.26	`dcm_registerUserObject`	Registers a pointer to a user-defined object with the *Standard Structure* so that it may be deleted by `dcm_DeleteRegisteredUserObjects` or `dcm_DeleteOneUserObject`.

Table 8-11—Run-time library *(continued)*

Section	Initialization function	Description
8.17.11.16	`dcm_rule_init`	Entry called to initialize the DPCM previously loaded.[a]
8.17.11.19	`dcm_setTechnology`	Set the technology mapping in the *Standard Structure* to the specified technology name.
8.17.11.25	`dcm_takeMappingOfNugget`	Sets the *Standard Structure* to point to the technology contained in the nugget.
8.17.11.27	`dcm_DeleteRegisteredUserObjects`	Deletes the user objects associated with the specified *Standard Structure*.
8.17.11.28	`dcm_DeleteOneUserObject`	Deletes a single user object that was registered to the specified *Standard Structure*.
8.17.11.13	`dcmHardErrorRC`	Returns the constructed return code "hard error rc"
8.17.11.17	`DCM_new_DCM_STD_STRUCT`	Allocates and initializes a new standard structure
8.17.11.18	`DCM_delete_DCM_STD_STRUCT`	Deletes a previously allocated standard structure
8.17.10.1	`dcm_copy_DCM_ARRAY`	Copies the contents from one DCM_ARRAY to another DCM_ARRAY
8.17.10.2	`dcm_new_DCM_ARRAY`	Allocates a new DCM_ARRAY
8.17.10.3	`dcm_sizeof_DCM_ARRAY`	Calculates the size of a DCM_ARRAY
8.17.10.4	`dcm_lock_DCM_ARRAY`	Locks a DCM_ARRAY
8.17.10.5	`dcm_unlock_DCM_ARRAY`	Unlocks a DCM_ARRAY
8.17.10.6	`dcm_getNumDimensions`	Returns the number of dimensions of a DCM_ARRAY
8.17.10.7	`dcm_getNumElementsPer`	Returns the number of elements in each dimension of a DCM_ARRAY
8.17.10.8	`dcm_getNumElements`	Returns the number of elements in a specific dimension of a DCM_ARRAY
8.17.10.9	`dcm_getElementType`	Returns the element type contained in a DCM_ARRAY
8.17.10.10	`dcm_arraycmp`	Tests to see if two DCM_ARRAYs have the identical contents

[a] *dcm_rule_init* is invoked with the function pointer returned by `dcmBindRule` (see 8.17.11.6) rather than by a call to that literal name.

8.16.3.3 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 8-12.

8.16.3.4 Modeling functions

An application's call to `modelSearch` (see 8.17.13.1) 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 `dcm_rule_init` (see 8.17.11.16). The application shall define all of the functions described in Table 8-13.

Table 8-12—Calculation functions

Section	Calculation function[a]	Description
8.17.12.3	`check`	Calculates values to be used for timing checks for the specified timing arc.
8.17.12.1	`delay`	Calculates the delay for the specified timing arc.
8.17.12.2	`slew`	Calculates the slew for the specified timing arc.

[a]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 dcm_`rule_init` (see 8.17.11.16).

Table 8-13—Modeling functions

Section	Modeling function	Description
8.17.13.1	`modelSearch`	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.
8.17.13.12	`newAltTestSegment`	Called by the DPCM to create a timing check arc for the specified from-to pin pair.
8.17.13.7	`newDelayMatrixRow`	Called by the DPCM to describe the propagation characteristics for arcs created by `PATH`, `BUS`, `INPUT`, `OUTPUT`, `DO: NODE IMPORT`, and `DO: NODE EXPORT`.
8.17.13.8	`newNetSinkPropagateSegments`	Called by the DPCM to create delay arcs to a specified pin from all sources.
8.17.13.9	`newNetSourcePropagateSegments`	Called by the DPCM to create delay arcs from a specified pin to all loads (sinks).
8.17.13.10	`newPropagateSegment`	Called by the DPCM to creates a delay arc for a specified from-to pin pair.
8.17.13.11	`newTestMatrixRow`	Called by the DPCM to describe the propagation characteristics for timing arcs created by DCL `TEST` statements.
8.17.13.6	`newTimingPin`	Called by the DPCM to create storage for a timing node internal to a cell.

8.16.4 PI function table description

The description of each PI functions listed in 8.17 includes a two-part table and a function description.

The first part of the table 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. The example table below (see Figure 8-3) and the subsections following it describe the different characteristics of a PI function.

Function name	Arguments	Result	Standard Structure fields
The interface function name. For application functions, this may be different from the actual name declared in the code (see 8.16.2).	An argument is a value passed by the caller to the function.	A result is a value or values returned through a structure pointer passed in as an argument	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.

Figure 8-3 PI function table example

8.16.4.1 Arguments

When there is duplication of data for arguments of functions, the passed parameters take precedence over data contained in the *Standard Structure* field.

8.16.4.2 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 8.7.1).

Common semantics for values are summarized in Table 8-14. 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 8-14—Standard Structure field semantics

Value	Description
`block`	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 `FROM_POINT` or `TO_POINT` in other fields of the `std_struct`.
`CellName`	*CellName* represents the following three fields in the *Standard Structure*: `cell`, `cellQual`, and `modelDomain`. These three fields together uniquely qualify a particular model in the DPCM.
`cellData` `pathData`	Pointers to the current `PATH` and `CELL` (returned by the DPCM to the application at `modelSearch` 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 `METHOD`s for that PI call.
`calcMode`	*calcMode* defines whether the computation is for the best case, worse case, or nominal. All PI calls which mention `calcMode` in the list of *Standard Structure* fields shall return a best case, worse case, or nominal result based on the value of `calcMode`.

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`).

8.16.4.3 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 6).

8.16.4.4 C syntax

Functions with one or more `Result` value(s) are declared with a pointer to a specially declared `struct`.

8.17 PI function descriptions

This section details the DPCS standard PI functions. Each DCL statement can be accessed through a PI function.

8.17.1 Interconnect loading related functions

This subsection describes the interconnect loading related functions.

8.17.1.1 appGetTotalLoadCapacitanceByPin

Function name	Arguments	Result	Standard Structure fields
appGetTotalLoadCapacitanceByPin	Pin pointer	Total load capacitance	calcMode

DCL syntax

```
EXTERNAL(appGetTotalLoadCapacitanceByPin):
passed(pin: outputPin)
result(double: loadCapByPin);
```

C syntax

```
typedef struct {
  DOUBLE loadCap;
} T_TotalLoadCapByPin;

int appGetTotalLoadCapacitanceByPin
  (DCM_STD_STRUCT      *std_struct,
   T_TotalLoadCapByPin *rtn,
   PIN                 outputPin
  );
```

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.

8.17.1.2 appGetTotalLoadCapacitanceByName

Function name	Arguments	Result	Standard Structure fields
appGetTotalLoadCapacitanceByName	Pin name	Total load capacitance	block calcMode

DCL syntax

```
EXTERNAL(appGetTotalLoadCapacitanceByName):
passed(string: outputPin)
result(double: loadCapByName);
```

C syntax

```
typedef struct {
  DOUBLE loadCap;
} T_TotalLoadCapByName;

int appGetTotalLoadCapacitanceByName
  (DCM_STD_STRUCT       *std_struct,
   T_TotalLoadCapByName *rtn,
   STRING               outputPin
  );
```

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.

8.17.1.3 appGetTotalPinCapacitanceByPin

Function name	Arguments	Result	Standard Structure fields
appGetTotalPinCapacitanceByPin	Pin pointer	Total pin capacitance	calcMode

DCL syntax

```
EXTERNAL(appGetTotalPinCapacitanceByPin):
passed(pin: outputPin)
result(double: totalPinCapByPin);
```

C syntax

```
typedef struct {
  DOUBLE totalPinCap;
} T_TotalPinCapByPin;

int appGetTotalPinCapacitanceByPin
  (DCM_STD_STRUCT     *std_struct,
   T_TotalPinCapByPin *rtn,
   PIN                outputPin
  );
```

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.

8.17.1.4 appGetTotalPinCapacitanceByName

Function name	Arguments	Result	Standard Structure fields
appGetTotalPinCapacitanceByName	Pin name	Total pin capacitance	block calcMode

DCL syntax

```
EXTERNAL(appGetTotalPinCapacitanceByName):
passed(string: outputPin)
result(double: totalPinCapByName);
```

C syntax

```
typedef struct {
  DOUBLE totalPinCap;
} T_TotalPinCapByName;

int appGetTotalPinCapacitanceByName
  (DCM_STD_STRUCT      *std_struct,
   T_TotalPinCapByName *rtn,
   STRING              outputPin
  );
```

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.

8.17.1.5 appGetSourcePinCapacitanceByPin

Function name	Arguments	Result	Standard Structure fields
appGetSourcePinCapacitanceByPin	Driver (source) pin pointer	Total sourcePin capacitance	calcMode

DCL syntax

```
EXTERNAL(appGetSourcePinCapacitanceByPin):
passed(pin: sourcePin)
result(double: sourcePinCapByPin);
```

C syntax

```
typedef struct {
  DOUBLE sourcePinCap;
} T_sourcePinCapacitanceByPin;

int appGetSourcePinCapacitanceByPin
  (DCM_STD_STRUCT              *std_struct,
   T_sourcePinCapacitanceByPin *rtn,
   PIN                         sourcePin
  );
```

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).

8.17.1.6 appGetSourcePinCapacitanceByName

Function name	Arguments	Result	Standard Structure fields
appGetSourcePinCapacitanceByName	Driver (source) pin name	Total driver (source) pin capacitance	block calcMode

DCL syntax

```
EXTERNAL(appGetSourcePinCapacitanceByName):
passed(string: sourcePin)
result(double: sourcePinCapByName);
```

C syntax

```
typedef struct {
  DOUBLE sourcePinCap;
} T_sourcePinCapacitanceByName;

int appGetSourcePinCapacitanceByName
  (DCM_STD_STRUCT                *std_struct,
   T_sourcePinCapacitanceByName *rtn,
   STRING                        sourcePin
  );
```

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).

8.17.1.7 dpcmGetDefCellSize

Function name	Arguments	Result	Standard Structure fields
dpcmGetDefCellSize		Cell's size metric for interconnect load estimation	CellName

DCL syntax

```
EXPOSE(dpcmGetDefCellSize):
result(double: cellSize);
```

C syntax

```
typedef struct {
  DOUBLE  cellSize;
} T_defCellSize;

int dpcmGetDefCellSize
  (DCM_STD_STRUCT *std_struct,
   T_defCellSize  *rtn
  );
```

Returns the cell's size metric for interconnect load estimation.

8.17.1.8 dpcmGetEstLoadCapacitance

Function name	Arguments	Result	Standard Structure fields
dpcmGetEstLoadCapacitance	Pin pointer	Estimated interconnect load capacitance	CellName calcMode pathData (timing-pin-specific) cellData (timing)

DCL syntax

```
EXPOSE(dpcmGetEstLoadCapacitance):
passed(pin: outputPin)
result(double: estLoadCap);
```

C syntax

```
typedef struct {
  DOUBLE estLoadCap;
} T_estLoadCap;

int dpcmGetEstLoadCapacitance
  (DCM_STD_STRUCT *std_struct,
   T_estLoadCap   *rtn,
   PIN            outputPin
  );
```

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.

8.17.1.9 dpcmGetEstWireCapacitance

Function name	Arguments	Result	Standard Structure fields
dpcmGetEstWireCapacitance	Pin pointer	Estimated interconnect wire capacitance	CellName calcMode pathData (timing-pin-specific) cellData (timing) block

DCL syntax

```
EXPOSE(dpcmGetEstWireCapacitance):
passed(pin: outputPin)
result(double: estWireCap);
```

C syntax

```
typedef struct {
  DOUBLE estWireCap;
} T_estWireCap;

int dpcmGetEstWireCapacitance
  (DCM_STD_STRUCT *std_struct,
   T_estWireCap   *rtn,
   PIN            outputPin
  );
```

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.

8.17.1.10 dpcmGetEstWireResistance

Function name	Arguments	Result	Standard Structure fields
dpcmGetEstWireResistance	Pin pointer	Estimated interconnect wire resistance	CellName calcMode pathData (timing-pin-specific) cellData (timing) block

DCL syntax

```
EXPOSE(dpcmGetEstWireResistance):
passed(pin: outputPin)
result(double: estWireResistance);
```

C syntax

```
typedef struct {
  DOUBLE estWireResistance;
} T_estWireResistance;

int dpcmGetEstWireResistance
  (DCM_STD_STRUCT      *std_struct,
   T_estWireResistance *rtn,
   PIN                 outputPin
  );
```

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.

8.17.1.11 dpcmGetPinCapacitance

Function name	Arguments	Result	Standard Structure fields
dpcmGetPinCapacitance	Pin name	Rise pin capacitance Fall pin capacitance	block CellName calcMode pathData (timing-pin-specific) cellData (timing) block

DCL syntax

```
EXPOSE(dpcmGetPinCapacitance):
passed(string: pinName)
result(double: risePinCap, fallPinCap);
```

C syntax

```
typedef struct
  {DOUBLE risePinCap, fallPinCap;}
   T_riseFallCap;
int dpcmGetPinCapacitance
  (DCM_STD_STRUCT *std_struct,
   T_riseFallCap  *rtn,
   STRING         pinName
  );
```

Returns the pin capacitance for the passed *pinName* of the *cellName* in the *Standard Structure*.

8.17.1.12 dpcmGetCellIOlists

Function name	Arguments	Result	Standard Structure fields
dpcmGetCellIOlists		Input pins Output pins Bidirectional pins	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
  (DCM_STD_STRUCT *std_struct,
   T_IO_results   *rtn
  );
```

Returns the names of the input, output, and the bi-directional 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, bi-directionals).

8.17.2 Interconnect delay related functions

This subsection describes the interconnect delay related functions.

8.17.2.1 appGetRC

Function name	Arguments	Result	Standard Structure fields
appGetRC	Driver (source) pin pointer Receiver (sink) pin pointer	Interconnect RC delay	block cellName calcMode

DCL syntax

```
EXTERNAL(appGetRC):
passed(pin: fromPin,toPin)
result(double: netSegRC);
```

C syntax

```
typedef struct {
  DOUBLE rcDelay;
} T_rc;

int appGetRC
  (DCM_STD_STRUCT *std_struct,
   T_rc           *rtn,
   PIN            fromPin,
   PIN            toPin
  );
```

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 8.17.2.4) to get an estimated value for the RC.

8.17.2.2 dpcmGetDelayGradient

Function name	Arguments	Result	Standard Structure fields
dpcmGetDelayGradient		Rate of change of delay	block

DCL syntax

```
EXPOSE(dpcmGetDelayGradient):
result(double: LateRateofChangeWithRespectToOutputCapacitance,
               LateRateofChangeWithRespectToInputSlew,
               EarlyRateofChangeWithRespectToOutputCapacitance,
               EarlyRateofChangeWithRespectToInputSlew);
```

C syntax

```
typedef struct
  {DOUBLE LateRateofChangeWithRespectToOutputCapacitance,
          LateRateofChangeWithRespectToInputSlew,
          EarlyRateofChangeWithRespectToOutputCapacitance,
          EarlyRateofChangeWithRespectToInputSlew;}
  T_RecDel;
int dpcmGetDelayGradient
  (DCM_STD_STRUCT *std_struct,
   T_RecDel       *rtn
  );
```

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.

8.17.2.3 dpcmGetSlewGradient

Function name	Arguments	Result	Standard Structure fields
dpcmGetSlewGradient		Rate of change of slew	block

DCL syntax

```
EXPOSE(dpcmGetSlewGradient):
result(double: LateRateofChangeWithRespectToOutputCapacitance,
               LateRateofChangeWithRespectToInputSlew,
               EarlyRateofChangeWithRespectToOutputCapacitance,
               EarlyRateofChangeWithRespectToInputSlew);
```

C syntax

```
typedef struct
  {DOUBLE LateRateofChangeWithRespectToOutputCapacitance,
          LateRateofChangeWithRespectToInputSlew,
          EarlyRateofChangeWithRespectToOutputCapacitance,
          EarlyRateofChangeWithRespectToInputSlew;}
  T_RecSlew;
int dpcmGetSlewGradient
  (DCM_STD_STRUCT *std_struct,
   T_RecSlew      *rtn
  );
```

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.

8.17.2.4 dpcmGetEstimateRC

Function name	Arguments	Result	Standard Structure fields
dpcmGetEstimateRC	Driver (source) pin pointer Receiver (sink) pin pointer	Interconnect RC delay	block CellName calcMode pathData (timing-pin-specific) cellData (timing)

DCL syntax

```
EXPOSE(dpcmGetEstimateRC):
passed(pin: fromPin,toPin)
result(double: netSegRC);
```

C syntax

```
typedef struct {
  DOUBLE netSegRC;
} T_estRC;

int dpcmGetEstimateRC
  (DCM_STD_STRUCT *std_struct,
   T_estRC        *rtn,
   PIN            fromPin,
   PIN            toPin
  );
```

Returns an 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).

8.17.2.5 dpcmGetDefPortSlew

Function name	Arguments	Result	Standard Structure fields
dpcmGetDefPortSlew		Default slew	calcMode

DCL syntax

```
EXPOSE(dpcmGetDefPortSlew):
result(double: defSlew);
```

C syntax

```
typedef struct {
  DOUBLE defSlew;
} T_defPortSlew;

int dpcmGetDefPortSlew
  (DCM_STD_STRUCT *std_struct,
   T_defPortSlew  *rtn
  );
```

Returns a default value for the slew of a signal presented to a chip primary input.

8.17.2.6 dpcmGetDefPortCapacitance

Function name	Arguments	Result	Standard Structure fields
dpcmGetDefPortCapacitance		Default capacitance	calcMode

DCL syntax

```
EXPOSE(dpcmGetDefPortCapacitance):
result(double: defPortCap);
```

C syntax

```
typedef struct {
  DOUBLE defPortCap;
} T_defPortCap;

int dpcmGetDefPortCapacitance
  (DCM_STD_STRUCT *std_struct,
   T_defPortCap   *rtn
  );
```

Returns a default value of the capacitance load on a chip primary output.

8.17.3 Functions accessing netlist information

This subsection lists the functions that access netlist information.

8.17.3.1 appGetNumDriversByPin

Function name	Arguments	Result	Standard Structure fields
appGetNumDriversByPin	Pin pointer	Number of sourcePins on the interconnect	

DCL syntax

```
EXTERNAL(appGetNumDriversByPin):
passed(pin: inputPin)
result(integer: numDrivers);
```

C syntax

```
typedef struct {
  INTEGER drivers;
} T_numDriversByPin;

int appGetNumDriversByPin
  (DCM_STD_STRUCT    *std_struct,
   T_numDriversByPin *rtn,
   PIN               inputPin
  );
```

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.

8.17.3.2 appGetNumDriversByName

Function name	Arguments	Result	Standard Structure fields
appGetNumDriversByName	Pin name	Number of driver (source) pins on the interconnect	block

DCL syntax

```
EXTERNAL(appGetNumDriversByName):
passed(string: inputPin)
result(integer: numDrivers);
```

C syntax

```
typedef struct {
  INTEGER drivers;
} T_numDriversByName;

int appGetNumDriversByName
  (DCM_STD_STRUCT     *std_struct,
   T_numDriversByName *rtn,
   STRING             inputPin
  );
```

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.

8.17.3.3 appForEachParallelDriverByPin

Function name	Arguments	Result	Standard Structure fields
appForEachParallelDriverByPin	Pin pointer Function pointer to call at each parallel driver pin. Function pointer to call to perform an operation on the data. Initial value	Integer count of drivers working in parallel with the pin passed as the argument. Computed value	calcMode

DCL syntax

```
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
    &
  EXPOSE_STATEMENT(): AnyStatementThatHasTheSamePrototypeSignature
    &
  OPERATOR_STATEMENT():anyStatementThatHasTheSamePrototypeSignature
    &
  double: initialValue)
result(integer: parallelDriverCount
    &
       double: computedValue);
```

C syntax

```
typedef struct
  {INTEGER parallelDriverCount; DOUBLE computedValue;}
  T_ParaDrivPin;

typedef int(exposeType*)
  (DCM_STD_STRUCT *std_struct,
   DOUBLE         *exposeValue,
   PIN            parallelPin );

typedef int(operatorType*)
   (DCM_STD_STRUCT *std_struct,
    DOUBLE         *resultValue,
    DOUBLE         initialValue,
    DOUBLE          exposeValue );

int appForEachParallelDriverByPin
   (DCM_STD_STRUCT *std_struct,
    T_ParaDrivPin  *rtn,
    PIN            outputPin,
    exposeType     expose_function,
    operatorType   operator_function,
    DOUBLE         initialValue
   );
```

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 which belong to cells on the interconnect that are wired identically to the cell of the passed pin.

For example, in Figure 8-4 `PinA` and `PinB` are parallel drivers, `PinC` and `PinD` are not.

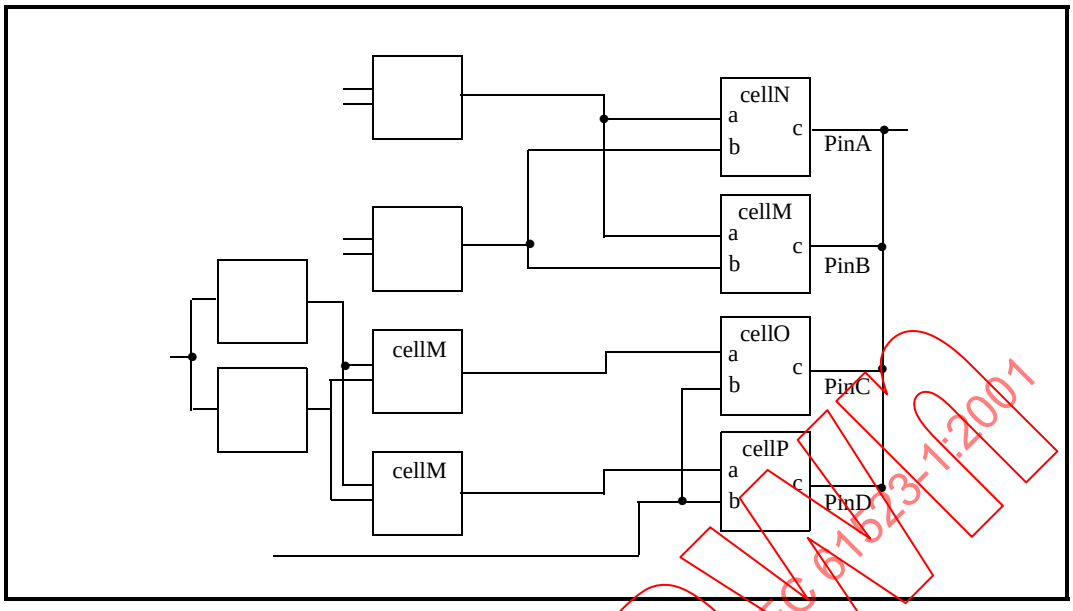


Figure 8-4 Parallel drivers example

`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:

a) `appForEachParallelDriverByPin` is called (from the DPCM) with

   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; and
   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.

8.17.3.4 appForEachParallelDriverByName

Function name	Arguments	Result	Standard Structure fields
appForEachParallelDriverByName	Pin name Function pointer to call at each parallel driver pin. Function pointer to call to perform an operation on the data. Initial value	Integer count of drivers working in parallel with the pin passed as the argument. Computed value	calcMode block

DCL syntax

```
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
    &
  EXPOSE_STATEMENT(): AnyStatementThatHasTheSamePrototypeSignature
    &
  OPERATOR_STATEMENT():anyStatementThatHasTheSamePrototypeSignature
    &
  double: initialValue)
result(integer: parallelDriverCount
    &
       double: computedValue);
```

C syntax

```
typedef struct
  {INTEGER parallelDriverCount; DOUBLE computedValue}
  T_ParaDrivName;

typedef int(*exposeType)
  (DCM_STD_STRUCT *std_struct,
   DOUBLE         *resultValue,
   STRING          outputPin);

typedef int(*operatorType)
  (DCM_STD_STRUCT *std_struct,
   DOUBLE         *resultValue,
   DOUBLE          initialValue,
   DOUBLE          exposeValue);

int appForEachParallelDriverByName
  (DCM_STD_STRUCT *std_struct,
   T_ParaDrivName *rtn,
   STRING          outputPin,
   exposeType      expose_function,
   operatorType    operator_function,
   DOUBLE          initialValue
  );
```

`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.

8.17.3.5 appGetNumPinsByPin

Function name	Arguments	Result	Standard Structure fields
appGetNumPinsByPin	Pin pointer	Total number of pins on the interconnect	

DCL syntax

```
EXTERNAL(appGetNumPinsByPin):
passed(pin: outputPin)
result(integer: totalPins);
```

C syntax

```
typedef struct {
  INTEGER totalPins;
} T_NumPinsByPin;

int appGetNumPinsByPin
  (DCM_STD_STRUCT *std_struct,
   T_NumPinsByPin *rtn,
   PIN            outputPin
  );
```

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`.

8.17.3.6 appGetNumPinsByName

Function name	Arguments	Result	Standard Structure fields
appGetNumPinsByName	Pin name	Total number of pins on the interconnect	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
  );
```

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.

8.17.3.7 appGetNumSinksByPin

Function name	Arguments	Result	Standard Structure fields
appGetNumSinksByPin	Pin pointer	Total number of sink pins on the interconnect	

DCL syntax

```
EXTERNAL(appGetNumSinksByPin):
passed(pin: outputPin)
result(integer: totalLoadPins);
```

C syntax

```
typedef struct {
  INTEGER totalLoadPins;
} T_NumSinksByPin;

int appGetNumSinksByPin
  (DCM_STD_STRUCT  *std_struct,
   T_NumSinksByPin *rtn,
   PIN             outputPin
  );
```

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`.

8.17.3.8 appGetNumSinksByName

Function name	Arguments	Result	Standard Structure fields
appGetNumSinksByName	Pin name	Total number of sink pins on the interconnect	block

DCL syntax

```
EXTERNAL(appGetNumSinksByName):
passed(string: outputPin)
result(integer: totalLoadPinsByName);
```

C syntax

```
typedef struct {
  INTEGER totalLoadPins;
} T_NumSinksByName;

int appGetNumSinksByName
  (DCM_STD_STRUCT    *std_struct,
   T_NumSinksByName *rtn,
   STRING           outputPin
  );
```

Returns the total number of load (sink) pins (including bidirectional pins) on the interconnect to which the passed pin name is connected.

8.17.3.9 appGetPiModel

Function name	Arguments	Result	Standard Structure fields
appGetPiModel	Pin pointer	estFlag Capacitance value (nearest the driver) Capacitance value (nearest the load) Resistance value	

DCL syntax

```
EXTERNAL(appGetPiModel):
passed(pin: outputPin)
result(integer: estFlag
    &
       double: capNear,capFar,Resistance);
```

C syntax

```
typedef struct
  {INTEGER estFlag ; DOUBLE capNear, capFar, Resistance}
  T_capResValue;
int appGetPiModel
  (DCM_STD_STRUCT *std_struct,
   T_capResValue  *rtn,
   PIN            outputPin
  );
```

Returns the capacitance and resistance values for the π model of the interconnect to which the passed pin is connected. 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 8-5.

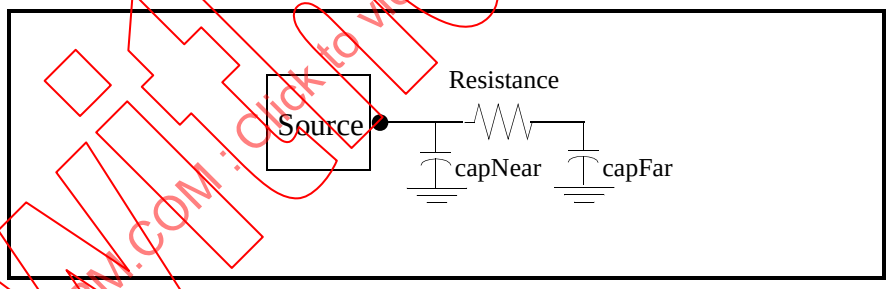


Figure 8-5 Capacitance value example

estFlag indicates whether or not 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 non-zero value. Since 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 non-zero value in accordance with the rules described in 8.11.1. In this error situation, the DPCM (or DCL subrule) is expected to use the appropriate default model.

8.17.3.10 dpcmGetWireLoadModelForBlockSize

Function name	Arguments	Result	Standard Structure fields
dpcmGetWireLoadModelForBlockSize	Area	Array index of the current wire load models according to area	calcMode

DCL syntax

```
EXPOSE(dpcmGetWireLoadModelForBlockSize):
passed(double: area)
result(integer: modelIndex);
```

C syntax

```
typedef struct {
  INTEGER modelIndex;
} T_WireLoadModelForBlockSize;

int dpcmGetWireLoadModelForBlockSize
  (DCM_STD_STRUCT               *std_struct,
   T_WireLoadModelForBlockSize *rtn,
   DOUBLE                       area
  );
```

This call requests the index number of the appropriate wire load model given the specified area 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 area to be passed into this function, the application shall first determine the highest level hierarchical block which 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.

8.17.3.11 dpcmAddWireLoadModel

Function name	Arguments	Result	Standard Structure fields
dpcmAddWireLoadModel	Model name Extrapolation constant Unit resistance Unit capacitance Length matrix	Model index	calcMode

DCL syntax

```
EXPOSE(dpcmAddWireLoadModel):
passed(string: modelName
    &
       double: extrapolationConstant,
    &
       double[*]: unitResistance, unitCapacitance, lengthMatrix)
result(integer: modelIndex);
```

C syntax

```
typedef struct {
  INTEGER modelIndex;
} T_AddWireLoadModel;

int dpcmAddWireLoadModel
  (DCM_STD_STRUCT     *std_struct,
   T_AddWireLoadModel *rtn,
   STRING             modelName,
   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 obtained from a PDEF file (see Clause 10) and 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 index 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 wireload 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:

a) 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.
b) If the name matches a wire load model previously supplied by the application, then the previous wire load model data is 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.

8.17.3.12 dpcmGetWireLoadModel

Function name	Arguments	Result	Standard Structure fields
dpcmGetWireLoadModel	Model index	Extrapolation constant Unit resistance per length Unit capacitance per length Length matrix Size	calcMode

DCL syntax

```
EXPOSE(dpcmGetWireLoadModel):
passed(integer: modelIndex)
result(double: extrapolationConstant,
    &
       double[*]: unitResistance, unitCapacitance, lengthMatrix
    &
       double: size);
```

C syntax

```
typedef struct
  {double extrapolationConstant;
   DCM_DOUBLE_ARRAY *unitResistance, *unitCapacitance, *lengthMatrix;
   DOUBLE size;}
  T_result;
int dpcmGetWireLoadModel
  (DCM_STD_STRUCT *std_struct,
   T_result       *rtn,
   INTEGER         modelIndex
  );
```

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 is given to the application consists 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.

8.17.3.13 appGetPolesAndResidues

Function name	Arguments	Result	Standard Structure fields
appGetPolesAndResidues	Driver (source) pin pointer Receiver (sink) pin pointer Order number	An array of real and imaginary poles and an array of real and imaginary residues	

DCL syntax

```
EXTERNAL(appGetPolesAndResidues):
passed(pin: sourcePin, receiverPin
    &
      integer:orderNum)
result(double[*]: rel_poles, img_poles, rel_residues, img_residues);
```

C syntax

```
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,
   PIN            sourcePin,
   PIN            receiverPin,
   INTEGER        orderNum
  );
```

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:

$$v_{step}(t) = v_{dc} + \sum_{i=1}^{q} k_i e^{P_i * t} \quad (1)$$

Where q is the number of poles, k_i are the residues and P_i are the poles (radians/sec).

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`.

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).

8.17.3.14 appGetCeffective

Function name	Arguments	Result	Standard Structure fields
appGetCeffective	delay calculation function pointer slew calculation function pointer PI model	late Ceffective early Ceffective	

DCL syntax

```
forward calc(stdDelaySlewEq):
passed(double: loadCap, inputTransition)
result(double);

forward calc(getPiModel):
passed(pin: outputPin)
result(integer: estFlag
    &
       double:nearCap,farCap,resistance);

EXTERNAL(appGetCeffective):
passed(stdDelaySlewEq(): delayEq
    &
       stdDelaySlewEq(): slewEq
    &
      getPiModel: piModel)
result(double: lateCeff, earlyCeff);
```

C syntax

```
typedef struct
  {INTEGER estFlag; DOUBLE nearCap, farCap, resistance;}
  getPIModelStruct;

typedef struct
  {DOUBLE lateCeffective, earlyCeffective;}
  T_Ceffective;

int appGetCeffective
  (DCM_STD_STRUCT      *std_struct,
   T_Ceffective        *rtn,
   DCM_GeneralFunction delayEq,
   DCM_GeneralFunction slewEq,
   getPIModelStruct    *piModel
  );
```

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 routine passes through the application the necessary parameters for the calculation of *C-effective*. The application shall pass these parameters to `dcpmCalcCeffective` without altering them.

8.17.3.15 appGetRLCnetworkByPin

Function name	Arguments	Result	Standard Structure fields
appGetRLCnetworkByPin	Pin pointer RLC network handle	newRLCnetwork handle	

DCL syntax

```
EXTERNAL(appGetRLCnetworkByPin):
passed(pin: inputPin, void:RLCnetwork)
result(void: newRLCnetwork);
```

C syntax

```
typedef struct {
  VOID newRLCnetwork;
} T_RLCnetworkByPin;

int appGetRLCnetworkByPin
  (DCM_STD_STRUCT    *std_struct,
   T_RLCnetworkByPin *rtn,
   PIN               inputPin,
   VOID              RLCnetwork
  );
```

Returns the *RLCnetwork* handle for the interconnect specified by the `PASSED` pin pointer argument. If the `PASSED RLCnetwork` argument is `0`, an existing *RLCnetwork* handle may be returned. If an appropriate *RLCnetwork* does not yet exist, the application is expected to request the DPCM build one

If the `PASSED RLCnetwork` argument is non-zero, it signifies a nested call as a result of an `dpcmAppendPinAdmittance` call (on a pass-through device, for example) while the application was constructing an *RLCnetwork* from an earlier `appGetRLCnetworkByPin` call.

8.17.3.16 appGetRLCnetworkByName

Function name	Arguments	Result	Standard Structure fields
appGetRLCnetworkByName	Pin name RLC network handle	Address of the structure that contains the RLC network in the DPCM (see 8.17.3.21)	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
  );
```

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 `appGetRLCnetworkByPin` (see 8.17.3.15).

8.17.3.17 appGetInstanceCount

Function name	Arguments	Result	Standard Structure fields
appGetInstanceCount		Count of instances	toPoint

DCL syntax

```
EXTERNAL(appGetInstanceCount):
result(integer: countOfInstances);
```

C syntax

```
typedef struct {
  INTEGER countOfInstances;
} T_InstanceCount;

int appGetInstanceCount
  (DCM_STD_STRUCT  *std_struct,
   T_InstanceCount *rtn
  );
```

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.

8.17.3.18 dpcmCalcPiModel

Function name	Arguments	Result	Standard Structure fields
dpcmCalcPiModel	Driver (source) pin pointer RLC network pointer	estFlag Capacitance value (nearest the driver) Capacitance value (nearest the load) Resistance value	block CellName pathData (timing-arc-specific) cellData (timing)

DCL syntax

```
(dpcmCalcPiModel):
passed(pin: sourcePin
       &
       void: RLCnetwork)
result(integer: estFlag
       &
       double: capNear, capFar, Resistance);
```

C syntax

```
typedef struct
  {INTEGER estFlag ; DOUBLE capNear, capFar, Resistance;}
  T_calcCapRes;
int dpcmCalcPiModel
  (DCM_STD_STRUCT *std_struct,
   T_calcCapRes   *rtn,
   PIN            sourcePin,
   VOID           RLCnetwork
  );
```

Requests the DPCM 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 8-5.

See 8.5 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 non-zero value.

8.17.3.19 dpcmCalcPolesAndResidues

Function name	Arguments	Result	Standard Structure fields
dpcmCalcPolesAndResidues	Driver (source) pin pointer Receiver (sink) pin pointer Order number RLC network pointer	An array of real and imaginary poles and an array of real and imaginary residues	CellName block pathData (timing-pin-specific) cellData (timing)

DCL syntax

```
EXPOSE(dpcmCalcPolesAndResidues):
passed(pin: sourcePin, loadPin
    &
      integer: orderNum
    &
      void:RLCnetwork)
result(double[*]: rel_poles, img_poles, rel_residues, img_residues);
```

C syntax

```
typedef struct
  {DCM_DOUBLE_ARRAY *rel_poles, *img_poles;
   DCM_DOUBLE_ARRAY *rel_residues, *img_residues;}
  T_calcPolesRes;

int dpcmCalcPolesAndResidues
  (DCM_STD_STRUCT *std_struct,
   T_calcPolesRes *rtn,
   PIN             sourcePin,
   PIN             sinkPin,
   INTEGER         orderNum,
   VOID            RLCnetwork
  );
```

Requests the DPCM 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 equation (1).

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 result arguments is actually 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).

8.17.3.20 dpcmCalcCeffective

Function name	Arguments	Result	Standard Structure fields
dpcmCalcCeffective	delay calculation function pointer slew calculation function pointer PI model	late Ceffective early Ceffective	CellName block sourceEdge sinkEdge pathData (timing-arc-specific) cellData (timing) toPoint fromPoint earlySlew lateSlew

DCL syntax

```
forward calc(stdDelaySlewEq):
passed(double: loadCap, inputTransition)
result(double);

forward calc(getPiModel):
passed(pin: outputPin)
result(integer: estFlag
    &
       double:capNear,capFar,Resistance);

EXPOSE(dpcmCalcCeffective):
passed(stdDelaySlewEq(): delayEq
    &
       stdDelaySlewEq(): slewEq
    &
       getPIModel: piModel)
result(double: lateCeff, earlyCeff );
```

C syntax

```
typedef struct
 {INTEGER estFlag; DOUBLE capNear, capFar, resistance;}
 getPIModelStruct;

typedef struct
 {DOUBLE lateCeffective, earlyCeffective;}
 T_Ceffective;

int dpcmCalcCeffective
  (DCM_STD_STRUCT      *std_struct,
   T_Ceffective        *rtn,
   DCM_GeneralFunction DelayEq,
   DCM_GeneralFunction slewEq,
   getPIModelStruct    *piModel
  );
```

Returns an equivalent “effective” capacitance as seen by the passed `toPoint`.

8.17.3.21 dpcmSetRLCmember

Function name	Arguments	Result	Standard Structure fields
dpcmSetRLCmember	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	newRLCnetwork	

DCL syntax

```
EXPOSE(dpcmSetRLCmember):
passed(void: RLCnetwork
  &
       pin: terminal1, terminal2
  &
       string: elementType, elementName
  &
       double: elementValue
  &
       integer: terminal1Type, terminal2Type)
result(void: newRLCnetwork);
```

C syntax

```
typedef struct {
  VOID newRLCnetwork;
} T_SetRLCmember;

int dpcmSetRLCmember
  (DCM_STD_STRUCT *std_struct,
   T_SetRLCmember *rtn,
   VOID           RLCnetwork,
   PIN            terminal1,
   PIN            terminal2,
   STRING         elementType,
   STRING         elementName,
   DOUBLE         elementValue,
   INTEGER        terminal1Type,
   INTEGER        terminal2Type
  );
```

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 non-zero, 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 argu-

ment 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 *RLCnetwork*s as requested by an application until they are explicitly removed via `dpcmDeleteRLCnetwork` (see 8.17.3.23). 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 8-6).

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.

Example:

In the following example (see Figure 8-6), `Pin1` and `Pin2` are terminal type 1 indicating ports, all others are terminal type 0 indicating internal nodes.

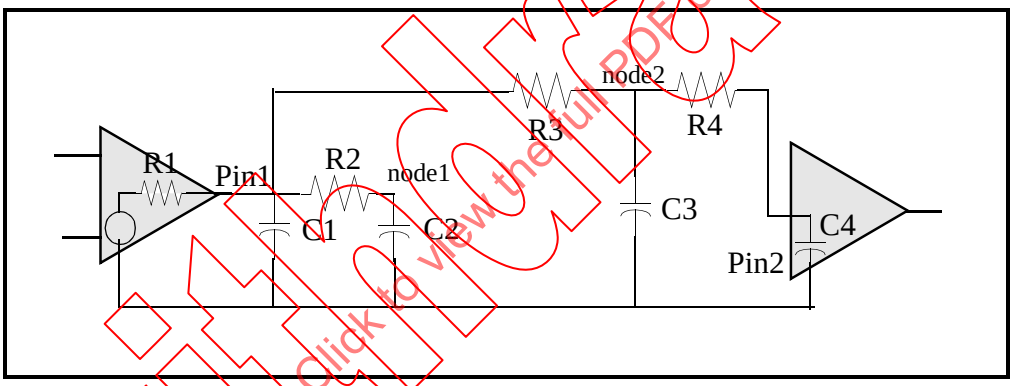


Figure 8-6 Passed and receiver pin pointers example

8.17.3.22 dpcmAppendPinAdmittance

Function name	Arguments	Result	Standard Structure fields
dpcmAppendPinAdmittance	Sink pin handle RLC network handle	RCLnetwork handle	block CellName pathData (timing-pin-specific) cellData (timing)

DCL syntax

```
EXPOSE(dpcmAppendPinAdmittance):
passed(pin: sinkPin
    &
       void: RLCnetwork)
result(void: newRLCnetwork);
```

C syntax

```
typedef struct {
  VOID newRLCnetwork;
} T_AppendPinAdmittance;

int dpcmAppendPinAdmittance
  (DCM_STD_STRUCT        *std_struct,
   T_AppendPinAdmittance *rtn,
   PIN                   inputPin,
   VOID                  RLCnetwork
  );
```

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 non-zero, 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.

8.17.3.23 dpcmDeleteRLCnetwork

Function name	Arguments	Result	Standard Structure fields
dpcmDeleteRLCnetwork	RLC network handle	return code	

DCL syntax	`EXPOSE(dpcmDeleteRLCnetwork):`
	`passed(void: RLCnetwork)`
	`result(integer: rc);`

C syntax	`typedef struct {`
	`  INTEGER rc;`
	`} T_DeleteRLCnetwork;`
	
	`int dpcmDeleteRLCnetwork`
	`  (DCM_STD_STRUCT  *std_struct,`
	`  T_DeleteRLCnetwork *rtn,`
	`  VOID  RLCnetwork`
	`  );`

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.

8.17.4 Functions exporting limit information

This subsection lists the functions that export limit information.

8.17.4.1 dpcmGetCapacitanceLimit

Function name	Arguments	Result	Standard Structure fields
dpcmGetCapacitanceLimit	Pin name	Maximum and minimum capacitance values	CellName block calcMode pathData (timing-pin-specific) cellData(timing)

DCL syntax	`EXPOSE(dpcmGetCapacitanceLimit):`
	`passed(string: pinName)`
	`result(double: lowerLimit, upperLimit);`

C syntax	`typedef struct`
	`  {DOUBLE lowerLimit, upperLimit;}`
	`  T_minMaxCap;`
	`int dpcmGetCapacitanceLimit`
	`  (DCM_STD_STRUCT *std_struct,`
	`   T_minMaxCap    *rtn,`
	`   STRING         pinName`
	`  );`

Returns the minimum and maximum capacitance the passed pin name is allowed to drive. The passed pin shall be either an output or bi-directional pin.

NOTES

- The intent of this function is to enable an application to ensure a cell operates within its design limits.

8.17.4.2 dpcmGetSlewLimit

Function name	Arguments	Result	Standard Structure fields
dpcmGetSlewLimit	Pin name Transition type	Minimum slew value Maximum slew value	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`
	`  {DOUBLE lowerLimit, upperLimit}`
	`  T_slewLimits;`
	`int dpcmGetSlewLimit`
	`  (DCM_STD_STRUCT      *std_struct,`
	`   T_slewLimits        *rtn,`
	`   STRING               pinName,`
	`   STRING               transitionType`
	`  );`

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 ensure a cell operates within its design limits.

8.17.4.3 dpcmGetXovers

Function name	Arguments	Result	Standard Structure fields
dpcmGetXovers	Pin name	Nominal capacitance Slow capacitance Fast capacitance	CellName block pathData (timing-pin-specific) cellData (timing)

DCL syntax	`EXPOSE(dpcmGetXovers):`
	`passed(string: pinName)`
	`result(double: nominal,slow,fast);`

C syntax	`typedef struct`
	`  {DOUBLE nominal, slow, fast;}`
	`  T_nsfCap;`
	`int dpcmGetXovers`
	`  (DCM_STD_STRUCT *std_struct,`
	`   T_nsfCap       *rtn,`
	`   STRING         pinName`
	`  );`

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 PVT cases chosen by the library developer.

8.17.5 Functions getting/setting model information

This subsection describes the functions that get or set model information.

8.17.5.1 dpcmGetFunctionalModeArray

Function name	Arguments	Result	Standard Structure fields
dpcmGetFunctionalModeArray		Array of functional mode group names Array of functional mode names	CellName

DCL syntax	`EXPOSE(dpcmGetFunctionalModeArray):` `result(string[*]:modeGroupArray, modeNameArray);`

C syntax	`typedef struct {`
	`DCM_STRING_ARRAY *modeGroupArray, *modeNameArray;`
	`} T_fModes;`
	
	`int dpcmGetFunctionalModeArray`
	`  (DCM_STD_STRUCT   *std_struct,`
	`   T_fModes         *rtn`
	`  );`

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"
```

8.17.5.2 dpcmGetBaseFunctionalMode

Function name	Arguments	Result	Standard Structure fields
dpcmGetBaseFunctionalMode	FuncModeGroupIndex	Index of the default functional mode	CellName

DCL Syntax

```
EXPOSE(dpcmGetBaseFunctionalMode):
passed(integer: FuncModeGroupIndex)
result(integer: modeIndex);
```

C syntax

```
typedef struct {
  INTEGER modeIndex;
} T_BaseFunctionalMod;

int dpcmGetBaseFunctionalMode
   (DCM_STD_STRUCT      *std_struct,
    T_BaseFunctionalMod *rtn,
    INTEGER             FuncModeGroupIndex
   );
```

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.

8.17.5.3 appGetCurrentFunctionalMode

Function name	Arguments	Result	Standard Structure fields
appGetCurrentFunctionalMode	FuncModeGroupIndex	Index of the current functional mode	block

DCL syntax

```
EXTERNAL(appGetCurrentFunctionalMode):
passed(integer: FuncModeGroupIndex)
result(integer: modeIndex);
```

C syntax

```
typedef struct {
  INTEGER modeIndex;
} T_modeIndex;

int appGetCurrentFunctionalMode
  (DCM_STD_STRUCT *std_struct,
   T_modeIndex    *rtn ,
   INTEGER        FuncModeGroupIndex
  );
```

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.

8.17.5.4 dpcmGetControlExistence

Function name	Arguments	Result	Standard Structure fields
dpcmGetControlExistence		FunctionalModes Expression	fromPoint toPoint pathData (timing-arc-specific) cellData (timing) block

DCL syntax

```
EXPOSE(dpcmGetControlExistence):
result(integer[*]: FunctionalModes
    &
        string: Expression);
```

C syntax

```
typedef struct
  {DCM_INTEGER_ARRAY *FunctionalModes; char *Expression;}
  T_ControlExistence
int dpcmGetControlExistence
  (DCM_STD_STRUCT     *std_struct,
   T_ControlExistence *rtn
  );
```

Returns to the application information which controls the existence of the segment identified by `pathData`:

a) The integers returned through the `FunctionalModes` result encode the elements of the `modeGroupArray` and `modeNameArray` returned by `dpcmGetFunctionalModeArray` for which the segment exists.

   The `FunctionalModes` array contains zero or more contiguous integer sequences. For each sequence:

   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]`

b) The string returned through the `Expression` result is a *ConditionalExpression* using the syntax and semantics of the "Group Condition Language" (see 7.11) 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.

8.17.5.5 dpcmSetLevel

Function name	Arguments	Result	Standard Structure fields
dpcmSetLevel	Desired DPCM computation mode (performance or accuracy) PVT derating and other scopes	Previous DPCM computation mode (performance or accuracy) and scopes	

DCL syntax

```
EXPOSE(dpcmSetLevel):
passed(integer: perfLevel, temperatureScope, voltageScope,
  functionalModeScope, wire loadModelScope)
result(integer: oldPerfLevel, oldtemperatureScope, oldvoltageScope,
  oldfunctionalModeScope, oldwireloadModelScope);
```

C syntax

```
typedef struct old_per_level {
   INTEGER oldPerLevel, oldtemperatrureScope, oldvoltageScope,
           oldfunctionalModeScope, oldwireloadModelScope;
} T_oldPerLevels;

int dpcmSetLevel
  (DCM_STD_STRUCT *std_struct,
   T_oldPerLevels *rtn,
   INTEGER        perfLevel,
   INTEGER        temperatureScope,
   INTEGER        voltageScope,
   INTEGER        functionalModeScope,
   INTEGER        wireloadModelScope
  );
```

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:

0 - Indicates calculations for maximum performance
1 - Indicates calculations for maximum accuracy

The values of the `Scope` parameters are:

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 which require model elaboration.

NOTES

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.

2 — Since no re-elaboration of models is required due to a change in `dpcmSetLevel`, the model writer shall consider what is `STORE`d to support high accuracy vs. high performance modes.

8.17.5.6 dpcmGetRailVoltageArray

Function name	Arguments	Result	Standard Structure fields
dpcmGetRailVoltageArray		Array of rail voltage	

DCL syntax

```
EXPOSE(dpcmGetRailVoltageArray):
Result(string[*]:railArray);
```

C syntax

```
typedef struct rail_array {
  DCM_STRING_ARRAY *railArray;
} T_railArray;

int dpcmGetRailVoltageArray
  (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 which doesn't 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.

8.17.5.7 dpcmGetBaseRailVoltage

Function name	Arguments	Result	Standard Structure fields
dpcmGetBaseRailVoltage	Integer index value for the rail	Voltage value for the requested rail (volts)	calcMode

DCL syntax

```
EXPOSE(dpcmGetBaseRailVoltage):
passed(integer: railIndex)
result(double:railVoltage);
```

C syntax

```
typedef struct {
  DOUBLE railVoltage;
}T_railVoltage;

int dpcmGetBaseRailVoltage
  (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.

8.17.5.8 appGetCurrentRailVoltage

Function name	Arguments	Result	Standard Structure fields
appGetCurrentRailVoltage	Integer index value for the rail	Voltage value for the current rail (volts)	calcMode block

DCL syntax

```
EXTERNAL(appGetCurrentRailVoltage):
passed(integer: railIndex)
result(double: railVoltage);
```

C syntax

```
typedef struct {
  DOUBLE railVoltage;
}T_railVoltage;

int appGetCurrentRailVoltage
  (DCM_STD_STRUCT *std_struct,
   T_railVoltage  *rtn,
   INTEGER        railIndex
  );
```

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.

8.17.5.9 dpcmGetWireLoadModelArray

Function name	Arguments	Result	Standard Structure fields
dpcmGetWireLoadModelArray		An array of wire load models	calcMode

DCL syntax

```
EXPOSE(dpcmGetWireLoadModelArray):
result(string[*]: modelArray);
```

C syntax

```
typedef struct {
  DCM_STRING_ARRAY *modelArray;
} T_WireLoadModelArray;

int dpcmGetWireLoadModelArray
  (DCM_STD_STRUCT       *std_struct,
   T_WireLoadModelArray *rtn
  );
```

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 which doesn't 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.

8.17.5.10 dpcmGetBaseWireLoadModel

Function name	Arguments	Result	Standard Structure fields
dpcmGetBaseWireLoadModel		Array index of the default wire load models	calcMode

DCL syntax

```
EXPOSE(dpcmGetBaseWireLoadModel):
result(integer: modelIndex);
```

C syntax

```
typedef struct {
  INTEGER modelIndex;
} T_BaseWireLoadModel;

int dpcmGetBaseWireLoadModelWireLoadModel
  (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 8.17.5.9) to retrieve the default wire load model name.

8.17.5.11 appGetCurrentWireLoadModel

Function name	Arguments	Result	Standard Structure fields
appGetCurrentWireLoadModel	Pin pointer	Array index of the current wire load models to use	calcMode

DCL syntax

```
EXTERNAL(appGetCurrentWireLoadModel):
passed(pin: pinPointer)
result(integer: modelIndex);
```

C syntax

```
typedef struct {
  INTEGER modelIndex;
} T_CurrentWireLoadModel;

int appGetCurrentWireLoadModel
  (DCM_STD_STRUCT         *std_struct,
   T_CurrentWireLoadModel *rtn,
   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 model index.

8.17.5.12 dpcmGetBaseTemperature

Function name	Arguments	Result	Standard Structure fields
dpcmGetBaseTemperature		Default temperature	calcMode

DCL syntax

```
EXPOSE(dpcmGetBaseTemperature):
result(double: temperature);
```

C syntax

```
typedef struct {
  DOUBLE temperature;
} T_BaseTemperature;

int dpcmGetBaseTemperature
  (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`.

8.17.5.13 dpcmGetBaseOpRange

Function name	Arguments	Result	Standard Structure fields
dpcmGetBaseOpRange		The index (from array) of the base operating range	

DCL syntax	`EXPOSE(dpcmGetBaseOpRange):` `result(integer:opRangeIndex);`

C syntax	`typedef struct {`
	`  INTEGER opRangeIndex;`
	`} T_BaseOpRange;`
	
	`int dpcmGetBaseOpRange`
	`  (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 there are not distinct operating range names 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.

8.17.5.14 dpcmGetOpRangeArray

Function name	Arguments	Result	Standard Structure fields
dpcmGetOpRangeArray		Array of operating ranges	

DCL syntax	`EXPOSE(dpcmGetOpRangeArray):` `result(string[*]:opRangeArray);`

C syntax	`typedef struct {`
	`  DCM_STRING_ARRAY *opRangeArray;`
	`} T_OpRangeArray;`
	
	`int dpcmGetOpRangeArray`
	`  (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.

8.17.5.15 appGetCurrentTemperature

Function name	Arguments	Result	Standard Structure fields
appGetCurrentTemperature		Current temperature	block calcMode

DCL syntax

```
EXTERNAL(appGetCurrentTemperature):
result(double: temperature);
```

C syntax

```
typedef struct {
  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.

8.17.5.16 appGetCurrentOpRange

Function name	Arguments	Result	Standard Structure fields
appGetCurrentOpRange		Array index of current operating range	

DCL syntax

```
EXTERNAL(appGetCurrentOpRange):
result(integer: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.

8.17.5.17 dpcmGetTimingStateArray

Function name	Arguments	Result	Standard Structure fields
dpcmGetTimingStateArray		Array of valid states and cells	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`
	`  (DCM_STD_STRUCT  *std_struct,`
	`  T_timingStateArray *rtn`
	`  );`

Returns an array of strings which represent the states for the given segment. The syntax and semantics of the state array elements are described in 7.11, except where modified below.

The state array is an ordered list of states; the application shall scan the list from the first array element to 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 *conditionExpression*s, separated by a comma and optionally surrounded by *whitespace*.

The semantics of the state array element *conditionExpression*s are the same as the "Group condition language" semantics (see 7.11.2), except:

— If two *conditionExpression*s 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] = "*"
```

8.17.5.18 appGetCurrentTimingState

Function name	Arguments	Result	Standard Structure fields
appGetCurrentTimingState		Index of current state	block pathData (timing-arc-specific) cellData (timing)

DCL syntax	`EXTERNAL(appGetCurrentTimingState):` `result(integer: 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`).

8.17.6 Functions importing instance name information

This subsection describes importing instance name information functions.

8.17.6.1 dpcmGetCellList

Function name	Arguments	Result	Standard Structure fields
dcpmGetCellList		Array of cell names Array of cellname qualifiers Array of model domains	

DCL syntax	`EXPOSE(dpcmGetCellList):` `result(string[*]: cellNameArray, cellQuallArray, model_domainArray);`

C Syntax	`typedef struct dcm_T_dcmCellList`
	`  {DCM_STRING_ARRAY *cellNameArray;`
	`  DCM_STRING_ARRAY *cellQualArray;`
	`  DCM_STRING_ARRAY *model_domainArray;}`
	`  T_dcmCellList;`
	
	`int dpcmGetCellList`
	`  (DCM_STD_STRUCT *std_struct,`
	`  T_dcmCellList  *rtn`
	`  );`

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 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 which model multiple cells and the DPCM wants to return a fully enumerated cell list.

8.17.6.2 appGetCellName

Function name	Arguments	Result	Standard Structure fields
appGetCellName	Pin pointer	Cell name, cell qualifier, model domain	

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     *ctn,
   PIN             cellPin
  );
```

Returns the cell name to which the passed pin belongs.

8.17.6.3 appGetHierPinName

Function name	Arguments	Result	Standard Structure fields
appGetHierPinName	Pin pointer	Hierarchical pin name	

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,
   PIN            pinPointer
  );
```

Returns the full hierarchical pin name for the passed pin.

8.17.6.4 appGetHierBlockName

Function name	Arguments	Result	Standard Structure fields
appGetHierBlockName	Pin pointer	Hierarchical cell instance name	

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,
   PIN             pinPointer
  );
```

Returns the full hierarchical name of the instance to which the passed pin is connected.

8.17.6.5 appGetHierNetName

Function name	Arguments	Result	Standard Structure fields
appGetHierNetName	Pin pointer	Hierarchical interconnect name	

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,
   PIN            pinPointer
  );
```

Returns the full hierarchical name of the electrical net to which the passed pin is connected.

8.17.7 Process information functions

This subsection lists the process information functions.

8.17.7.1 dpcmGetThresholds

Function name	Arguments	Result	Standard Structure fields
dpcmGetThresholds	pin pointer	Voltage transition delay points	calcMode CellName block pathData (timing-pin-specific) cellData (timing)

DCL syntax

```
EXPOSE(dpcmGetThresholds):
passed(pin: pinPointer)
result(double: vol, voh, lowerTransitionThreshold,
 upperTransitionThreshold, riseSwitchLevel, fallSwitchLevel);
```

C syntax

```
typedef struct
  {DOUBLE vol, voh, lowerTransitionThreshold,
  upperTransitionThreshold, riseSwitchLevel, fallSwitchLevel;}
  T_thresholds;
int dpcmGetThresholds
  (DCM_STD_STRUCT *std_struct,
   T_thresholds   *rtn,
   PIN            pinPointer
  );
```

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`, `fallSwitchLevel` is defined as the points of delay characterization. `voh`, `vol` is defined as the maximum/minimum voltage swing at which a particular pin was modeled.

8.17.7.2 appGetThresholds

Function name	Arguments	Result	Standard Structure fields
appGetThresholds	pin pointer	voltage low voltage high low transition threshold high transition threshold rise switch level fall switch level	

DCL syntax

```
EXTERNAL(appGetThresholds):
passed(pin: pinPointer)
result (double: vol, voh, lowerTransitionThreshold,
  upperTransitionThreshold, riseSwitchLevel, fallSwitchLevel);
```

C syntax

```
typedef struct
  {DOUBLE vol, voh, lowerTransitionThreshold,
   upperTransitionThreshold, riseSwitchLevel, fallSwitchLevel;
  } T_thresholds;
int appGetThresholds
  (DCM_STD_STRUCT *std_struct,
   T_thresholds   *rtn,
   PIN            pinPointer
  );
```

This function allows the DPCM to retrieve voltage, transition, and delay points. The application shall call `dpcmGetThresholds` to get this information (see 8.17.7.1).

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 switch it back when it returns the answer to the requesting DPCM).

8.17.8 Miscellaneous standard interface functions

This subsections shows the miscellaneous standard interface functions.

8.17.8.1 appGetExternalStatus

Function name	Arguments	Result	Standard Structure fields
appGetExternalStatus	String containing the name of the EXTERNAL	Integer encoding status	

DCL syntax

```
EXTERNAL(appGetExternalStatus):
passed(string: externalName)
result(integer: externalStatus);
```

C syntax

```
typedef struct {
  INTEGER externalStatus;
} T_extenalStatus;

int appGetExternalStatus
  (DCM_STD_STRUCT  *std_struct,
   T_extenalStatus *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:

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.

8.17.8.2 appGetVersionInfo

Function name	Arguments	Result	Standard Structure fields
appGetVersionInfo		version of P1481 with which application is compliant	

DCL syntax

```
EXTERNAL(appGetVersionInfo):
result(string: P1481_version);
```

C syntax

```
typedef struct {
  STRING P1481_version;
} T_VersionInfo;

int appGetVersionInfo
  (DCM_STD_STRUCT *std_struct,
   T_VersionInfo  *rtn
  );
```

Returns the version of P1481 with which the application is compliant. The string for an application compliant with this version of P1481 shall be "IEEE 1481-1998."

8.17.8.3 appGetResource

Function name	Arguments	Result	Standard Structure fields
appGetResource	String containing the name of the resource desired. String containing the resource's description.	String containing the value of the named resource.	

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
  );
```

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.

8.17.8.4 dpcmGetRuleUnitToSeconds

Function name	Arguments	Result	Standard Structure fields
dpcmGetRuleUnitToSeconds		Scale factor power	

DCL syntax

```
EXPOSE(dpcmGetRuleUnitToSeconds):
result(integer: scaleFactorPower);
```

C syntax

```
typedef struct {
  INTEGER scaleFactorPower;
} T_RuleUnitToSecond;

int dpcmGetRuleUnitToSeconds
  (DCM_STD_STRUCT     *std_struct,
   T_RuleUnitToSecond *rtn
  );
```

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.

Example

The following example shows how a DPCM indicates the time unit in nanoseconds:

```
EXPOSE calc(dpcmGetRuleUnitToSeconds): result(integer: -9);
```

8.17.8.5 dpcmGetRuleUnitToOhms

Function name	Arguments	Result	Standard Structure fields
dpcmGetRuleUnitToOhms		Scale factor power	

DCL syntax

```
EXPOSE(dpcmGetRuleUnitToOhms):
result(integer: scaleFactorPower);
```

C syntax

```
typedef struct {
  INTEGER scaleFactorPower;
} T_RuleUnitToOhms;

int dpcmGetRuleUnitToOhms
  (DCM_STD_STRUCT   *std_struct,
   T_RuleUnitToOhms *rtn
  );
```

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.

Example

The following example shows how a DPCM indicates the resistance unit in Kohms:

```
EXPOSE calc(dpcmGetRuleUnitToOhms): result(integer: 3);
```

8.17.8.6 dpcmGetRuleUnitToFarads

Function name	Arguments	Result	Standard Structure fields
dpcmGetRuleUnitToFarads		Scale factor power	

DCL syntax

```
EXPOSE(dpcmGetRuleUnitToFarads):
result(integer: scaleFactorPower);
```

C syntax

```
typedef struct {
  INTEGER scaleFactorPower;
} T_RuleUnitToFarads;

int dpcmGetRuleUnitToFarads
  (DCM_STD_STRUCT     *std_struct,
   T_RuleUnitToFarads *rtn
  );
```

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.

Example

The following example shows how a DPCM indicates the capacitance unit in picoFarads:

```
EXPOSE calc(dpcmGetRuleUnitToFarads): result(integer: -12);
```

8.17.8.7 dpcmGetRuleUnitToHenries

Function name	Arguments	Result	Standard Structure fields
dpcmGetRuleUnitToHenries		Scale factor power	

DCL syntax

```
EXPOSE(dpcmGetRuleUnitToHenries):
result(integer: scaleFactorPower);
```

C syntax

```
typedef struct {
  INTEGER scaleFactorPower;
} T_RuleUnitToHenries;

int dpcmGetRuleUnitToHenries
  (DCM_STD_STRUCT      *std_struct,
   T_RuleUnitToHenries *rtn
  );
```

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.

Example

The following example demonstrates how a DPCM indicates the inductance unit in microHenries:

```
EXPOSE calc(dpcmGetRuleUnitToHenries): result(integer: -6);
```

8.17.8.8 dpcmGetRuleUnitToWatts

Function name	Arguments	Result	Standard Structure fields
dpcmGetRuleUnitToWatts		Scale factor power	

DCL syntax

```
EXPOSE(dpcmGetRuleUnitToWatts):
result(integer: scaleFactorPower);
```

C syntax

```
typedef struct {
  INTEGER scaleFactorPower;
} T_RuleUnitToWatts;

int dpcmGetRuleUnitToWatts
  (DCM_STD_STRUCT    *std_struct,
   T_RuleUnitToWatts *rtn
  );
```

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.

Example

The following example demonstrates how a DPCM indicates the power unit in microWatts:

```
EXPOSE calc(dpcmGetRuleUnitToWatts): result(integer: -6);
```

8.17.8.9 dpcmGetRuleUnitToJoules

Function name	Arguments	Result	Standard Structure fields
dpcmGetRuleUnitToJoules		Scale factor power	

DCL syntax

```
EXPOSE(dpcmGetRuleUnitToJoules):
result(integer: scaleFactorPower);
```

C syntax

```
typedef struct {
  INTEGER scaleFactorPower;
} T_RuleUnitToJoules;

int dpcmGetRuleUnitToJoules
  (DCM_STD_STRUCT     *std_struct,
   T_RuleUnitToJoules *rtn
  );
```

Returns the basic energy units the library assumes, expressed as an integer power of 10. The value $10^{\text{scaleFactorPower}}$, 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);
```

8.17.8.10 dpcmIsSlewTime

Function name	Arguments	Result	Standard Structure fields
dpcmIsSlewTime		indicator	

DCL syntax

```
EXPOSE(dpcmIsSlewTime):
result(integer:slewTime);
```

C syntax

```
typedef struct {
  INTEGER slewTime;
} T_IsSlewTime;

int dpcmIsSlewTime
  (DCM_STD_STRUCT *std_struct,
   T_IsSlewTime   *rtn
  );
```

Returns the units for calculated slews as absolute time or rate of change. If `slewTime` is non-zero, the slew values are in time units. If the `slewTime` is zero, the slew values are rate of change (time/volts) units.

8.17.8.11 dpcmDebug

Function name	Arguments	Result	Standard Structure fields
dpcmDebug	Debug level	Previous level	

DCL syntax

```
EXPOSE(dpcmDebug):
passed(integer: dpcmDebugLevel)
result(integer: prevLevel);
```

C syntax

```
typedef struct {
  INTEGER prevLevel;
} T_Debug;

int dpcmDebug
  (DCM_STD_STRUCT *std_struct,
   T_Debug        *rtn ,
   INTEGER        dpcmDebugLevel
  );
```

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.

8.17.8.12 dpcmGetVersionInfo

Function name	Arguments	Result	Standard Structure fields
dpcmGetVersionInfo		Library identifier version of P1481 with which library is compliant	

DCL syntax

```
EXPOSE(dpcmGetVersionInfo):
result(string: libIdentification, P1481_version);
```

C syntax

```
typedef struct
  {STRING libIdentification, P1481_version;}
  T_intVer;
int dpcmGetVersionInfo
  (DCM_STD_STRUCT *std_struct,
   T_intVer       *rtn
  );
```

Returns strings which identify the technology library and the version of P1481 with which the library is compliant. The library identification is an arbitrary string. The `P1481_version` result variable shall be set to "IEEE 1481-1998."

8.17.8.13 dpcmHoldControl

Function name	Arguments	Result	Standard Structure fields
dpcmHoldControl		Pointer to application's node structure	CellName fromPoint toPoint sourceEdge sinkEdge

DCL syntax	`EXPOSE(dpcmHoldControl):` `result(integer: doHoldControlSnip);`

C syntax	`typedef struct {`
	`  INTEGER doHoldControlSnip;`
	`} T_HoldControl;`
	
	`int dpcmHoldControl`
	`  (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 C.

8.17.8.14 dpcmFillPinCache

Function name	Arguments	Result	Standard Structure fields
dpcmFillPinCache	Pin pointer Resistance load Capacitance load Slew In Slew Out Memory handle in	Memory handle out	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
  (DCM_STD_STRUCT    *std_struct,
   T_FillPinCache    *rtn,
   PIN               pinPointer,
   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. Each time this function is called all data for the specified pin is replaced.

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 of the `SINK_EDGE` enumerations. The `slewOut` array is indexed by the `SOURCE_EDGE_SCALAR` enumeration. The application shall supply the output slew value for each of the `SOURCE_EDGE` enumerations. For requested data that is 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. (see section 7.3 for more information loading the pin cache)

8.17.8.15 dpcmFreePinCache

Function name	Arguments	Result	Standard Structure fields
dpcmFreePinCache	Memory handle	Return code	CellName block cellData (timing or power)

DCL syntax

```
EXPOSE(dpcmFreePinCache):
passed(void: memoryHandleIn)
result(integer: rc);
```

C syntax

```
typedef struct {
  INTEGER rc;
} T_FreePinCache;

int dpcmFreePinCache
  (DCM_STD_STRUCT *std_struct,
   T_FreePinCache *rtn,
   VOID           memoryHandleIn
  );
```

This function is called to free the load and slew cache when no longer needed.

8.17.8.16 appRegisterCellInfo

Function name	Arguments	Result	Standard Structure fields
appRegisterCellInfo	Integers indicating whether capacitance load, resistance load or slew is needed	Memory Handle	block cellName

DCL syntax

```
EXTERNAL(appRegisterCellInfo):
passed(integer: FillCapLoad, FillResLoad, FillSlew)
result(void: memoryHandleOut );
```

C syntax

```
typedef struct {
 VOID memoryHandleOut;
} T_memoryHandle;

int appRegisterCellInfo
  (DCM_STD_STRUCT *std_struct,
   T_memoryHandle *rtn,
   INTEGER        FillCapLoad,
   INTEGER        FillResLoad,
   INTEGER        FillSlew
  );
```

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 who's types (inputs, bidis 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 7.3).

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.

8.17.9 Power related functions

This subsection shows the power related functions.

8.17.9.1 dpcmGetCellPowerInfo

Function name	Arguments	Result	Standard Structure fields
dpcmGetCellPowerInfo		Group pin list Group condition list Sensitivity list Initial state choices Supported methods	CellName cellData (power)

DCL syntax

```
EXPOSE(dpcmGetCellPowerInfo):
result(string[*]: groupPinList, groupConditionList, sensitivityList,
  initialStateChoices
    &
    integer: aet_supported, group_supported, pin_supported);
```

C syntax

```
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
  (DCM_STD_STRUCT  *std_struct,
   T_lists         *rtn
  );
```

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:

— `aet_supported`: `dpcmGetAETCellPowerWithSensitivity`

— `group_supported`: `dpcmGetPowerWithState`
— `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 `dcpmAetCellPowerWithSensitivity`.
— The `initial_state_choices` for all three power calculation methods.

A `0` length array is returned for each of the resultant arrays (the `group_pin_list`, `group_condition_list`, `sensitivity_list`, and `initial_state_choices`) when this information is not needed or not available.

8.17.9.2 dpcmGetCellPowerWithState

Function name	Arguments	Result	Standard Structure fields
dpcmGetPowerWithState	Group index Condition index	Energy/rail(static) Static power/rail(static) Total energy Total static power	CellName block cellData (power) calcMode

DCL syntax

```
EXPOSE(dpcmGetPowerWithState);
passed(integer: groupIndex, conditionIndex )
result(double[*]: energyPerRail, staticPowerPerRail
    &
        double: totalEnergy, totalStaticPower
    );
```

C syntax

```
typedef struct
  {DCM_DOUBLE_ARRAY *energyPerRail, *staticPowerPerRail;
   DOUBLE            totalEnergy;
   DOUBLE            totalStaticPower;
   } T_energy;

int dpcmGetPowerWithState
   (DCM_STD_STRUCT *std_struct,
    T_energy       *rtn,
    INTEGER         groupIndex,
    INTEGER         conditionIndex
   );
```

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.

8.17.9.3 dpcmGetAETCellPowerWithSensitivity

Function name	Arguments	Result	Standard Structure fields
dpcmGetAETCellPowerWithSensitivity	Sensitivity mask	Energy/rail(static) Static power/rail(static) Total energy Total static power	CellName block cellData calcMode (power)

DCL syntax

```
EXPOSE(dpcmGetAETCellPowerWithSensitivity):
passed(integer[*]: sensitivityMask )
result(double[*]: energyPerRail, staticPowerPerRail
    &
        double:totalEnergy, totalStaticPower
  );
```

C syntax

```
typedef struct
  {DCM_DOUBLE_ARRAY *energyPerRail, *staticPowerPerRail;
   DOUBLE           totalEnergy;
   DOUBLE           totalStaticPower;
   } T_ energy;

int dpcmGetAETCellPowerWithSensitivity
  (DCM_STD_STRUCT   *std_struct,
   T_energy         *rtn,
   DCM_DOUBLE_ARRAY *sensitivityMask
  );
```

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 upon 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:

a) When the element of the sensitivity list array (returned by `dpcmGetCellPowerInfo`) contains a single pin:

1) Each element of the mask array encodes the `from` and `to` states in the two least-significant bytes of the integer, as shown in the example below (Figure 8-7).

```
MSB                                LSB
+------+------+-------+------+
|      |      | From  |  To  |
+------+------+-------+------+
```

Figure 8-7 Integer LSB example

2) For both `from` and `to`, the encoding is shown in Table 8-15.

Table 8-15—Mask encoding

Value	Pin State
0	0
1	1
2	Z
3	X

b) When the element of the sensitivity list array (returned by `dpcmGetCellPowerInfo`) contains multiple pins:

1) Each element of the mask array encodes the state of these pins in the least-significant byte of the integer, using the 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`

2) 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`.

8.17.9.4 dpcmGetPinPower

Function name	Arguments	Result	Standard Structure fields
dpcmGetPinPower	Pin pointer Ask for registration	Energy/rail(static) Static power/rail(static) totalEnergy totalStaticPower	CellName block cellData (power) pathData (power-pin-specific) calcMode

DCL syntax

```
EXPOSE(dpcmGetPinPower):
passed(pin: pinPointer
    &
       integer: ask_for_registration
    )
result(double[*]:  energyPerRail, staticPowerPerRail
        &
       double: totalEnergy, totalStaticPower
   );
```

C syntax

```
typedef struct
  {DCM_DOUBLE_ARRAY *energyPerRail, *staticPowerPerRail;
   DOUBLE            totalEnergy;
   DOUBLE            totalStaticPower;
   } T_energy;

int dpcmGetPinPower
  (DCM_STD_STRUCT *std_struct,
   T_energy       *rtn,
   PIN            pinPointer,
   INTEGER        ask_for_registration
  );
```

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 specified in the *Standard Structure*.

8.17.9.5 dpcmAETGetSettlingTime

Function name	Arguments	Result	Standard Structure fields
dpcmAETGetSettlingTime		Array of pins Array of settling times	CellName block cellData (power) calcMode

DCL syntax

```
EXPOSE(dpcmAETGetSettlingTime):
result(string[*]: pinList
    &
       double[*]: settlingTimes);
```

C syntax

```
typedef struct
  {DCM_STRING_ARRAY *pinList;
   DCM_DOUBLE_ARRAY *settlingTimes;}
  T_times;
int dpcmAETGetSettlingTime
  (DCM_STD_STRUCT *std_struct,
   T_times        *rtn
  );
```

Returns two parallel arrays. The first array contains strings, each of which is a pin list whose syntax is defined in 7.8. 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 (All Events Trace) related power computation.

See the definition of settling time as well as the method for calculating power in 7.5 and 7.6. Each pin in the pin list array shall be an actual pin name on the cell specified in the *Standard Structure*.

8.17.9.6 dpcmAETGetSimultaneousSwitchTime

Function name	Arguments	Result	Standard Structure fields
dpcmAETGetSimultaneousSwitchTime		Array of pins Array of simultaneous switch times	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
  (DCM_STD_STRUCT *std_struct,
   T_times        *rtn
  );
```

Returns two parallel arrays. The first array contains strings, each of which is a pin list whose syntax is defined in 7.8. 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 (All Events Trace) related power computation.

See the definition of simultaneous switching time as well as the method for calculating power in 7.4 and 7.6. Each pin in the pin list array shall be an actual pin name on the cell specified in the *Standard Structure*.

8.17.9.7 dpcmGroupGetSettlingTime

Function name	Arguments	Result	Standard Structure fields
dpcmGroupGetSettlingTime		Array of pins Array of settling times	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
  (DCM_STD_STRUCT *std_struct,
   T_times        *rtn
  );
```

Returns two parallel arrays. The first array contains strings, each of which is a pin list whose syntax is defined in 7.8. 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 7.5 and 7.6. Each pin in the pin list array shall be an actual pin name on the cell specified in the *Standard Structure*.

8.17.9.8 dpcmGroupGetSimultaneousSwitchTime

Function name	Arguments	Result	Standard Structure fields
dpcmGroupGetSimultaneousSwitchTime		Array of pins Array of simultaneous switch times	CellName block cellData (power) calcMode

DCL syntax

```
EXPOSE(dpcmGroupGetSimultaneousSwitchTime):
result(string[*]: pinList
    &
       double[*]: SimultaneousSwitchTimes);
```

C syntax

```
typedef struct
  {DCM_STRING_ARRAY *pinList;
   DCM_DOUBLE_ARRAY *SimultaneousSwitchTimes;}
  T_times;
int dpcmGroupGetSimultaneousSwitchTime
  (DCM_STD_STRUCT *std_struct,
   T_times        *rtn
  );
```

Returns two parallel arrays. The first array contains strings, each of which is a pin list whose syntax is defined in 7.8. 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 7.4 and 7.6. Each pin in the pin list array shall be an actual pin name on the cell specified in the *Standard Structure*.

8.17.9.9 dpcmCalcPartialSwingEnergy

Function name	Arguments	Result	Standard Structure fields
dcpmCalcPartialSwingEnergy	Pin pointer Pin group index Pin condition index Width of occurrence	Energy per rail Total energy	CellName block cellData (power) calcMode

DCL syntax

```
EXPOSE dpcmCalcPartialSwingEnergy):
passed(pin: pinPointer
    &
       integer: group_index, condition_index
    &
       double: width)
result(double[*]: energyPerRail & double: totalEnergy);
```

C syntax

```
typedef struct {
  DCM_DOUBLE_ARRAY *energyPerRail;
  DOUBLE totalEnergy;
} T_CalcPartialSwingEnergy;

int dpcmCalcPartialSwingEnergy
  (DCM_STD_STRUCT           *std_struct,
   T_CalcPartialSwingEnergy *rtn,
   PIN                      pinPointer,
   INTEGER                  Group_index,
   INTEGER                  Condition_index,
   DOUBLE                   width
  );
```

Returns the dynamic energy per rail and the total dynamic 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.

8.17.9.10 dpcmSetInitialState

Function name	Arguments	Result	Standard Structure fields
dcpmSetInitialState	Initial state index cache handle	Static power/rail Total static power cacheHandle	CellName block cellData (power)

DCL syntax

```
EXPOSE(dpcmSetInitialState):
passed(integer: initialStateIndex
    &
       void: cacheHandleIn)
result(double[*]: staticPowerPerRail
          &
        double:      totalStaticPower
          &
       void: cacheHandleOut);
```

C syntax

```
typedef struct
  {DCM_DOUBLE_ARRAY *staticPowerPerRail;
   double            totalStaticPower;
   VOID              cacheHandleOut;}
  T_ energy;

int dpcmSetInitialState
  (DCM_STD_STRUCT *std_struct,
   T_energy       *rtn,
   INTEGER        initialStateIndex,
   VOID           cacheHandleIn
  );
```

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 7.2. 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 which has initial states, the DPCM shall call `appGetStateCache` to retrieve this state cache handle.

8.17.9.11 dpcmFreeStateCache

Function name	Arguments	Result	Standard Structure fields
dpcmFreeStateCache	cache handle	Return code	CellName block cellData

DCL syntax

```
EXPOSE(dpcmFreeStateCache):
passed(void: cacheHandleIn)
result(integer: rc);
```

C syntax

```
typedef struct {
  INTEGER rc;
} T_FreeStateCache;

int dpcmFreeStateCache
  (DCM_STD_STRUCT   *std_struct,
   T_FreeStateCache *rtn,
   VOID             cacheHandleIn
  );
```

This function is called to free the state cache when no longer needed (see 7.2).

8.17.9.12 appGetStateCache

Function name	Arguments	Result	Standard Structure fields
appGetStateCache		cache Handle out	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 7.2). 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.

8.17.9.13 dpcmGetNetEnergy

Function name	Arguments	Result	Standard Structure fields
dpcmGetNetEnergy		Net energy	CellName block fromPoint cellData (power) pathData (power-pin-specific) calcMode

DCL syntax	`EXPOSE(dpcmGetNetEnergy):` `result(double: netEnergy);`

C syntax	`typedef struct {`
	`  DOUBLE netEnergy;`
	`} T_NetEnergy;`
	
	`int dpcmGetNetEnergy`
	`  (DCM_STD_STRUCT *std_struct,`
	`   T_NetEnergy    *rtn`
	`  );`

Returns the energy consumed by a transition on the net connected to `fromPoint`.

8.17.10 Array manipulation functions

These functions allow the application to manipulate array data that is returned by the DPCM.

8.17.10.1 dcm_copy_DCM_ARRAY

Function name	Arguments	Result	Standard Structure fields
dcm_copy_DCM_ARRAY	DCM_ARRAY DCM_AATTS	DCM_ARRAY	

C syntax	`DCM_ARRAY *dcm_copy_DCM_ARRAY` `  (DCM_ARRAY *originalArray, DCM_AATTS attributes);`

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.

8.17.10.2 dcm_new_DCM_ARRAY

Function name	Arguments	Result	Standard Structure fields
dcm_new_DCM_ARRAY	number of dimensions, vector of elements per dimension, size of each element,	DCM_ARRAY	

C syntax

```
DCM_ARRAY
  *dcm_new_DCM_ARRAY
  (int                          numDims,
   int                          *elementsPer,
   int                          elementSize,
   DCM_ATYPE                    elementType,
   DCM_AATTS                    attributes,
   DCM_AINIT                    initialize,
   DCM_ArrayInitUserFunction initializer
  );
```

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.

C syntax

```
typedef enum
  DCM_Array_Element_Types
       {DCM_ATYPE_ERROR,          /* Error                                */
        DCM_ATYPE_Integer,        /* INTEGER                              */
        DCM_ATYPE_String,         /* STRING                               */
        DCM_ATYPE_Double,         /* DOUBLE                               */
        DCM_ATYPE_Float,          /* FLOAT or NUMBER in TABLEDEF DATA*/
        DCM_ATYPE_Function,        /* Function array.                      */
        /*******************************************************
        ** Future additions go here.
        *******************************************************/
        DCM_ATYPE_MAX}            /* Ceiling.                             */
  DCM_ATYPE;
```

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` represents an enumeration which controls the initialization of `DCM_ARRAYs`.

C syntax

```
typedef enum
  DCM_Array_Initialization
       {DCM_AINIT_doNotInitialize,
        DCM_AINIT_initAllZeroes,
        DCM_AINIT_initByType,
        DCM_AINIT_useFunction,
        DCM_AINIT_MAX}
  DCM_AINIT;
```

- 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 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 dcm_new_DCM_ARRAY to call the supplied initializer routine. If this scalar value is present and the initializer parameter is 0 (zero), no initialization of the array elements takes place.

If DCM_AINIT_initByType is passed in then the following type pre-initialization patterns occur.

INTEGER	sets each element to MININT
STRING	sets each element to NULL (0)
DOUBLE	sets each element to NaN
FLOAT	sets each element to NaN

The DCM_ArrayInitUserFunction 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.

C syntax	typedef int(*DCM_ArrayInitUserFunction)(DCM_ARRAY *);

8.17.10.3 dcm_sizeof_DCM_ARRAY

Function name	Arguments	Result	Standard Structure fields
dcm_sizeof_DCM_ARRAY	DCM_ARRAY	Size of the DCM array	

C syntax	int dcm_sizeof_DCM_ARRAY(DCM_ARRAY *array);

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.

8.17.10.4 dcm_lock_DCM_ARRAY

Function name	Arguments	Result	Standard Structure fields
dcm_lock_DCM_ARRAY	DCM_ARRAY	Return code	

C syntax	int dcm_lock_DCM_ARRAY(DCM_ARRAY *array);

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. Neither the application nor DPCM shall unlock the array more times than the application or DPCM respectively locked it.

If for any reason the system encounters an error a non-zero value is returned, otherwise a successful return value of zero is returned.

8.17.10.5 dcm_unlock_DCM_ARRAY

Function name	Arguments	Result	Standard Structure fields
dcm_unlock_DCM_ARRAY	DCM_ARRAY	Return code	

C syntax	`int dcm_unlock_DCM_ARRAY(DCM_ARRAY *array);`

`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 non-zero value is returned, otherwise a successful return value of zero is returned.

8.17.10.6 dcm_getNumDimensions

Function name	Arguments	Result	Standard Structure fields
dcm_getNumDimensions	DCM_ARRAY	Number of dimensions	

C syntax	`int dcm_getNumDimensions(DCM_ARRAY *array);`

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.

8.17.10.7 dcm_getNumElementsPer

Function name	Arguments	Result	Standard Structure fields
dcm_getNumElementsPer	DCM_ARRAY	Number of elements in each dimension	

C syntax	`int *dcm_getNumElementsPer(DCM_ARRAY *array, int *answer);`

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.

8.17.10.8 dcm_getNumElements

Function name	Arguments	Result	Standard Structure fields
dcm_getNumElements	DCM_ARRAY	Number of dimensions	

C syntax	`int dcm_getNumElements(DCM_ARRAY *array, int dimension);`

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*.

8.17.10.9 dcm_getElementType

Function name	Arguments	Result	Standard Structure fields
dcm_getElementType	DCM_ARRAY	Type of element	

C syntax	`DCM_ATYPE dcm_getElementType(DCM_ARRAY *array);`

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.

8.17.10.10 dcm_arraycmp

Function name	Arguments	Result	Standard Structure fields
dcm_arraycmp	Two DCM_ARRAYs		

C syntax	`int dcm_arraycmp(DCM_ARRAY *a1, DCM_ARRAY *a2);`

The application service `dcm_arraycmp` is passed two `DCM_ARRAY`s and compares them for equality. If the two arrays contain (bit-by-bit) identical data the value of zero is returned, otherwise a non-zero value is returned.

8.17.11 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 function between the DPCM and the application.

8.17.11.1 dcmCellList

Function name	Arguments	Result	Standard Structure fields
dcmCellList	*Standard Structure* pointer	Array of cell names Array of cellname qualifiers Array of model domain	

C syntax

```
typedef struct
  dcm_T_dcmCellList {DCM_STRING_ARRAY *cellNameArray;
                     DCM_STRING_ARRAY *cellQualArray;
                     DCM_STRING_ARRAY *model_domainArray;}
  T_dcmCellList;
int dcmCellList (DCM_STD_STRUCT *std_struct, T_dcmCellList *rtn);
```

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 `MODEL`s in the DPCM.

8.17.11.2 dcmSetNewStorageManager

Function name	Arguments	Result	Standard Structure fields
dcmSetNewStorageManager	malloc function pointer free function pointer realloc function pointer		

C syntax

```
int dcmSetNewStorageManager
  (DCM_Malloc_Type  malloc,
   DCM_Free_Type    free,
   DCM_Realloc_Type realloc
  );
```

Sets the function pointers for memory allocation, free, and reallocation functions for the DPCM. The typedefs match the ISO C (ISO/IEC 9899:1990, Programming Languages — C) `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 returns a non-zero 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, then calls to `dcmMalloc`, `dcmFree`, or `dcmRealloc` result in the DPCM calling back the designated application functions.

8.17.11.3 dcmMalloc

Function name	Arguments	Result	Standard Structure fields
dcmMalloc	Number of bytes to allocate		

C syntax	`void *dcmMalloc(size_t numBytes);`

Returns a pointer to a block of memory at least *`numBytes`* long, using the storage management function currently in effect.

8.17.11.4 dcmFree

Function name	Arguments	Result	Standard Structure fields
dcmFree	Pointer to memory block to be freed		

C syntax	`void dcmFree(void  *allocatedBlock);`

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`.

8.17.11.5 dcmRealloc

Function name	Arguments	Result	Standard Structure fields
dcmRealloc	Number of bytes to reallocate		

C syntax	`void *dcmRealloc(void  *allocatedBlock, size_t numBytes);`

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`.

8.17.11.6 dcmBindRule

Function name	Arguments	Result	Standard Structure fields
dcmBindRule	Rule name		

C syntax	`void *dcmBindRule(const char *rootSubruleName);`

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 8.15).

`dcmBindRule` returns a pointer to the rule initialization entry point `dcm_rule_init` (see 8.17.11.16) if the primary (`root`) library rule can be found and loaded. If the rule can not be found or there was an error in loading, the pointer returned is zero. If this call is successful, the application shall not call it again until a successful response from `dcmUnbindRule` occurs.

8.17.11.7 dcmAddRule

Function name	Arguments	Result	Standard Structure fields
dcmAddRule	rule name	return code	

C syntax	`void* dcmAddRule(const char subruleName, int *returnCode);`

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 which contains the subrule name for the technology library to be added (see 8.15). returnCode is set to the integer return code for this function call. (see section 8.11.1)

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 can not be found or a loading error occurs, the pointer returned is zero.

8.17.11.8 dcmUnbindRule

Function name	Arguments	Result	Standard Structure fields
dcmUnbindRule			

C syntax	`int dcmUnbindRule(void *initFunction);`

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 non-zero value is returned.

8.17.11.9 dcmFindFunction

Function name	Arguments	Result	Standard Structure fields
dcmFindFunction	EXPOSE function name Function table		

C syntax	`DCM_GeneralFunction dcmFindFunction(char *fcnName,` `  DCM_FunctionTable exposes);`

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 8.17.11.16).

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.

8.17.11.10 dcmFindAppFunction

Function name	Arguments	Result	Standard Structure fields
dcmFindAppFunction	EXTERNAL function name		

C syntax	`int dcmFindAppFunction(char *fcnName);`

Determines whether the application defined the indicated EXTERNAL function. This function returns a non-zero value if the application did define the function; otherwise, a zero value is returned.

8.17.11.11 dcmQuietFindFunction

Function name	Arguments	Result	Standard Structure fields
dcmQuietFindFunction	EXPOSE function name Function table		

C syntax	`DCM_GeneralFunction dcmQuietFindFunction(char *fcnName,` `  DCM_FunctionTable *exposes);`

This function is the same as `dcmFindFunction`, except no error is issued if the function is not found (see 8.17.11.9).

8.17.11.12 dcmMakeRC

Function name	Arguments	Result	Standard Structure fields
dcmMakeRC	Message number Message severity Error code address	complete error code	

C syntax	`int dcmMakeRC`
	`  (int  messageNumber,`
	`  DCM_Message_Severities  severity,`
	`  int  *errorCode`
	`  );`

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 `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, the returned value will 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.

8.17.11.13 dcmHardErrorRC

Function name	Arguments	Result	Standard Structure fields
dcmHardErrorRC	Message severity		

C syntax	`int dcmHardErrorRC(DCM_Message_Severities severity);`

Returns a return code constructed from the message severity argument. If the message severity is inform or warning (see Table 8-2) the return code is 0. Otherwise the return code has the passed message severity with a message number of 0x00EEEEEE.

8.17.11.14 dcmSetMessageIntercept

Function name	Arguments	Result	Standard Structure fields
dcmSetMessageIntercept	Application-defined function for printing messages		

C syntax	`DCM_Message_Intercept_Type dcmSetMessageIntercept(DCM_Message_Intercept_Type msgfn);`

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 a pointer to the previous message intercept function or 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`.

8.17.11.15 dcmIssueMessage

Function name	Arguments	Result	Standard Structure fields
dcmIssueMessage	*Standard Structure* pointer Message number Message severity Message format string [format arguments]		

C syntax

```
int dcmIssueMessage
  (DCM_STD_STRUCT          *std_struct,
   int                     msgNum,
   DCM_Message_Severities  msgSev,
   char*                   msgFormat [, ...]
  );
```

Prints a message using the current message function in effect. This function assembles from the severity, message number, format, and format arguments a complete DCL message. `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 ensure proper ordering of both DPCM and application messages in the same output stream (see 8.17.11.14).

This function takes a minimum of 4 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 8.11.1). The fourth argument is a format string which follows the conventions of the C `printf` function. The remaining arguments, if any, fulfill 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 8.17.11.12).

This function shall not buffer any messages.

8.17.11.16 dcm_rule_init

Function name	Arguments	Result	Standard Structure fields
dcm_rule_init			

C syntax

```
int dcm_rule_init(DCMTransmittedInfo *xmitStruct,
  DCM_FunctionTable *externals);
```

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 `EXPORT`s and `IMPORT`s, 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 `EXPOSE`s and modeling functions.

After the `root` subrule is loaded and its initialization routine has been called, the application then:

— 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 pointier supplied by the return value dcmBindRule.

8.17.11.17 DCM_new_DCM_STD_STRUCT

Function name	Arguments	Result	Standard Structure fields
DCM_new_DCM_STD_STRUCT			

C syntax	`DCM_new_DCM_STD_STRUCT *DCM_new_STD_STRUCT(void);`

Constructor function to allocate and properly initialize a *Standard Structure*.

8.17.11.18 DCM_delete_DCM_STD_STRUCT

Function name	Arguments	Result	Standard Structure fields
DCM_delete_DCM_STD_STRUCT			

C syntax	`void DCM_delete_DCM_STD_STRUCT(DCM_STD_STRUCT *std_struct);`

Destructor function to free a *Standard Structure*.

8.17.11.19 dcm_setTechnology

Function name	Arguments	Result	Standard Structure fields
dcm_setTechnology	*Standard Structure* pointer Pointer to technology name		

C syntax

```
const char* dcm_setTechnology(DCM_STD_STRUCT *std_struct,
  const char *tech_name);
```

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:

1) If the DPCM has no technology or has a single technology, that technology is selected.
2) 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 8.17.11.25) 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.

8.17.11.20 dcm_getTechnology

Function name	Arguments	Result	Standard Structure fields
dcm_getTechnology	*Standard Structure* pointer		

C syntax

```
const char* dcm_getTechnology(DCM_STD_STRUCT *std_struct);
```

Returns the technology name of the *Standard Structure* in use. Returns a `0` value if completion is unsuccessful.

NOTE — Do not free the result string as it is constant.

8.17.11.21 dcm_getAllTechs

Function name	Arguments	Result	Standard Structure fields
dcm_getAllTechs	*Standard Structure* pointer		

C syntax	`char ** dcm_getAllTechs(DCM_STD_STRUCT *std_struct);`

Returns an array of all technologies named within the current DPCM. Returns a `0` value if completion is unsuccessful.

NOTE — Do not free the result within the calling application, use `dcm_FreeAllTechs` instead.

8.17.11.22 dcm_freeAllTechs

Function name	Arguments	Result	Standard Structure fields
dcm_freeAllTechs	*Standard Structure* pointer Array pointer		

C syntax	`void dcm_freeAllTechs(DCM_STD_STRUCT *std_struct, char **techArray);`

Frees storage occupied by the string array returned by a call to `dcm_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.

8.17.11.23 dcm_isGeneric

Function name	Arguments	Result	Standard Structure fields
dcm_isGeneric	*Standard Structure* pointer		

C syntax	`int dcm_isGeneric(DCM_STD_STRUCT *std_struct);`

Returns whether or not the *Standard Structure* is currently pointing to the generic technology. A non-zero 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.

8.17.11.24 dcm_mapNugget

Function name	Arguments	Result	Standard Structure fields
dcm_mapNugget	*Standard Structure* pointer Technology name		

C syntax

```
int dcm_mapNugget
  (DCM_STD_STRUCT       *std_struct,
   const char           *tech_name,
   DCM_TechFamilyNugget *tech_nugget
  );
```

Maps the passed technology name into the "nugget", `tech_nugget`. This enables rapid technology switching with the `dcm_takeMappingOfNugget` function.

NOTES

1 — The application is responsible for setting the current technology in the DCL *Standard Structure* when the DPCM contains multiple technologies. This is not required if the DPCM contains only one technology.

2 — `dcm_setTechnology` (see 8.17.11.19) may also be used to set the current technology.

8.17.11.25 dcm_takeMappingOfNugget

Function name	Arguments	Result	Standard Structure fields
dcm_takeMappingOfNugget	*Standard Structure* pointer Technology nugget		

C syntax

```
int dcm_takeMappingOfNugget
  (DCM_STD_STRUCT       *std_struct,
   DCM_TechFamilyNugget *tech_nugget
  );
```

Sets the *Standard Structure* argument to use the technology for which `tech_nugget` was computed (using `dcm_mapNugget`).

A non-zero return code indicates the function did not successfully complete. A zero return code indicates success.

8.17.11.26 dcm_registerUserObject

Function name	Arguments	Result	Standard Structure fields
dcm_registerUserObject	*Standard Structure* pointer Pointer to application structure to be registered		

C syntax

```
int dcm_registerUserObject
  (DCM_STD_STRUCT *std_struct,
   void           *app_struct_to_register
  );
```

Registers an application-specific data structure with the passed *Standard Structure*. A registered user object is a structure which 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 8.17.11.27). This function is passed a pointer to the application structure to be registered. A non-zero return code indicates the function did not successfully complete. A zero return code indicates successful registration.

8.17.11.27 dcm_DeleteRegisteredUserObjects

Function name	Arguments	Result	Standard Structure fields
dcm_DeleteRegisteredUserObjects	*Standard Structure* pointer		

C syntax

```
void dcm_DeleteRegisteredUserObjects(DCM_STD_STRUCT *std_struct);
```

Deletes all the registered user objects that were registered to the specified *Standard Structure*.

8.17.11.28 dcm_DeleteOneUserObject

Function name	Arguments	Result	Standard Structure fields
dcm_DeleteOneUserObject	Pointer to object to be deleted		

C syntax

```
void dcm_DeleteOneUserObject
  (DCM_STD_STRUCT *std_struct,
   void           *userObject
  );
```

Locates and deletes the user object contained within the specified *Standard Structure*. See `dcm_DeleteRegisteredUserObjects` for a description of registered user object.

8.17.12 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.

8.17.12.1 delay

Function name	Arguments	Result	Standard Structure fields
delay	*Standard Structure* pointer	DCM_DELAY_REC pointer	CellName fromPoint toPoint calcMode slew.early slew.late block sourceEdge sinkEdge sourceMode sinkMode pathData (timing arc or pin specific) cellData (timing)

C syntax

```
typedef struct
  {float early, late;}
  DCM_DELAY_REC;
int delay
  (DCM_STD_STRUCT *std_struct,
   DCM_DELAY_REC  *delay_value
  );
```

This function is called by the application to calculate the delay for a modeled arc. 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 8.17.13.1), 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 `dcmBindRule` (see 8.17.11.6).

8.17.12.2 slew

Function name	Arguments	Result	Standard Structure fields
slew	*Standard Structure* pointer	DCM_SLEW_REC pointer	CellName fromPoint toPoint calcMode slew.early slew.late block sourceEdge sinkEdge sourceMode sinkMode pathData (timing arc or pin specific) cellData (timing)

C syntax

```
typedef struct
  {float early, late;}
  DCM_SLEW_REC;
int slew
  (DCM_STD_STRUCT *std_struct,
   DCM_SLEW_REC   *slew_value
  );
```

This function is called by the application to calculate the slew for a modeled arc. 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 8.17.13.1), 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 `dcmBindRule` (see 8.17.11.6).

8.17.12.3 check

Function name	Arguments	Result	Standard Structure fields
check	*Standard Structure* pointer	DCM_CHECK_REC pointer	CellName fromPoint toPoint calcMode slew.early slew.late sourceEdge sinkEdge sourceMode sinkMode pathData (timing-arc-specific) cellData (timing)

C syntax

```
typedef struct
  {float bias;}
  DCM_CHECK_REC;
int check
  (DCM_STD_STRUCT *std_struct,
   DCM_CHECK_REC  *test_bias
  );
```

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 8.13.1).

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 8-8). 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 `dcmBindRule` (see 8.17.11.6).

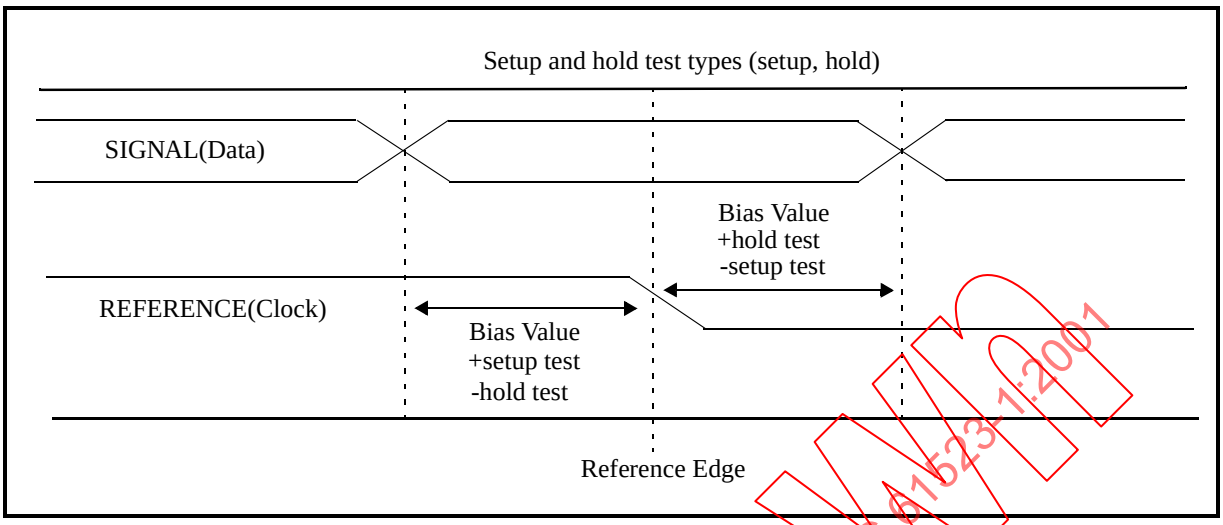


Figure 8-8 Bias calculation

Clock separation implies the reference and signal edges are two different clocks, as shown in Figure 8-9.

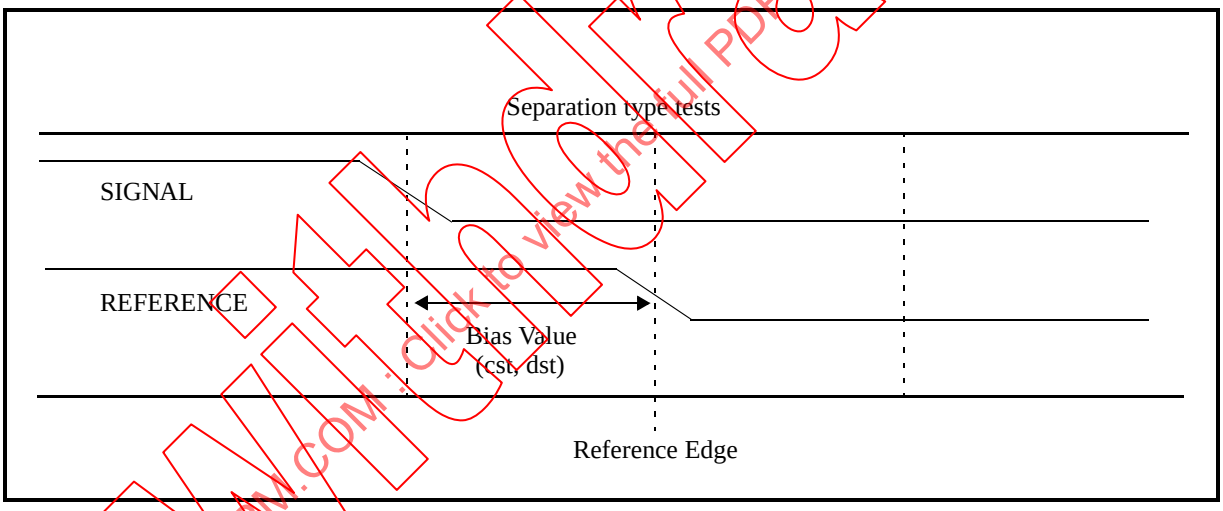


Figure 8-9 Clock separation

Clock pulse width implies the signal and reference are the same clock but different edges (see Figure 8-10). 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`.

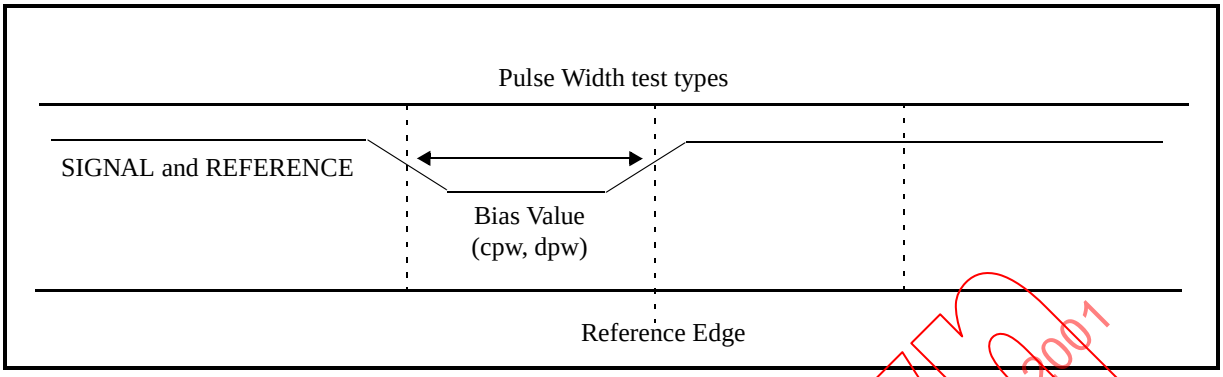


Figure 8-10 Different edges

8.17.13 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 of these modeling callback functions shall be implemented by any standards-compliant application. The modeling functions return the number 0 (zero) on success and non-zero on error or failure.

8.17.13.1 modelSearch

Function name	Arguments	Result	Standard Structure fields
modelSearch	*Standard Structure* pointer		block CellName inputPins outputPins inputPinCount outputPinCount nodes nodeCount

C syntax	`int modelSearch(DCM_STD_STRUCT *std_struct);`

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 two approaches:

a) Choose to always elaborate (i.e., call `modelSearch`) models for every instance
b) 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 which 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 8.17.13). The application shall save this information since 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. There are three different types of timing arcs that can be generated:

— within in a cell,
— from a cell's output to all receivers it drives, or
— 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 inter-cell and intra-cell 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 of 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 `dcmBindRule` (see 8.17.11.6).

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).

8.17.13.2 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 8-16. `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 8-16—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 8.17.13.5).

The mode operators for delay and slew are shown in Table 8-17.

Table 8-17—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_EarlyMode	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 8-18.

Table 8-18—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.

8.17.13.3 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:

— 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:

— 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.

8.17.13.4 Edge Propagation communication to the application

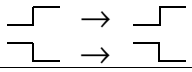
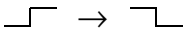
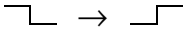
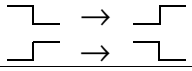
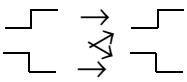
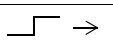
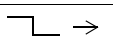
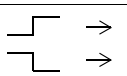
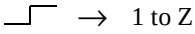
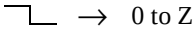
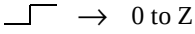
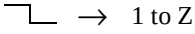
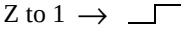
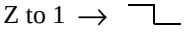
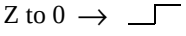
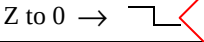
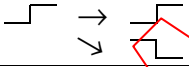
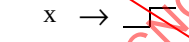
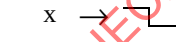
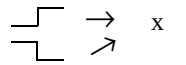
The DPCM represents edge propagation as a pair of enumerations. Table 8-19 is a representative sampling of enumeration pairs which can exist. The enumeration pairs are the same for test segments as they are for propagation segments. Also,

— 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 8-19—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

Table 8-19—Enumeration pairs *(continued)*

Propagation	Starting enumeration/ Reference enumeration	Ending enumeration/ Signal enumeration	Meaning
	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
	DCM_RisingEdge	DCM_Terminate	Rise to Terminate
	DCM_FallingEdge	DCM_Terminate	Fall to Terminate
	DCM_RisingEdge DCM_FallingEdge	DCM_Terminate DCM_Terminate	Rise to Terminate and Fall to Terminate
	DCM_RisingEdge	DCM_OneToZ	Rise to One_To_Z
	DCM_FallingEdge	DCM_ZeroToZ	Fall to Zero_To_Z
	DCM_RisingEdge	DCM_ZeroToZ	Rise to Zero_To_Z
	DCM_FallingEdge	DCM_OneToZ	Fall to One_To_Z
	DCM_ZtoOne	DCM_RisingEdge	Z_to_One to Rise
	DCM_ZtoOne	DCM_FallingEdge	Z_to_One to Fall
	DCM_ZtoZero	DCM_RisingEdge	Z_to_Zero to Rise
	DCM_ZtoZero	DCM_FallingEdge	Z_to_Zero to Fall
	DCM_BothEdges	DCM_RisingEdge	Both to Rise
	DCM_RisingEdge	DCM_BothEdges	Rise to Both
	DCM_FallingEdge	DCM_BothEdges	Fall to Both
	DCM_Terminate	DCM_RisingEdge	Terminate to Rise
	DCM_Terminate	DCM_FallingEdge	Terminate to Fall
	DCM_Terminate	DCM_BothEdges	Terminate to Both
	DCM_BothEdges	DCM_Terminate	Both to Terminate
	DCM_Terminate	DCM_BothEdges	Terminate to Both

8.17.13.5 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 8-20 on page 261 is a representative sampling of enumeration pairs which are allowed.

Table 8-20—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).

8.17.13.6 newTimingPin

Function name	Arguments	Result	Standard Structure fields
newTimingPin	*Standard Structure* pointer Pointer to node name	Pointer to application's node structure	pathData (timing)

C syntax

```
int newTimingPin
  (DCM_STD_STRUCT   *std_struct,
   DCM_HandleStruct *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 8.17.13.1).

8.17.13.7 newDelayMatrixRow

Function name	Arguments	Result	Standard Structure fields
newDelayMatrixRow	*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	Pointer to application structure for the delay matrix	

C syntax

```
int newDelayMatrixRow
  (DCM_STD_STRUCT       *std_struct,
   DCM_HANDLE           *delayMatrix,
   DCM_EdgeTypes        edge1,
   DCM_PropagationTypes mode1,
   DCM_EdgeTypes        edge2,
   DCM_PropagationTypes mode2
  );
```

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`; signalling 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(std_struct,
      &dpcmDelayHandle,
      DCM_RisingEdge,
      DCM_EarlyMode,
      DCM_SameEdge,
      DCM_SameMode ); ...
newDelayMatrixRow(std_struct,
```

```
        &dpcmDelayHandle,
        DCM_FallingEdge,
        DCM_LateMode,
        DCM_SameEdge,
        DCM_SameMode ); ...
```

8.17.13.8 newNetSinkPropagateSegments

Function name	Arguments	Result	Standard Structure fields
newNetSinkPropagateSegments	*Standard Structure* pointer Node (sink) pin pointer Import pin pointer Delay matrix (created by *newDelayMatrixRow*)		pathData (power or timing pin-specific)

C syntax

```
int newNetSinkPropagateSegments
  (DCM_STD_STRUCT *std_struct,
   DCM_HANDLE     importPin,
   DCM_HANDLE     sinkPin,
   DCM_HANDLE     delayMatrix
  );
```

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:

a) An `INPUT` function generates calls where the node (sink) pin and the import pin arguments are the same pointer.

b) 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.

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 the examples shown in Figure 8-11.

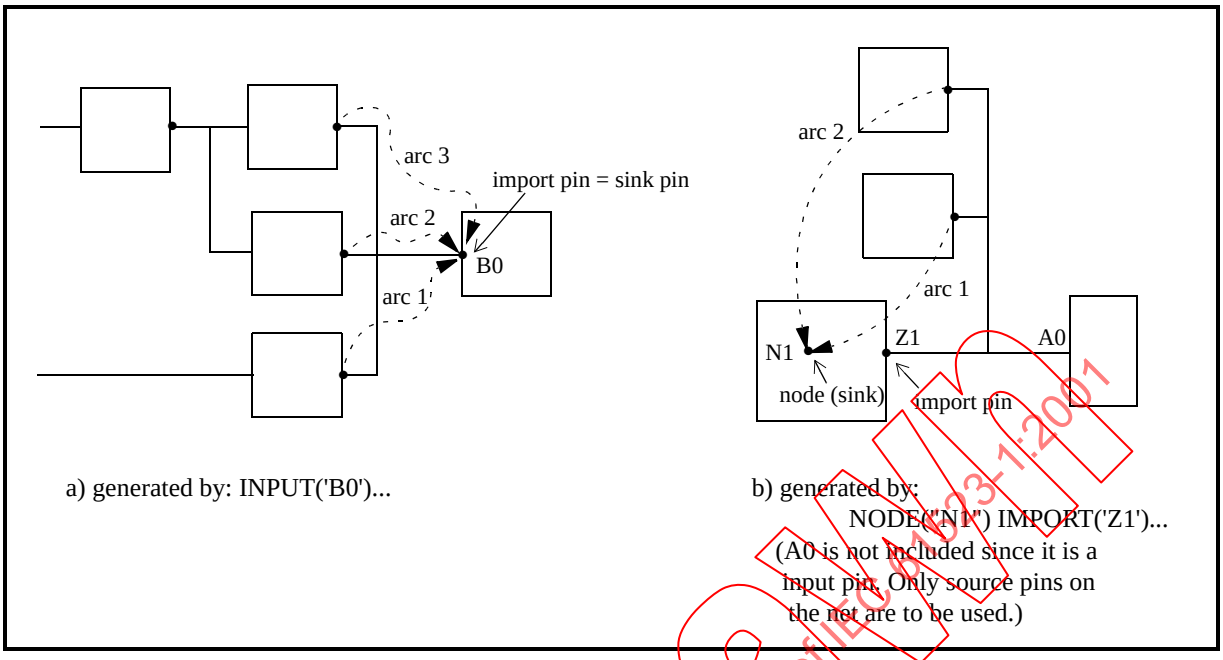


Figure 8-11 Sample MODELPROC results

8.17.13.9 newNetSourcePropagateSegments

Function name	Arguments	Result	Standard Structure fields
newNetSourcePropagateSegments	*Standard Structure* pointer node (source) pin pointer export pin pointer Delay matrix (created by *newDelayMatrixRow*)		pathData (power or timing pin-specific)

C syntax

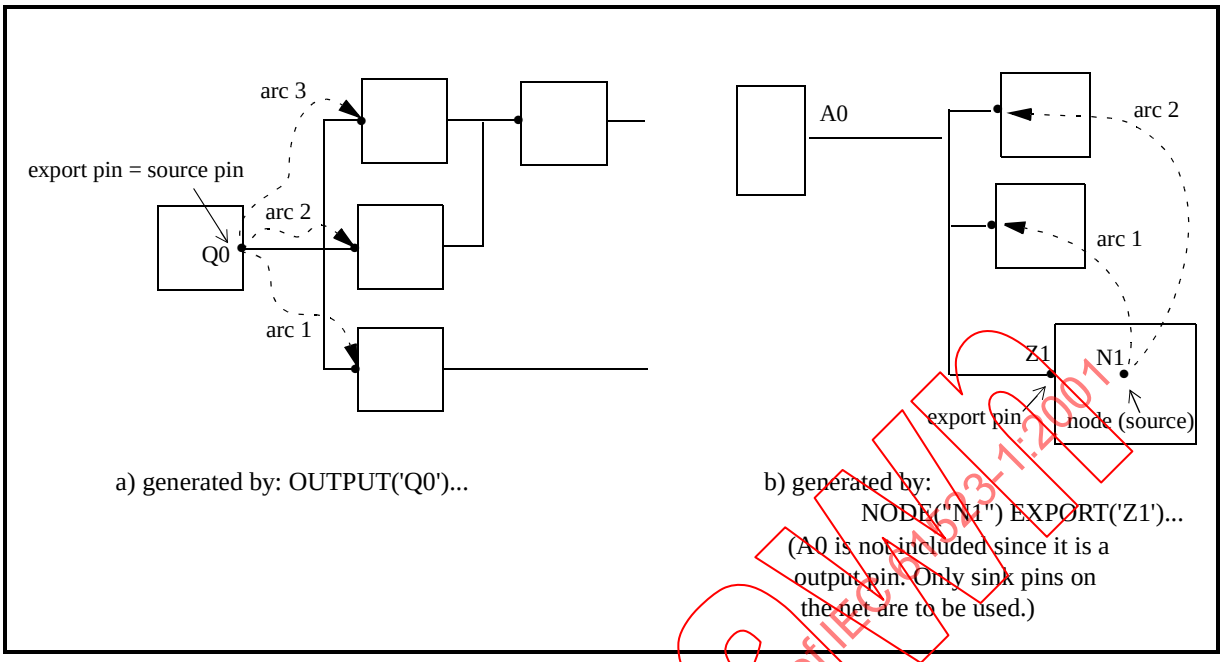
```
int newNetSourcePropagateSegments
 (DCM_STD_STRUCT  *std_struct,
  DCM_HANDLE      sourcePin,
  DCM_HANDLE      exportPin,
  DCM_HANDLE      delayMatrix
 );
```

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. 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:

a) An `OUTPUT` function generates calls where the export pin and the node (source) pin arguments are the same pointer.
b) 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.

Both of these are demonstrated in the examples shown in Figure 8-12.



a) generated by: OUTPUT('Q0')...

b) generated by: NODE("N1") EXPORT('Z1')... (A0 is not included since it is a output pin. Only sink pins on the net are to be used.)

Figure 8-12 Additional MODELPROC results

8.17.13.10 newPropagateSegment

Function name	Arguments	Result	Standard Structure fields
newPropagateSegment	*Standard Structure* pointer Driver (source) pin Receiver (sink) pin Delay matrix (created by *newDelayMatrixRow*)	New timing arc handle pointer	pathData (timing-arc-specific)

C syntax

```
int newPropagateSegment
(DCM_STD_STRUCT *std_struct,
 DCM_HANDLE     *output,
 DCM_HANDLE     sourcePin,
 DCM_HANDLE     sinkPin,
 DCM_HANDLE     delayMatrix
);
```

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 `clkflg` value is made available to the application at `pathData->pcdb->clkflg`.

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.

8.17.13.11 newTestMatrixRow

Function name	Arguments	Result	Standard Structure fields
newTestMatrixRow	*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	Pointer to the application's test matrix structure	

C syntax

```
int newTestMatrixRow
  (DCM_STD_STRUCT       *std_struct,
   DCM_HANDLE           *testMatrix,
   DCM_EdgeTypes        edge1,
   DCM_PropagationTypes mode1,
   DCM_EdgeTypes        edge2,
   DCM_PropagationTypes mode2,
   DCM_TestTypes        testType
  );
```

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 6-7.

8.17.13.12 newAltTestSegment

Function name	Arguments	Result	Standard Structure fields
newAltTestSegment	*Standard Structure* pointer From pin To pin Test matrix (created by *newTestMatrixRow*)	Pointer to the test arc handle	pathData (timing-arc-specific)

C syntax

```
int newAltTestSegment
  (DCM_STD_STRUCT *std_struct,
   DCM_HANDLE     *output,
   DCM_HANDLE     clock,
   DCM_HANDLE     data,
   DCM_HANDLE     testMatrix
  );
```

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 propa-

gation 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.

8.17.13.13 Interactions between interconnect modeling and modeling functions

There are two strategies for modeling interconnect in DCL:

— use `OUTPUT` and/or `INPUT` statements in the model
— use the `DEFAULT` delay and slew calculation statement.

The application shall determine if 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 correctly handle interconnect calculations:

a) 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.

b) At interconnect delay/slew calculation time:

   1) For situations when non-zero 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.
   2) 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.
   3) For situations where only zero (`0`) values for `delayMatrix` are received for call backs 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.
   4) For situations where no call backs 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.

c) The application shall ensure all the fields required by the call to delay and slew are filled, and for interconnect calculation, the application shall also insure `allName` and `block` fields such as `cell`, `cellQual`, and `modelDomain` represent the cell driving the net.

8.18 86Standard structure (dcmstd_stru.h) file

This section lists the `std_stru.h` file.

```
#ifndef _DCMSTDSTRUCT_H
#define _DCMSTDSTRUCT_H
/*****************************************************************
** INCLUDE NAME..... std_stru.h
**
** PURPOSE.........
** This is the DCL standard structure definition.
**
** NOTES.............
**
** ASSUMPTIONS.......
**
** RESTRICTIONS.....
**
** LIMITATIONS......
**
** DEVIATIONS.......
**
** AUTHOR(S)......... H. John Beatty, Peter C. Elmendorf
**
** CHANGES:
**
*****************************************************************/
#include <dcmpltfm.h>
#include <dcmutab.h>
#include <dcmnewdef.h>
/*****************************************************************
** Forward structures.
*****************************************************************/
typedef struct DCM_STD_STRUCT    DCM_STD_STRUCT;
typedef struct DCM_PathDataBlock DCM_PathDataBlock;
typedef struct DCM_CellDataBlock DCM_CellDataBlock;
typedef struct DCM_FunctionTable DCM_FunctionTable;

/*****************************************************************
** Storage management for user C code.
*****************************************************************/
DCM_XC void *dcmMalloc(size_t);
DCM_XC void  dcmFree(void *);
DCM_XC void *dcmRealloc(void *it, size_t size);

/*****************************************************************
** Set memory management intercepts - for applications.
**
** The memory management intercept can be called ONLY ONCE, and shall
** be called before calling dcmLoadRule() to load the very first DCM.
**
** All three parms shall be non-NULL or else the request is ignored.
**
```

```
** If called after dcmLoadRule(), messages will be issued and the call
** will be ignored.
*****************************************************************/

typedef void * (*DCM_Malloc_Type)  (size_t);
typedef void   (*DCM_Free_Type)    (void *);
typedef void * (*DCM_Realloc_Type) (void *, size_t);

DCM_XC int dcmSetNewStorageManager(DCM_Malloc_Type   new_malloc,
                DCM_Free_Type     new_free,
                DCM_Realloc_Type  new_realloc);

/*****************************************************************
** Set message handling intercept - for applications (usually).
**
** DCM_Message_Severities tells what severity level the message is.
*****************************************************************/
typedef enum DCM_Message_Severities {
  DCM_Msg_Inform    = 0x00,
  DCM_Msg_Warning   = 0x01,
  DCM_Msg_Error     = 0x02,
  DCM_Msg_Severe    = 0x03,
  DCM_Msg_Terminate = 0x04
} DCM_Message_Severities;

/*****************************************************************
** Type for the user's message intercept function.
**
** NOTE: the message text is place in a buffer that is reused upon
** issuing the next message.  Copy the information out of the buffer
** if you intend to keep it.
*****************************************************************/
typedef void (*DCM_Message_Intercept_Type)
     (int                     msgnum,   /* message number.            */
      DCM_Message_Severities sev,      /* message severity.          */
      const char             *msgText); /* message text.              */

/*****************************************************************
** This function sets a new message intercept and returns the current
** intercept.Intercepts may be changed as often as desired during a run.
**
** NULL means "no intercept in use", is a legal parameter, and also a
** legal return value.
*****************************************************************/
DCM_XC
DCM_Message_Intercept_Type
dcmSetMessageIntercept(DCM_Message_Intercept_Type);

/*****************************************************************
** This function issues an official-looking message from DCM C code
** or from the DCM message built-in function (when that exists.)
*****************************************************************/
DCM_XC
int dcmIssueMessage                /* Returns standard rc format.     */
```

```
    (const DCM_STD_STRUCT   *unused, /* SO it looks like a DCM statment.*/
     int                     msgNum, /* user's message number.          */
     DCM_Message_Severities  sev,    /* message severity.               */
     const char             *format, /* printf format.                  */
     ...);                           /* variable number of args.        */

/*********************************************************************
** Resource resolution, for applications or DCM C-code.
*********************************************************************/
DCM_XC char *dcmGetResource(const char *resourceName,
   const char *description);

/*********************************************************************
** Return code assistants.
*********************************************************************/

/*********************************************************************
** Returns "hard error" that breaks EXPOSE chaining
** User is allowed to set the severity.
*********************************************************************/
DCM_XC int dcmHardErrorRC                /* Returns standard rc format. */
  (DCM_Message_Severities  sev);         /* message severity.           */

/*********************************************************************
** Returns "soft error" that stops the function at hand, but
** allows EXPOSE chaining and DEFAULT clauses to work.
**
** User is allowed to set the severity.
**
** Severe or Terminate levels will still disallow EXPOSE chaining
** and DEFAULT clauses.
*********************************************************************/
DCM_XC int dcmSoftErrorRC                /* Returns standard rc format. */
  (DCM_Message_Severities  sev);         /* message severity.           */

/*********************************************************************
** Makes a standard style return code from a given message number
** and severity. Adds 10,000 to the msgNum just like dcmIssueMessage()
** does.
*********************************************************************/
DCM_XC int dcmMakeRC                     /* Returns standard rc format. */
  (int                     msgNum,     /* user's message number.      */
   DCM_Message_Severities  sev,        /* message severity.           */
   int                    *userNum);   /* Number after massage.       */

/*********************************************************************
** Given a return code, return..
**  1 if the return code is one that breaks EXPOSE chaining.
**  0 if the return code is one that does NOT break EXPOSE chaining.
*********************************************************************/
DCM_XC int dcmBreaksExposeChains(int rc);

/*********************************************************************
** Given a return code, return..
```

```
**  1 if the return code is severe enough to stop DEFAULT clauses
**     from working.
**  0 if the return code is not that severe.
*****************************************************************************/
DCM_XC int dcmBreaksDefaultClauses(int rc);

/*****************************************************************************
** Technology nugget.
** DO NOT MODIFY THE CONTENTS!
*****************************************************************************/
typedef struct DCM_TechFamilyNugget {
  const    char *name;
           int   num;
  unsigned int   dcmInfo;
} DCM_TechFamilyNugget;

/*****************************************************************************
** This is how DCM views a database handle.
*****************************************************************************/
typedef struct DCM_HandleStruct {
  char *name;                          /* Name of the object.         */
                                       /* We don't care what the rest */
                                       /* of the object looks like.   */
} DCM_HandleStruct;
typedef DCM_HandleStruct *DCM_HANDLE;

/*****************************************************************************
** DCM_HANDLE to name conversion.
*****************************************************************************/
#define DCM_NameInHandle(h) ((h)->name)

/*****************************************************************************
** Other handy definitions.
*****************************************************************************/
typedef double              DCM_DOUBLE;
typedef float               DCM_FLOAT;
typedef int                 DCM_INTEGER;
typedef double              DCM_NUMBER;
typedef DCM_HandleStruct *DCM_PIN;
typedef DCM_PIN           *DCM_PINLIST;
typedef char              *DCM_STRING;
typedef void              *DCM_VOID;

/*****************************************************************************
** COMPLEX numbers.
*****************************************************************************/
typedef struct DCM_COMPLEX {
  DCM_DOUBLE realPart;
  DCM_DOUBLE imagPart;
} DCM_COMPLEX;

/*****************************************************************************
** The delay record.
*****************************************************************************/
```

```
typedef struct DCM_DELAY_REC {
  DCM_FLOAT early;                      /* The early delay value.      */
  DCM_FLOAT late;                       /* The late delay value.       */
} DCM_DELAY_REC;

/*****************************************************************
** The slew record.
*****************************************************************/
typedef struct DCM_SLEW_REC {
  DCM_FLOAT early;                      /* The early delay value.      */
  DCM_FLOAT late;                       /* The late delay value.       */
} DCM_SLEW_REC;

/*****************************************************************
** The check record.
*****************************************************************/
typedef struct DCM_CHECK_REC {
  DCM_FLOAT bias;                       /* The time difference.        */
} DCM_CHECK_REC;

/*****************************************************************
** Typedef for a general function.
*****************************************************************/
#ifdef __cplusplus
typedef int (*DCM_GeneralFunction)(...);
#else
typedef int (*DCM_GeneralFunction)();
#endif

typedef DCM_GeneralFunction FUNCTION;

#include <dcmstate.h>                   /* The state block.            */

/*****************************************************************
** The edge types.
*****************************************************************/
typedef enum DCM_EdgeTypes {
  DCM_RisingEdge,
  DCM_FallingEdge,
  DCM_BothEdges,
  DCM_SameEdge,
  DCM_ComplimentEdge,
  DCM_Terminate,
  DCM_TerminateBoth,
  DCM_OneToZ,
  DCM_ZtoOne,
  DCM_ZeroToZ,
  DCM_ZtoZero,
  DCM_AllEdges
} DCM_EdgeTypes;

/*****************************************************************
** The propagation types.
*****************************************************************/
```

```
typedef enum DCM_PropagationTypes {
  DCM_EarlyMode,
  DCM_LateMode,
  DCM_BothModes,
  DCM_SameMode,
  DCM_ComplimentMode,
  DCM_NoMode
} DCM_PropagationTypes;

/*************************************************************************
** Calculation mode: best case, worst case, nominal.
*************************************************************************/
typedef enum DCM_CalculationModes {
  DCM_BestCase,
  DCM_WorstCase,
  DCM_NominalCase
} DCM_CalculationModes;

/*************************************************************************
** The test types.
*************************************************************************/
typedef enum DCM_TestTypes {
  DCM_SetupTest,
  DCM_HoldTest,
  DCM_ClockPulseWidthTest,
  DCM_ClockSeparationTest,
  DCM_DataPulseWidthTest,
  DCM_DataSeparationTest,
  DCM_ClockGatingPulseWidthTest,
  DCM_ClockGatingHoldTest,
  DCM_ClockGatingSetupTest,
  DCM_EndOfCycleTest,
  DCM_DataHoldTest,
  DCM_RecoveryTest,
  DCM_RemovalTest,
  DCM_SkewTest,
  DCM_NoChangeTest
} DCM_TestTypes;

#include <dcmgarray.h>
#include <dcmgstruct.h>

/*************************************************************************
** Typedef for a general DCM statement function.
*************************************************************************/
typedef int (*DCM_StatementFunction)(DCM_STD_STRUCT *, void *, ...);

/*************************************************************************
** STORE package function.
*************************************************************************/
typedef int (*DCM_StoreFunction)(DCM_STD_STRUCT *);

/*************************************************************************
** Typedef for the delay search function.
```

```
1       ******************************************************************/
        typedef int (*DCM_DelayFunctionType)(DCM_STD_STRUCT *, DCM_DELAY_REC *);

        /******************************************************************
5       ** Typedef for the slew function.
        ******************************************************************/
        typedef int (*DCM_SlewFunctionType)(DCM_STD_STRUCT *, DCM_SLEW_REC *);

        /******************************************************************
10      ** Typedef for the check function.
        ******************************************************************/
        typedef int (*DCM_CheckFunctionType)(DCM_STD_STRUCT *, DCM_CHECK_REC *);

        /******************************************************************
15      ** Typedef for the method mapper function.
        ******************************************************************/
        typedef struct DCM_MethodMap_DummyStruct {
          int a;                            /* Actual data are hidden.        */
        } DCM_MethodMap_DummyStruct;        /* Dummy struct for unique typing.*/
20
        typedef DCM_MethodMap_DummyStruct *DCM_MethodMap;

        typedef DCM_MethodMap (*DCM_MethodMapFunctionType)
             (DCM_STD_STRUCT *,             /* I: -> std struct to use.    */
25            const char     *);           /* I: method class name.       */

        /******************************************************************
        ** Typedef for the method caller function.
        ******************************************************************/
30      typedef int (*DCM_MethodCallFunctionType)
             (DCM_STD_STRUCT *,             /* I: -> std struct to use.       */
              void           *rtn,          /* O: -> results area for method. */
              DCM_MethodMap);               /* I: Method map for method class.*/

35      /******************************************************************
        ** Function to return all the names of the method classes.
        ** ASSUMPTIONS..... NEVER FREE THE RESULTS OR CHANGE THE DATA!
        ******************************************************************/
        DCM_XC int dcmFindMethodClassNames
40        (DCM_STD_STRUCT *,              /* I: -> std structure.           */

           const char    ***,             /* O: -> vec of char * of names. */
                                          /* NULL terminated vector.       */
                                          /* Do NOT free/modify this space!*/
45
           const char     *);             /* I: name of tech family.       */
                                          /*  NULL == “all tech families. ”*/

        /******************************************************************
50      ** This structure defines ONE unit of STORE() function information.
        ******************************************************************/
        typedef struct DCM_SFI {
          DCMTableDescriptor **table;        /* -> table if STORE() of table.*/
          DCM_StoreFunction    storer;       /* Who created the recall data.  */
```

```
  char                *storerName;   /* -> name of func doing store.  */
  int                  recordSize;   /* Size of data record.          */
} DCM_SFI;

/*************************************************************************
** Cell Constant Data Block
*************************************************************************/
typedef struct DCM_CCDB {
  /***********************************************************************
  ** Items below this line are used by DCM for internal maintenance and
  ** consistency checking.  DO NOT TOUCH!
  ***********************************************************************/
  int             sfiCount;           /* # of stored function results.*/
  DCM_SFI        *sfi;                /* -> store function info.      */
  int (*destructor)(DCM_CellDataBlock *); /* special destructor.      */
  DCM_GeneralFunction *methods;       /* -> methods block if any.     */
  void           *anchor;             /* Anchor.                      */
  unsigned short  reserved0;
  unsigned short  flags;
  /***********************************************************************
  ** For use in possible future extensions.
  ***********************************************************************/
  void           *reserved1;
  void           *reserved2;
  void           *reserved3;
  void           *reserved4;
} DCM_CCDB;

/*************************************************************************
** These flags control cell constant space.
*************************************************************************/
typedef enum DCM_CCDBFlags {
  DCM_CCDBInconsistent      = 0x0001,
  DCM_CCDBInconsistentStore = 0x0002
} DCM_CCDBFlags;

/*************************************************************************
** Cell Data Block
*************************************************************************/
struct DCM_CellDataBlock {
  void                **recallData;    /* points to the stored data    */
  /***********************************************************************
  ** -> cell constants data block.
  ** The CCDB contains both user and DCM control data which is constant
  ** for each circuit.
  ***********************************************************************/
  DCM_CCDB            *ccdb;
  int                  usageCount;   /* # of times this block is used.*/
  unsigned short       reserved1;
  unsigned short       flags;        /* Control flags.                */
  /***********************************************************************
  ** For use in possible future extensions.
  ***********************************************************************/
```

```
  void                *reserved2;
  void                *reserved3;
  void                *reserved4;
};

/*****************************************************************
** Marks that STORE() completed OK.
** Valid for both CellData and PathData.
** MUST BE the value 0x0004!
*****************************************************************/
#define DCM_Pdb_Cdb_StoreComplete 0x0004
/*****************************************************************
** These flags control cell data space.
*****************************************************************/
typedef enum DCM_CellDataFlags {
  /***************************************************************
  ** Means that the modelproc is CAPABLE of being inconsistent.
  ** Therefore, the model must be called and made every time.
  **   (Example: possible path differences.)
  ** All MODELPROCs executed by 2.2 or earlier DCM will leave
  ** this bit set.
  ***************************************************************/
  DCM_CdbInconsistentModel      = 0x0001,
  /***************************************************************
  ** Means that an inconsistent STORE actually executed in this
  ** modelling.(Not potential, but actual.)
  ***************************************************************/
  DCM_CdbInconsistentStore      = 0x0002,
  /***************************************************************
  ** Marks that STORE() completed OK.
  ** MUST BE the value 0x0004!
  ** ==> MAKE SURE this is the same bit as that used for
  **     DCM_PdbStoreComplete!
  ***************************************************************/
  DCM_CdbStoreComplete          = DCM_Pdb_Cdb_StoreComplete,
  /***************************************************************
  ** These bits are set to 1 when their respective qualifier fields
  ** matched the default qualifier (*) during model search.
  ***************************************************************/
  DCM_CdbCellDefault            = 0x0010,
  DCM_CdbCellQualDefault        = 0x0020,
  DCM_CdbModelDomainDefault     = 0x0040,
  /***************************************************************
  ** Means that the PROPERTIES stmt has not run yet.
  ** Remains set when 2.2 or earlier DCM is used to make the model.
  ***************************************************************/
  DCM_CdbPropertiesNotSet       = 0x8000
} DCM_CellDataFlags;

/*****************************************************************
** Destroy DCM_CellDataBlocks.
** Obeys the built-in destructor ptr.
** Always call this function to delete DCM_CellDataBlock items.
*****************************************************************/
```

```
DCM_XC int dcmDeleteCellDataBlock(DCM_CellDataBlock *);

/*************************************************************************
** Path Constant Data Block
*************************************************************************/
typedef struct DCM_PCDB {
  DCM_STRING clkflg;                /* clock identification string   */
  DCM_STRING ckttype;               /* circuit identification string */
  int       delayAdj;               /* holds the numbers of cycles   */
                                    /* this path should be adjusted  */
                                    /* by.                           */
  DCM_DelayFunctionType delay;      /* Delay function ptr.           */
  DCM_SlewFunctionType  slew;       /* Slew function ptr.            */
  /*************************************************************************
  ** Items below this line are used by DCM for internal maintenance and
  ** consistency checking.  DO NOT TOUCH!
  *************************************************************************/
  int             ci;               /* Consistency index.            */
  int             sfiCount;         /* # of stored function results.*/
  DCM_SFI        *sfi;              /* -> store function info.       */
  int (*destructor)(DCM_PathDataBlock *); /* special destructor.      */
  void           *anchor;           /* Anchor.                       */
  DCM_GeneralFunction *methods;     /* -> methods block if any.      */
  /*************************************************************************
  ** For use in possible future extensions.
  *************************************************************************/
  void           *reserved2;
  void           *reserved3;
  void           *reserved4;
} DCM_PCDB;

/*************************************************************************
** These flags control path data space.
*************************************************************************/
typedef enum DCM_PathDataFlags {
  DCM_PathFromMalloc = 0x0001,      /* PATH name was malloc'd.       */
  DCM_PdbInCT        = 0x0002,      /* PDB is in a consistency tree.*/
  /*************************************************************************
  ** Marks that STORE() completed OK.
  ** MUST BE the value 0x0004!
  ** ==> MAKE SURE this is the same bit as that used for
  **     DCM_CdbStoreComplete!
  *************************************************************************/
  DCM_PdbStoreComplete = DCM_Pdb_Cdb_StoreComplete,
  DCM_NewerPCDB       = 0x0008      /* PCDB is newer style.          */
} DCM_PathDataFlags;

/*************************************************************************
** Path Data Block.
*************************************************************************/
struct DCM_PathDataBlock {
  /*************************************************************************
  ** This information is intended for timer or application use.
  *************************************************************************/
```

```
    DCM_STRING path;                  /* path name under calculation    */
                                      /* this variable is set by the    */
                                      /* calculator during model build  */
                                      /* and may be ignored by other    */
                                      /* programs                       */
    void     **recallData;            /* points to store clause data    */
    /*********************************************************************
    ** -> path constants data block.
    ** The PCDB contains both user and DCM control data which is 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       usageCount;              /* # of times this block is used.*/
    short     flags;                   /* Control flags.                */
    /*********************************************************************
    ** This section is present ONLY for TEST segments.
    ** This space does NOT exist for other segments.
    *********************************************************************/
    short     cycle_adj;               /* Cycle adjust.                 */
    short     corrind;                 /* Correlation index.            */
};

/*************************************************************************
** Destroy DCM_PathDataBlocks.
** Obeys the built-in destructor ptr.
** Always call this function to delete DCM_PathDataBlock items.
*************************************************************************/
DCM_XC int dcmDeletePathDataBlock(DCM_PathDataBlock *);

/*************************************************************************
** Direct functions that implement the methods in all cases.
*************************************************************************/
#define dcm_FoundDefaultCell(std) \
((std)->cellData->flags & DCM_CdbCellDefault)

#define dcm_FoundDefaultCellQual(std) \
  ((std)->cellData->flags & DCM_CdbCellQualDefault)

#define dcm_FoundDefaultModelDomain(std) \
  ((std)->cellData->flags & DCM_CdbModelDomainDefault)

DCM_XC DCM_STD_STRUCT *dcm_new_DCM_STD_STRUCT();

DCM_XC DCM_STD_STRUCT *dcm_assign_DCM_STD_STRUCT(DCM_STD_STRUCT *target,
                                                 const DCM_STD_STRUCT *source);

DCM_XC void            dcm_delete_DCM_STD_STRUCT(DCM_STD_STRUCT *);

DCM_XC const char     *dcm_setTechnology(DCM_STD_STRUCT *,const char *);
DCM_XC const char     *dcm_getTechnology(const DCM_STD_STRUCT *);
```

```
DCM_XC       char    **dcm_getAllTechs(const DCM_STD_STRUCT *);
DCM_XC       char   ***dcm_getAllTechsAsPinlist(const DCM_STD_STRUCT *);

DCM_XC void            dcm_freeAllTechs(const DCM_STD_STRUCT *,char **);
DCM_XC void            dcm_freeAllTechsAsPinlist(const DCM_STD_STRUCT *,
                                                 char ***);

DCM_XC int  dcm_isGeneric(const DCM_STD_STRUCT *);

DCM_XC int  dcm_mapNugget(const DCM_STD_STRUCT *,
                          const char *name,
                          DCM_TechFamilyNugget *);

DCM_XC int  dcm_takeMappingOfNugget(DCM_STD_STRUCT *,
                                    const DCM_TechFamilyNugget *);

DCM_XC int  dcm_registerUserObject(DCM_STD_STRUCT *,const void *);
DCM_XC void dcm_deleteRegisteredUserObjects(DCM_STD_STRUCT *);
DCM_XC void dcm_deleteOneUserObject(DCM_STD_STRUCT *, void *);

/*************************************************************************
** Return nonzero if the path has something stored on it.
*************************************************************************/
DCM_XC int  dcm_isSomethingStored(const DCM_STD_STRUCT *);

/*************************************************************************
** Return nonzero if there is a STORE of a function with the
** given name.
** Note this does not resolve possible name ambiguity.
*************************************************************************/
DCM_XC int  dcm_isThisFunctionStored(const DCM_STD_STRUCT *,
                                     const char *name);

/*************************************************************************
** Return nonzero if this PATH is inconsistent.
** (Means instance-specific data exists.)
*************************************************************************/
DCM_XC int  dcm_isPathInconsistent(const DCM_STD_STRUCT *);

/*************************************************************************
** Return nonzero if this model presents inconsistent STOREs anywhere.
** (Means instance-specific data exists.)
*************************************************************************/
DCM_XC int  dcm_isModelDataInconsistent(const DCM_STD_STRUCT *);

/*************************************************************************
** Return nonzero if this model is possibly inconsistent.
** (Means do NOT attempt to reuse model, must call for model each time.)
*************************************************************************/
DCM_XC int  dcm_isModelPossiblyInconsistent(const DCM_STD_STRUCT *);

/*************************************************************************
** Structure for getting cell list .
*************************************************************************/
```

```
typedef struct dcm_T_dcmCellList {
  DCM_STRING_ARRAY *cellNameArray;
  DCM_STRING_ARRAY *cellQualArray;
  DCM_STRING_ARRAY *model_domainArray;
} dcm_T_dcmCellList;

/*************************************************************************
** Routine for getting cell list .
*************************************************************************/
DCM_XC int dcmCellList(const DCM_STD_STRUCT *std,
             dcm_T_dcmCellList    *rtn);

/*************************************************************************
** The standard structure.
*************************************************************************/
struct DCM_STD_STRUCT {
  unsigned int    dcmInfo;         /* Special DCM state: DO NOT TOUCH!*/
  DCM_StateBlock *dcmStates;       /* Special DCM state: DO NOT TOUCH!*/
  /*************************************************************
  ** Strings.
  *************************************************************/
  DCM_STRING cell;                 /* first circuit qualifier         */
  DCM_STRING cellQual;             /* second circuit qualifier        */
  DCM_STRING modelDomain;          /* third and last circuit qualifier*/
  DCM_STRING ctl;                  /* Specifies latch FLUSH/NOFLUSH.  */
                                   /* (or other things.)              */
  DCM_STRING model_name;           /* the identifing name of the      */
                                   /* model to be called when         */
                                   /* building a circuit schematic    */
                                   /* this variable is set by the     */
                                   /* calculator and is set once as   */
                                   /* the model is entered.           */
  DCM_STRING instantiated;         /* indicates to the calculator     */
                                   /* that the database exists for    */
                                   /* this book. it is an optional    */
                                   /* variable which can be used in   */
                                   /* conditional expressions to      */
                                   /* alter the behavior of the       */
                                   /* calculator                      */
                                   /* 'y' = exists in the database    */
                                   /* 'n' = doesnot exist in the      */
                                   /* database                        */
  DCM_STRING expanded;             /* indicates to the calculator     */
                                   /* that the macro has been         */
                                   /* expanded in the database and    */
                                   /* the model for the macor need    */
                                   /* not be built. in actuality it   */
                                   /* is a variable that can be       */
                                   /* tested in the logical           */
                                   /* expressions where               */
                                   /* 'y' = expanded and 'n' = not    */
  /*************************************************************
  ** Handles.
  *************************************************************/
```

```
        DCM_HANDLE   block,                /* the block that is to be built */
                                           /* or have a delay calculated for*/
                    *inputPins,            /* a vecor of input pins.        */
                    *outputPins,           /* a vector of output pins.      */
                    *nodes,                /* a vector of node points that  */
                                           /* were added to the block when  */
                                           /* the model was built.          */
                    fromPoint,             /* is set prior to delay or slew */
                                           /* calculation by the calling    */
                                           /* subsystem                     */
                    toPoint;               /* is set prior to delay or slew */
                                           /* calculation                   */
        /*****************************************************************
        ** Counters.
        *****************************************************************/
        int       inputPinCount,           /* the number of input pins in   */
                                           /* the input pin list(input_pins)*/
                  outputPinCount,          /* the number of output pins on  */
                                           /* the output_pins list          */
                  nodeCount;               /* the number of nodes added by  */
                                           /* the calculator during the     */
                                           /* model building phase          */
        /*****************************************************************
        ** Edges and modes.
        *****************************************************************/
        DCM_EdgeTypes    sourceEdge;       /* (R, F, B) R=rise, F=fall and  */
                                           /* B=both rise and fall          */
        DCM_EdgeTypes    sinkEdge;         /* (R, F, B)                     */
        DCM_PropagationTypes sourceMode;   /*  timing segments mode (e, l,  */
                                           /*  b, x, xb) where              */
                                           /* e early, l late b both early  */
                                           /* and late,x is terminate and   */
                                           /* xb is both edges terminated   */
        DCM_PropagationTypes sinkMode;     /* sink mode same as sourcemode  */
        /*****************************************************************
        ** Calculation mode: best case, worst case, nominal.
        *****************************************************************/
        DCM_CalculationModes calcMode;
        /*****************************************************************
        ** Slew values.
        *****************************************************************/
        DCM_SLEW_REC slew;
        /*****************************************************************
        ** The following pointer is used to point to rule specific data
        ** for each segment modelled. If the distructor function pointer
        ** is null, the data applies to all circuits for which it is modelled
        ** and should not be deleted. If the function pointer is not null,
        ** then call it for each deleted segment.
        *****************************************************************/
        DCM_PathDataBlock   *pathData;
        /*****************************************************************
        ** The following pointer allows additions to the standard
        ** structure which the timer or other applications understand.  Changes
        ** here will not affect the compiler, as the macros to access the data
```

```
** can be stored in the .h file libraries required for each technology.
** DCM will not alter the data contained in this application block.
** So if the rule creates a new standard structure for its purposes
** it will merely copy the pointers value.  On the other hand, when
** the rule discards a standard structure it previously created it
** simply forgets the pointer.
*****************************************************************/
void  *applicationInfo;
/*****************************************************************
** The following pointer is used to point to rule specific data for
** each circuit modelled.  If the distructor function pointer is null
** the data applies to all circuits for which this is modelled for and
** should not be delete otherwise if the function pointer is not null
** then call it.
*****************************************************************/
DCM_CellDataBlock  *cellData;
/*****************************************************************
** reserved states.
*****************************************************************/
void  *ibm1;
void  *ibm2;
void  *ibm3;

#ifdef __cplusplus
/*****************************************************************
** Constructor.
*****************************************************************/
void *operator new(size_t s);
DCM_STD_STRUCT();
/*****************************************************************
** Destructor.
*****************************************************************/
void operator delete(void *item, size_t);
~DCM_STD_STRUCT();
/*****************************************************************
** Assignment.
*****************************************************************/
DCM_STD_STRUCT & operator =(DCM_STD_STRUCT &);
/*****************************************************************
** Technology mapping.
** The "generic" technology is the string "GENERIC".
** A 0 passed parameter is treated the same as the "generic"
** technology.setTechnology() returns a 0 on an error.
**
** The input string is NOT copied.  No storage management is performed
** by the method.  DO NOT FREE THE RESULT!
*****************************************************************/
const char *setTechnology(const char *);
                                  /* Set new mapping, return old one.*/
                                  /* NOTE: no strings are copied!    */
                                  /* only the pointer is remembered. */
/*****************************************************************
** The "generic" technology is the string "GENERIC".
** getTechnology() returns NULL on an error.
```

```
1       ** DO NOT FREE THE RESULT!
        ***************************************************************************/
        const char *getTechnology() const;
        /***************************************************************************
5       ** Return a 0-terminated pointer vector of technology names
        ** available in this run.
        **
        ** Both the vector and the strings exist in space. Caller must free.
        **
10      ** Returns 0 on error.
        ***************************************************************************/
        char **getAllTechs() const;
        char ***getAllTechsAsPinlist() const;

15      void freeAllTechs(char **) const;
        void freeAllTechsAsPinlist(char ***) const;
        /***************************************************************************
        ** Return nonzero if this object is mapped to the generic technology.
        ***************************************************************************/
20      int isGeneric() const;
        /***************************************************************************
        ** Return the given technology mapping in a nugget.
        ** Does NOT affect the current mapping of the standard structure.
        ***************************************************************************/
25      int mapNugget(const char *name, DCM_TechFamilyNugget *) const;
        /***************************************************************************
        ** Given a mapping in a nugget, convert the current standard structure
        ** to that technology mapping.
        ***************************************************************************/
30      int takeMappingOfNugget(const DCM_TechFamilyNugget *);
        /***************************************************************************
        ** User object registration and management.
        ***************************************************************************/
        int registerUserObject(const void *);
35      void deleteRegisteredUserObjects();
        void deleteOneUserObject(void *);

        /***************************************************************************
        ** Methods that deal with the hidden path information.
40      ** They depend on pathData being properly set.
        ***************************************************************************/

        /***************************************************************************
        ** Return nonzero if the path has something stored on it.
45      ***************************************************************************/
        int isSomethingStored() const;       /* zero means “no”              */
        /***************************************************************************
        ** Return non-zero if there is a STORE of a function with the
        ** given name.
50      **
        ** Note that this does not resolve possible name ambiguity.
        ***************************************************************************/
        int isThisFunctionStored(const char *name) const; /* Zero means “no”*/
        /***************************************************************************
```

```
    ** Return nonzero if this PATH is inconsistent.
    ** (Means instance-specific data exists.)
    ************************************************************************/
    int isPathInconsistent() const;            /* zero means “no”       */
    /************************************************************************
    ** Return nonzero if this model presents inconsistent STOREs anywhere.
    ** (Means instance-specific data exists.)
    ************************************************************************/
    int isModelDataInconsistent() const;       /* zero means “no”       */
    /************************************************************************
    ** Return nonzero if this model is possibly inconsistent.
    ** (Means do NOT attempt to reuse model, must call for model each
    ** time.)
    ************************************************************************/
    int isModelPossiblyInconsistent() const;   /* zero means “no”       */

#endif

};                                        /* End of STD_STRUCT.          */

#endif
```

8.19 Standard macros (dcmstd_macs.h) file

This section lists the dcm_std_macs.h file.

```
#ifndef _H_DCMSTD_MACROS
#define _H_DCMSTD_MACROS
/************************************************************************
** INCLUDE NAME..... dcm_std_macs.h
**
** PURPOSE..... ...
** These are the standard macros used by the NDCL language.
** They appear to be built-in functions from the rules coder’s viewpoint.
**
** NOTES.............
**
** ASSUMPTIONS.......
**
** RESTRICTIONS.....
**
** LIMITATIONS......
**
** DEVIATIONS.......
**
** AUTHOR(S)......... H. John Beatty, Peter C. Elmendorf
**
** CHANGES:
**
************************************************************************/
************************************************************************
** Constant tables for use by the built-in macros.
************************************************************************/
```

```
static const char * const dcmStdStructEdges[] = {
  "R",                                  /* Rising edge.                */
  "F",                                  /* Falling edge.               */
  "B",                                  /* Both edges.                 */
  "X",                                  /* Same edge.                  */
  "C",                                  /* Compliment edge.            */
  "T",                                  /* Terminate.                  */
  "Y",                                  /* Terminate both.             */
  "1Z",                                 /* OneToZ                      */
  "Z1",                                 /* ZtoOne                      */
  "0Z",                                 /* ZeroToZ                     */
  "Z0",                                 /* ZtoZero                     */
  "A"                                   /* All edges.                  */
  };

static const char * const dcmStdStructModes[] = {
  "E",                                  /* Early mode.                 */
  "L",                                  /* Late mode.                  */
  "B",                                  /* Both modes.                 */
  "X",                                  /* Same mode.                  */
  "C"                                   /* Compliment mode.            */
  };

static const char * const dcmCalcModes[] = {
  "B",                                  /* Best case.                  */
  "W",                                  /* Worst case.                 */
  "N"                                   /* Nominal case.               */
  };

/*****************************************************************************
** The built-in names for standard structure fields.
** The DCL Compiler may generate code that references these macros.
*****************************************************************************/
#define DCM__CELL               (std_struct->cell)
#define DCM__CELL_QUAL          (std_struct->cellQual)
#define DCM__MODEL_DOMAIN       (std_struct->modelDomain)
#define DCM__CLKFLG             (std_struct->pathData->pcdb->clkflg)
#define DCM__CKTTYPE            (std_struct->pathData->pcdb->ckttype)
#define DCM__CYCLEADJ           (std_struct->pathData->cycle_adj)
#define DCM__MODEL_NAME         (std_struct->model_name)
#define DCM__PATH               (std_struct->pathData->path)
#define DCM__PHASE              ((std_struct->sourceEdge==
                                  std_struct->sinkEdge) \ ? "I" : "O")
#define DCM__BLOCK              (std_struct->block)
#define DCM__INPUT_PINS         (std_struct->inputPins)
#define DCM__OUTPUT_PINS        (std_struct->outputPins)
#define DCM__NODES              (std_struct->nodes)
#define DCM__FROM_POINT         (std_struct->fromPoint)
#define DCM__TO_POINT           (std_struct->toPoint)

#define DCM__INPUT_PIN_COUNT  (std_struct->inputPinCount)
#define DCM__OUTPUT_PIN_COUNT (std_struct->outputPinCount)
#define DCM__NODE_COUNT       (std_struct->nodeCount)
#define DCM__EARLY_MODE      ((char *)dcmStdStructModes[std_struct->
```

```
                                  sourceMode])
#define DCM__LATE_MODE       ((char *)dcmStdStructModes[std_struct->
                                  sinkMode])
#define DCM__SOURCE_EDGE     ((char *)dcmStdStructEdges[std_struct->
                                  sourceEdge])
#define DCM__SINK_EDGE       ((char *)dcmStdStructEdges[std_struct->
                                  sinkEdge])
#define DCM__CALC_MODE       ((char *)dcmCalcModes[std_struct->
                                  calcMode])

#define DCM__EARLY_MODE_SCALAR  (std_struct->sourceMode)
#define DCM__LATE_MODE_SCALAR   (std_struct->sinkMode)
#define DCM__SOURCE_EDGE_SCALAR (std_struct->sourceEdge)
#define DCM__SINK_EDGE_SCALAR   (std_struct->sinkEdge)
#define DCM__CALC_MODE_SCALAR   (std_struct->calcMode)

#define DCM__DELAY_FUNC       (std_struct->pathData->pcdb->delay)
#define DCM__SLEW_FUNC        (std_struct->pathData->pcdb->slew)

#define DCM__EARLY            (dcm_rtn->early)
#define DCM__LATE             (dcm_rtn->late)

#define DCM__REFERENCE_POINT  DCM__FROM_POINT
#define DCM__REFERENCE_EDGE   DCM__SOURCE_EDGE
#define DCM__REFERENCE_EDGE_SCALAR   DCM__SOURCE_EDGE_SCALAR
#define DCM__TEST_MODE        DCM__EARLY_MODE

#define DCM__SIGNAL_POINT     DCM__TO_POINT
#define DCM__SIGNAL_MODE      DCM__SINK_MODE
#define DCM__SIGNAL_EDGE      DCM__SINK_EDGE
#define DCM__SIGNAL_EDGE_SCALAR      DCM__SINK_EDGE_SCALAR

#define DCM__PATH_DATA        (std_struct->pathData)
#define DCM__CELL_DATA        (std_struct->cellData)

/*****************************************************************
** Use these macros for accessing the different slew rates in the
** standard structure.
*****************************************************************/
#define DCM__LATE_SLEW              (std_struct->slew.late)
#define DCM__EARLY_SLEW             (std_struct->slew.early)

#define DCM__SIGNAL_SLEW            (std_struct->slew.late)
#define DCM__REFERENCE_SLEW         (std_struct->slew.early)

/*****************************************************************
** Use these macros to set the DEFAULT and specific RESULT fields
** of INTERNAL statements.
*****************************************************************/

#define DEFAULT_RESULT        DCM__DEFAULT_RESULT
#define RESULT(a)             DCM__RESULT(a)

#define DCM__DEFAULT_RESULT   (dcm_rtn->DEFAULT)
```

```
#define DCM__RESULT(a)        (dcm_rtn->a)

#endif                                  /*  _H_STD_MACROS                */
```

8.20 Standard interface structures (dcmintf.h) file

This section lists the `dcmintf.h` file.

```
#ifndef _H_DCMINTF
#define _H_DCMINTF
/*************************************************************************
** INCLUDE NAME..... dcmintf.h
**
** PURPOSE..........
** This include defines the interface structures used for communication
** between a DCM-compiled rule and the outside world.
**
** NOTES............
**
** ASSUMPTIONS.......
**
** RESTRICTIONS.....
**
** LIMITATIONS......
**
** DEVIATIONS.......
**
** AUTHOR(S)......... H. John Beatty, Peter C. Elmendorf
**
** CHANGES:
**
*************************************************************************/
#include <dcmpltfm.h>

/*************************************************************************
** This flag tells the library the vintage of the interface on the
** application end.
*************************************************************************/
#define DCM_BIND_RULE_VERSION_FLAGS 1

/*************************************************************************
** Now for our standard interface structure.
*************************************************************************/
#include <std_stru.h>
#include <dcmgarray.h>
#include <dcmgstruct.h>

/*************************************************************************
** These are the names of the environments which set directories in
** which to look for tables and rules.
*************************************************************************/

static const char DCMTableEnv[] = “DCMTABLEPATH”;
```

```
static const char DCMRuleEnv[]  = “DCMRULEPATH”;
static const char DCMErrEnv[]   = “DCMSTDERR”;

/*********************************************************************
** These are the names of optional functions which the DCM rule can call
** at initialization and termination.
*********************************************************************/
static const char DCMBeginRule[] = “DCM_BEGIN”;
static const char DCMEndRule[]   = “DCM_END”;

/*********************************************************************
** Typedef for the model procs.
*********************************************************************/
typedef int (*DCM_ModelProcType)(DCM_STD_STRUCT *);

/*********************************************************************
** Typedef for data passed to the DCM startup and termination functions.
*********************************************************************/
typedef struct DCMBeginAndEndData {
  int dcmRV;                    /* Runtime version #.                 */
  int dcmRL;                    /* Runtime level #.                   */
  const char *techFamily;       /* Tech family name.                  */
  const char *ruleLoadName;     /* Name of rule, as loaded.           */
  const char *ruleTimeStamp;    /* string timestamp, rule compilation.*/
} DCMBeginAndEndData;

/*********************************************************************
** Typedef for the DCM startup and termination functions.
*********************************************************************/
typedef int (*DCMBeginAndEndFunction)(DCMBeginAndEndData *packet);

/*********************************************************************
** Structure which associates string names to their functions.
**
** An array of these is passed to the DCM from the caller.  This allows
** the DCM to map external function names.
**
** An array of these is passed from the DCM to the caller.  This allows
** the caller to map DCM internal function names.
**
** The very last entry in these arrays has two NULL pointers therein.
*********************************************************************/
struct DCM_FunctionTable {
  char      *name;              /* -> string name of function.  */
  DCM_GeneralFunction function;  /* -> actual function.          */
};
/*********************************************************************
** This structure is passed from DCM to the external world.
*********************************************************************/
typedef struct DCMTransmittedInfo {
  DCM_ModelProcType       modelSearch;    /* -> model search function.*/
  DCM_DelayFunctionType   delayFunction;  /* -> delay function.       */
  DCM_SlewFunctionType    slewFunction;   /* -> slew function.        */
  DCM_CheckFunctionType   checkFunction;  /* -> check function.       */
```

```
    const DCM_FunctionTable *inits;   /* -> table of expose statements. */
    /*****************************************************************
    ** The following items are valid ONLY when dcmLoadRule() was called
    ** with a value for the flags of DCM_LOAD_RULE_FLAGS.
    **
    ** Older versions required a zero as the parameter.
    *****************************************************************/
    DCM_MethodMapFunctionType  methodMapper;     /* Method mapper.      */
    DCM_MethodCallFunctionType pathMethodCaller; /* PATH::Method caller.*/
    DCM_MethodCallFunctionType cellMethodCaller; /* CELL::Method caller.*/
    /*****************************************************************
    ** As further function/standards are defined, these reserved
    ** fields may be used to help implement them.
    **
    ** These fields are NOT initialized by the DCM unless that DCM
    ** is loaded with a VALID nonzero flag parameter to dcmLoadRule().
    *****************************************************************/
    DCM_GeneralFunction   reserved2;       /* Reserved for future use.  */
    DCM_GeneralFunction   reserved3;       /* Reserved for future use.  */
    DCM_GeneralFunction   reserved4;       /* Reserved for future use.  */
    DCM_GeneralFunction   reserved5;       /* Reserved for future use.  */
    DCM_GeneralFunction   reserved6;       /* Reserved for future use.  */
    DCM_GeneralFunction   reserved7;       /* Reserved for future use.  */
    DCM_GeneralFunction   reserved8;       /* Reserved for future use.  */
    DCM_GeneralFunction   reserved9;       /* Reserved for future use.  */
    DCM_GeneralFunction   reserved10;      /* Reserved for future use.  */
    DCM_GeneralFunction   reserved11;      /* Reserved for future use.  */
    DCM_GeneralFunction   reserved12;      /* Reserved for future use.  */
    DCM_GeneralFunction   reserved13;      /* Reserved for future use.  */
    DCM_GeneralFunction   reserved14;      /* Reserved for future use.  */
    DCM_GeneralFunction   reserved15;      /* Reserved for future use.  */
    DCM_GeneralFunction   reserved16;      /* Reserved for future use.  */
    DCM_GeneralFunction   reserved17;      /* Reserved for future use.  */
    DCM_GeneralFunction   reserved18;      /* Reserved for future use.  */
    DCM_GeneralFunction   reserved19;      /* Reserved for future use.  */
    DCM_GeneralFunction   reserved20;      /* Reserved for future use.  */
} DCMTransmittedInfo;

/*****************************************************************
** This function looks for a given function name in the passed
** table of functions, and returns the function pointer associated with
** the passed string name.
**
**
** Returns 0 if:
**    Name not found.
**    Either parameter 0.
*****************************************************************/
DCM_XC DCM_GeneralFunction dcmFindFunction(
     const char * functionName,      /* I  -> string name of function.*/
     const DCM_FunctionTable * table  /* I  -> function table to use. */
     );

/*****************************************************************
```

```
** This is a version of dcmFindFunction which issues no messages.
*****************************************************************/
DCM_XC DCM_GeneralFunction dcmQuietFindFunction(
     const char * functionName,      /* I  -> string name of function.*/
     const DCM_FunctionTable * table   /* I  -> function table to use.*/
     );

/*****************************************************************
** This function merges N FunctionTables into a single table.
**
** Pass: a NULL terminated vector of pointers to FunctionTable pointers.
** Rtns: -> the merged table, or NULL on error.
**
** ASSUMPTIONS....... Result allocated by us must be freed by the caller.
*****************************************************************/
DCM_XC DCM_FunctionTable *dcmMergeFunctionTable(DCM_FunctionTable **in);

/*****************************************************************
** DO NOT MODIFY THE CONTENTS!
*****************************************************************/
typedef struct DCM_ModelProcNugget {
  DCM_ModelProcType  proc;
  void         **cb;
} DCM_ModelProcNugget;

/*****************************************************************
** This function calls a modelproc from its function nugget.
** Zero return indicates success.
*****************************************************************/
DCM_XC int dcmCallModelProcNugget(DCM_STD_STRUCT *,
                                  DCM_ModelProcNugget *);

/*****************************************************************
** Timestamp services.
*****************************************************************/
typedef struct DCM_TableTimeStampInfo {
  const char *dcmTimeStamp;           /* timestamp.                */
  const char *name;                   /* table name.               */
} DCM_TableTimeStampInfo;

typedef struct DCM_TimeStampInfo {
  const char *dcmTimeStamp;           /* timestamp.                */
  const char *dcmTechFamily;          /* tech family (%TECHFAMILY)   */
  const char *localName;              /* local name  (%RULENAME)     */
  const char *loadPath;               /* -> path to exact rule file. */
  DCM_TableTimeStampInfo **tables;
} DCM_TimeStampInfo;

/*****************************************************************
** Get a pointer to a vector of pointers to DCM_TimeStampInfo objects,
** one per rule in the tech family.
*****************************************************************/

DCM_XC DCM_TimeStampInfo **dcmGetTechFamilyTS(const char *name);
```

```
/*****************************************************************
** Function to delete the storage returned by dcmGetTechFamilyTS.
*****************************************************************/
DCM_XC void dcmDelTechFamilyTS(DCM_TimeStampInfo **);

/*****************************************************************
** Allows application to restitch an EXTERNAL statement.
*****************************************************************/
typedef enum DCM_RestitchExternal_Status {
  DCM_RestitchExternal_OK           = 0,
  DCM_RestitchExternal_NoRuleLoaded = 1,
  DCM_RestitchExternal_NoSuchName   = 2,
  DCM_RestitchExternal_BadParms     = 3,
  DCM_RestitchExternal_OtherError   = 4
} DCM_RestitchExternal_Status;

typedef void *DCM_RestitchExternalRestore;

DCM_XC DCM_RestitchExternal_Status dcmRestitchExternal
  (const DCM_STD_STRUCT *std,          /* I: any valid std struct.    */
   const char         *name,           /* I: name of function.        */
   DCM_GeneralFunction func,           /* I: new function ptr to use. */
   DCM_RestitchExternalRestore *restore);/*O:restoration information. */

DCM_XC DCM_RestitchExternal_Status dcmRestoreExternal
  (const DCM_STD_STRUCT *std,          /* I: any valid std struct.   */
   DCM_RestitchExternalRestore restore);/* I: restoration information.*/

#endif                                 /* _H_DCMINTF                 */
```

8.21 Standard loading (dcmload.h) file

This section lists the `dcmload.h` file.

```
#ifndef _H_DCMLOAD
#define _H_DCMLOAD
/*****************************************************************
** INCLUDE NAME..... dcmload.h
**
** PURPOSE.......... Declare the means by which rules are loaded.
**
** NOTES............ Only dynamic load is supported.
**
** ASSUMPTIONS.......
**
** RESTRICTIONS..... S/6000, SUN, HP.
**
** LIMITATIONS......
**
** DEVIATIONS.......
**
** AUTHOR(S)......... Peter C. Elmendorf
** CHANGES:
```

```
**
*****************************************************************/
#include <dcmintf.h>

/*****************************************************************
** Dynamic load control.
** Scaffolding defines dcmLoadRule() as a macro.
*****************************************************************/
#ifdef DCM_SCAFFOLD
#  include <dcmscaff.h>               /* For testing.              */
/*****************************************************************
** Some platforms have dynamic load support.
*****************************************************************/
#else
#  ifdef DCM_DYNAMIC_LOAD_AVAILABLE
/*****************************************************************
** This function will load a rule, taking the rule path variable
** DCMRULEPATH into account.
*****************************************************************/
DCM_XC DCM_GeneralFunction dcmLoadRule(
     const char *ruleName,        /* I -> the name of the rule module.*/
     unsigned int  flags          /* I flags.                         */
          );

DCM_XC DCM_GeneralFunction dcmAddRule(
   const char *ruleName,          /* I -> the name of the rule module.*/
   int *RC                        /* O: dcm return code               */
   );

/*****************************************************************
** Some platforms do not have dynamic load support.
** These are not supported.
*****************************************************************/
#  else                                /* Else no dynamic load.       */
#     include <dcmscaff.h>
#  endif                               /* End no dynamic load.        */
#endif                                 /* End dynamic load.           */

/*****************************************************************
** This macro defines the standard routine that loads the root DCM.
** We supply the version flags so dcmLoadRule() can tell what kind
** of version of DCL was used to compile the application.
*****************************************************************/
#define dcmBindRule(_name_)
        dcmLoadRule(_name_,DCM_BIND_RULE_VERSION_FLAGS)

/*****************************************************************
** This function will unload a rule loaded by dcmLoadRule().
*****************************************************************/
DCM_XC int dcmUnloadRule(
    DCM_GeneralFunction rule          /* I Pointer from dcmLoadRule().*/
    );
#endif                                 /* _H_DCMLOAD                  */
```

8.22 Standard debug (dcmdebug.h) file

This section lists the `dcmdebug.h` file.

```
#ifndef _H_DCMDEBUG
#define _H_DCMDEBUG
/*****************************************************************************
** INCLUDE NAME..... dcmdebug.h
**
** PURPOSE.........
** This include defines all the items and functions needed for
** generated C code to utilize the DCM Debugging Functions.
**
** NOTES.............
**
** When the rule is compiled in debug mode, the necessary includes and
** declarations are available so that all debug facilities are
** accessible.
**
** When the rule is compiled in non-debug mode, the debug variables
** are bypassed.  And, instead of declarations for the debugging
** functions called by the DCM compiler, macros are included instead.
** These macros look just like the debug calls generated by DCM, but
** the expand into nothing.  Poof! the debug calls vanish.
**
** (Note that the debug TYPES are always included regardless of mode.)
**
** This eases code generation in the compiler.  The compiler generated
** code is independent of debug level used in the rule.  All debug code
** generation or elimination is handled when the rule is itself compiled.
**
** Use of this approach also improves readability of the DCM-generated
** code (and thus its debuggability) because the generated C code is
** not cluttered with #ifdef preprocessor statements.
**
** ASSUMPTIONS.......
** When not in debug or internals mode, define the debugging function
** calls to be null macros.
**
** This permits the calls to these functions to remain in the code
** because cpp will replace the function calls with whitespace.
**
** This is a very easy and reliable way of writing code or rules which
** always contain their debugging sections.  Use of numerous annoying
** ifdef's and endif's is avoided.  Code is cleaner, reads better, is
** easier to work with.
**
** RESTRICTIONS.....
**
** LIMITATIONS......
**
** DEVIATIONS.......
**
** AUTHOR(S)......... Peter C. Elmendorf
```

```
**
** CHANGES:
**
*****************************************************************************/

/*****************************************************************************
** These are the debugging switches.
*****************************************************************************/
typedef enum DCMDebugSettings {
  DCM_OFF,                              /* Debug turned off.           */
  DCM_LOW,                              /* Basic debug info.           */
  DCM_MEDIUM,                           /* More detail.                */
  DCM_HIGH,                             /* Lots of detail.             */
  DCM_FULLBORE                          /* You better be serious.      */
} DCMDebugSettings;

/*****************************************************************************
** Typedef for structure dumpers.
*****************************************************************************/
typedef void (*DCMStructureDumper)(const DCM_VOID);

#ifdef DCL_DEBUG
/*****************************************************************************
** These control array and structure initialization for DEBUG COMPILE.
*****************************************************************************/
#define DCM_AINIT_DEBUG_DEPENDENT DCM_AINIT_debugInits
#define DCM_SINIT_DEBUG_DEPENDENT DCM_SINIT_debugInits

#else
/*****************************************************************************
** These control array and structure initialization for NORMAL COMPILE.
*****************************************************************************/
#define DCM_AINIT_DEBUG_DEPENDENT DCM_AINIT_compilerInits
#define DCM_SINIT_DEBUG_DEPENDENT DCM_SINIT_compilerInits
#endif

/*****************************************************************************
** Generate this code in DEBUG mode OR when compiling the runtime
** support library and compiler.
*****************************************************************************/
#ifdef DCM_GUTS_OR_DEBUG

#define DCM_DEBUG_CONDX_PTR(x) (x)

/*****************************************************************************
** Returns zero if structure knows it is writable (not writable currently
** means “belongs to TABLEDEF.”  This “writable” is a runtime construct,
** NOT part of the VAR semantics of the language.
*****************************************************************************/
DCM_XC int  dcmCheckWritableStruct(const DCM_STRUCT *,
            const char *vn,
            const char *fn,
            const DCM_RuleScope *rs);
```

```
DCM_XC int  dcmCheckWritableArray(const DCM_ARRAY *,
           const char *vn,
           const char *fn,
           const DCM_RuleScope *rs);

/*****************************************************************
** This function is called on entry to an NDCL statement in debug mode.
*****************************************************************/
DCM_XC void dcm_debugEntry(
       const DCM_RuleScope *scope,     /* I  -> current rule  scope.  */
       const DCM_STD_STRUCT *std,      /* I  -> std struct.           */
       const char *stmtType,           /* I  -> name of stmt type.    */
       const char *name);              /* I  -> statement name.       */

/*****************************************************************
** This function is called on exit from an NDCL statement in debug mode.
*****************************************************************/
DCM_XC void dcm_debugExit(
       const DCM_RuleScope *scope,     /* I  -> current rule  scope.  */
       const DCM_STD_STRUCT *std,      /* I  -> std struct.           */
       const char *stmtType,           /* I  -> name of stmt type.    */
       const char *name,               /* I  -> statement name.       */
             int   rc);                /* I  stmt return code.        */

/*****************************************************************
** This function is called on exit from an NDCL statement in debug mode.
*****************************************************************/
DCM_XC void dcm_debugExitWithDump(
       const DCM_RuleScope *scope,     /* I -> current rule  scope.   */
       const DCM_STD_STRUCT *std,      /* I -> std struct.            */
       const char    *stmtType,        /* I -> name of stmt type.     */
       const char    *name,            /* I -> statement name.        */
             int      rc,              /* I stmt return code.         */
             DCMStructureDumper dumper,/* I printing function to call.*/
             DCM_VOID value);          /* I -> result structure.      */

/*****************************************************************
** This function is called on exit from an NDCL statement in debug mode.
*****************************************************************/
DCM_XC void dcm_debugExitBad(
       const DCM_RuleScope *scope,     /* I  -> current rule  scope.  */
       const DCM_STD_STRUCT *std,      /* I  -> std struct.           */
       const char *stmtType,           /* I  -> name of stmt type.    */
       const char *name,               /* I  -> statement name.       */
             int   rc);                /* I  stmt return code.        */

/*****************************************************************
** This function is called on entry to an NDCL statement in debug mode.
*****************************************************************/
DCM_XC void dcm_debugEntryNoStd(
       const DCM_RuleScope *scope,     /* I  -> current rule  scope.  */
       const char *stmtType,           /* I  -> name of stmt type.    */
       const char *name);              /* I  -> statement name.       */
```

```
/*****************************************************************
** This function is called on exit from an NDCL statement in debug mode.
*****************************************************************/
DCM_XC void dcm_debugExitNoStd(
       const DCM_RuleScope *scope,     /* I  -> current rule  scope.  */
       const char *stmtType,           /* I  -> name of stmt type.    */
       const char *name,               /* I  -> statement name.       */
             int   rc);                /* I  stmt return code.        */

/*****************************************************************
** This function is called on exit from an NDCL statement in debug mode.
*****************************************************************/
DCM_XC void dcm_debugExitWithDumpNoStd(
       const DCM_RuleScope *scope,     /* I -> current rule  scope.   */
       const char    *stmtType,        /* I -> name of stmt type.     */
       const char    *name,            /* I -> statement name.        */
             int      rc,              /* I stmt return code.         */
             DCMStructureDumper dumper,/* I printing function to call.*/
             DCM_VOID value);          /* I -> result structure.      */

/*****************************************************************
** This function is called on exit from an NDCL statement in debug mode.
*****************************************************************/
DCM_XC void dcm_debugExitBadNoStd(
       const DCM_RuleScope *scope,     /* I  -> current rule  scope.  */
       const char *stmtType,           /* I  -> name of stmt type.    */
       const char *name,               /* I  -> statement name.       */
             int   rc);                /* I  stmt return code.        */

/*****************************************************************
** This function blabs about a default clause being taken.
*****************************************************************/
DCM_XC void dcm_debugDefault(
       const DCM_RuleScope *scope,     /* I  -> current rule  scope.  */
             int   rc);                /* I  stmt return code.        */

/*****************************************************************
** This function blabs about a default clause being prohibited.
*****************************************************************/
DCM_XC void dcm_debugDefaultCantFire(
       const DCM_RuleScope *scope,     /* I  -> current rule  scope.  */
             int   rc);                /* I  stmt return code.        */

/*****************************************************************
** This function dumps the model proc trees.
*****************************************************************/
DCM_XC void dcm_debugModelProcInit(
       const DCM_RuleScope *scope,     /* I  -> current rule  scope.  */
       const DCM_tree      *nodesP,    /* I  -> tree anchor.          */
       const DCM_tree      *anyinP,    /* I  -> tree anchor.          */
       const DCM_tree      *anyoutP);  /* I  -> tree anchor.          */
*****************************************************************
** This function dumps the simple model proc data.
*****************************************************************/
```

```
DCM_XC void dcm_debugModelProcSimple(
       const DCM_RuleScope *scope,     /* I -> current rule  scope.   */
       const DCM_STD_STRUCT *std);     /* I -> std structure.         */

/*****************************************************************
** This function dumps From and To pin lists.
*****************************************************************/
DCM_XC void dcm_debugPathPins(
       const DCM_RuleScope *scope,   /* I -> current rule  scope.     */
       const char          *fromOrTo,/* I -> From/To string (for msg).*/
       const DCM_list      *list     /* I -> list to dump.            */
           );

/*****************************************************************
** This function dumps simple name lists.
*****************************************************************/
DCM_XC void dcm_debugNameList(
       const DCM_RuleScope *scope,   /* I -> current rule  scope.     */
       const char          *fromOrTo,/* I -> From/To string (for msg).*/
       const DCM_list      *list     /* I -> list to dump.            */
           );

/*****************************************************************
** This function dumps a single model proc pin tree.
*****************************************************************/
DCM_XC void dcm_dumpMPTree(
       const DCM_RuleScope *scope,     /* I  -> current rule  scope.  */
       const DCM_tree      *tree,      /* I  -> tree anchor.          */
       const char          *name       /* I  -> name of tree.         */
             );

/*****************************************************************
** This function dumps the names of the pins in a PINLIST vector.
*****************************************************************/
DCM_XC void dcm_dumpPinList(
       const DCM_RuleScope *scope,     /* I  -> current rule  scope.  */
       const DCM_HANDLE    *list       /* I  Pin List pointer.        */
           );

/*****************************************************************
** This function dumps the standard structure.
*****************************************************************/
DCM_XC void dcm_dump_STD_STRUCT(
       const DCM_RuleScope *scope,    /* I -> current rule  scope.    */
       const DCM_STD_STRUCT *std);    /* I -> the std struct to dump. */

/*****************************************************************
** Generate messages each time we try to stitch a function pointer.
*****************************************************************/
DCM_XC int dcm_debugStitchX(const DCM_RuleScope *scope,
                            const char          *name);

DCM_XC int dcm_debugStitchI(const DCM_RuleScope *scope,
                            const char          *name);
```

```
DCM_XC int dcm_debugStitchS(const DCM_RuleScope *scope,
                            const char          *name);

/*********************************************************************
** Generate a message concerning table loading.
** Always generated, since tables are loaded before any INIT statments
** can be executed.
*********************************************************************/
DCM_XC void dcm_debugTableLoading
            (const DCM_RuleScope       *rs,
             const char                *name,
             const DCMTableDescriptor  *table);

DCM_XC void dcm_debugTableDeferred
            (const DCM_RuleScope       *rs,
             const char                *name,
             const DCMTableDescriptor  *table);

/*********************************************************************
** Generate a message concerning dynamic table row addition.
*********************************************************************/
DCM_XC void dcm_debugTableAdd(const char **, void *, const char *,
                DCMStructureDumper, int, const DCM_RuleScope *);

/*********************************************************************
** Generate a message concerning dynamic table row deletion.
*********************************************************************/
DCM_XC void dcm_debugTableDel(const char **, const char *,
                              const DCM_RuleScope *);

/*********************************************************************
** This function will tell what model we are searching for.
*********************************************************************/
DCM_XC void dcm_debugModelSearch(
       const DCM_RuleScope  *scope,    /* I -> current rule  scope.   */
       const DCM_STD_STRUCT *std,      /* I -> standard struct.       */
       const char           *btr,      /* I -> BTR to search for.     */
       const char           *bhc,      /* I -> BHC to search for.     */
       const char           *brc);     /* I -> BRC to search for.     */

/*********************************************************************
** Initialize rule message.
*********************************************************************/
DCM_XC void dcm_debugInit(const DCM_RuleScope *);

/*********************************************************************
** Terminate rule message.
*********************************************************************/
DCM_XC void dcm_debugTerm(const DCM_RuleScope *);

/*********************************************************************
** Subrule loading.
*********************************************************************/
```

```
DCM_XC void dcm_debugSubruleLoad(const DCM_RuleScope *scope,
                                 const char          *subrule);
DCM_XC void dcm_debugSubrulePreLoad(const DCM_RuleScope *scope,
                                    const char          *subrule);

/*****************************************************************
** Segment creation trace messages.
*****************************************************************/
DCM_XC void dcm_debugSegData(const DCM_RuleScope *scope,
                             const DCM_PCDB      *pcdb);

DCM_XC void dcm_debugNodeCall(const DCM_RuleScope *scope,
                              const char          *name);

DCM_XC void dcm_debugStoreCall(const DCM_RuleScope *scope,
                               const char          *name);

DCM_XC void dcm_debugSlotStoreCall(const DCM_RuleScope *scope,
                                   const char          *name);

DCM_XC void dcm_debugDelayMatrix(const DCM_RuleScope *scope,
                                 const char *,const char *,
                                 const char *,const char *,
                                 const char *,const char *);

DCM_XC void dcm_debugMethodsClause(const DCM_RuleScope *scope,
                                   const char *,const char *);

DCM_XC void dcm_debugTestMatrix(const DCM_RuleScope *scope,
                                const char *,const char *,
                                const char *,const char *,
                                const char *, int, int, const char *);

DCM_XC void dcm_debugPathSegment(const DCM_RuleScope *scope,
                                 const DCM_STD_STRUCT *,
                                 char);

DCM_XC void dcm_debugTestSegment(const DCM_RuleScope *scope,
                                 const DCM_STD_STRUCT *,
                                 char);

DCM_XC void dcm_debugNetSegment(const DCM_RuleScope *scope,
                                const DCM_STD_STRUCT *,
                                const DCM_HANDLE,
                                const DCM_HANDLE,
                                char);

/*****************************************************************
** Printproc components.
*****************************************************************/
DCM_XC void dcmPrintProcNULL(const DCM_RuleScope *scope,
                             const char          *name);
```

```
DCM_XC void dcmPrintProcDouble(const DCM_RuleScope *scope,
                               const char          *name,
                               DCM_DOUBLE          value);

DCM_XC void dcmPrintProcFloat(const DCM_RuleScope *scope,
                              const char          *name,
                              DCM_DOUBLE          value);

DCM_XC void dcmPrintProcComplex(const DCM_RuleScope *scope,
                                const char          *name,
                                const DCM_COMPLEX   *value);

DCM_XC void dcmPrintProcSlew(const DCM_RuleScope *scope,
                             const char          *name,
                             const DCM_SLEW_REC  *value);

DCM_XC void dcmPrintProcInt(const DCM_RuleScope *scope,
                            const char          *name,
                            DCM_INTEGER         value);

DCM_XC void dcmPrintProcString(const DCM_RuleScope *scope,
                               const char          *name,
                               const char          *value);

DCM_XC void dcmPrintProcPin(const DCM_RuleScope *scope,
                            const char          *name,
                            const DCM_HANDLE    value);

DCM_XC void dcmPrintProcPinList(const DCM_RuleScope *scope,
                                const char          *name,
                                const DCM_HANDLE    *value);

DCM_XC void dcmPrintProcFunc(const DCM_RuleScope *scope,
                             const char          *name,
                             DCM_GeneralFunction value);

DCM_XC void dcmPrintProcVoid(const DCM_RuleScope *rs,
                                                 /* -> rule scope.    */
                             const char *name,   /* -> variable name. */
                             const DCM_VOID value);/* -> void value.  */

DCM_XC void dcmPrintProcErr(const DCM_RuleScope *scope,
                            const char *name);

/*****************************************************************
** FORCE protector.
*****************************************************************/
#define dcm_checkWritableArray(std,arr,vn,fn) \
  dcmCheckWritableArray((arr),(vn),(fn),DCM_RULE_ANCHOR())

#define dcm_checkWritableStruct(std,str,vn,fn) \
  dcmCheckWritableStruct((str),(vn),(fn),DCM_RULE_ANCHOR())
```

```
/*****************************************************************
** These macros save and restore the modelproc indentation settings.
** In case of model proc error, the indentation is reset correctly.
*****************************************************************/
DCM_XC int dcm_debugModelIndentSaver(const DCM_RuleScope *);
#define dcmModelIndentSaver() \
  int dcmIndentSave = dcm_debugModelIndentSaver(DCM_RULE_ANCHOR())

DCM_XC void dcm_debugModelIndentRestorer(const DCM_RuleScope *, int);
#define dcmModelIndentRestorer() \
  dcm_debugModelIndentRestorer(DCM_RULE_ANCHOR(),dcmIndentSave)

/*****************************************************************
** This macro delineates the end of the model search.
*****************************************************************/
DCM_XC int dcmEndModelProc(const DCM_RuleScope *scope);
#define dcm_debugModelSearchEnd() dcmEndModelProc(DCM_RULE_ANCHOR());

/*****************************************************************
** This macro sets the debug level, but ONLY if DCM compiled -debug.
*****************************************************************/
DCM_XC DCMDebugSettings dcm_debugSetDebugLevel(const DCM_RuleScope *,
                                               int);
#define SET_DCM_DEBUG_LEVEL(dcmLevel) \
  dcm_debugSetDebugLevel(DCM_RULE_ANCHOR(),dcmLevel)

#define SET_DCM_DEBUG_LEVEL_VOID(dcmLevel) \
  ((void) dcm_debugSetDebugLevel(DCM_RULE_ANCHOR(),dcmLevel))

/*****************************************************************
** This macro queries the debug level, but ONLY if DCM compiled -debug.
** The compiler generates calls to this macro.
*****************************************************************/
DCM_XC DCMDebugSettings dcm_debugGetDebugLevel(const DCM_RuleScope *);
#define GET_DCM_DEBUG_LEVEL() dcm_debugGetDebugLevel(DCM_RULE_ANCHOR())
#define DCM_STRUCT_DUMPER(a) ((DCMStructureDumper)(a))

/*****************************************************************
** Debug Functions for discrete math loops.
*****************************************************************/
DCM_XC void dcm_debugLoopStart(
       const DCM_RuleScope *scope,     /* I -> current rule  scope.   */
       const char          *name);     /* I loop var name.            */

DCM_XC void dcm_debugLoopEnd(
       const DCM_RuleScope *scope,     /* I -> current rule  scope.   */
       const char          *name);     /* I loop var name.            */

DCM_XC void dcm_debugIntLoop(
       const DCM_RuleScope *scope,     /* I -> current rule  scope.   */
       const char          *name,      /* I -> loop var name.         */
             int            init,      /* I initial loop value.       */
             int            term,      /* I termination value.        */
             int            inc);      /* I incrememt value.          */
```

```
DCM_XC void dcm_debugPinLoop(
       const DCM_RuleScope *scope,     /* I -> current rule  scope.   */
       const char          *name,      /* I -> loop var name.         */
       const DCM_HANDLE    *pl);       /* I the PINLIST.              */

DCM_XC void dcm_debugLoopResult(
       const DCM_RuleScope *scope,     /* I -> current rule  scope.   */
             DCM_DOUBLE     d);        /* I loop result.              */

/*****************************************************************
** Debug functions for chained EXPOSEs.
*****************************************************************/
DCM_XC void dcm_debugExposeChainForward(
       const DCM_RuleScope *scope,     /* I  -> current rule  scope.  */
       const char *name,               /* I  -> statement name.       */
             int   rc);                /* I  stmt return code.        */

DCM_XC void dcm_debugExposeChainStops(
       const DCM_RuleScope *scope,     /* I  -> current rule  scope.  */
       const char *name,               /* I  -> statement name.       */
             int   rc);                /* I  stmt return code.        */

DCM_XC int dcm_debugRevealVariable
  (const DCM_RuleScope *scope,         /* I -> current rule  scope.   */
   const char *string);

DCM_XC int dcm_debugGateForLevel(const DCM_RuleScope *scope, int level);

#define dcmVariablePrint(str,call) \
  (dcm_debugRevealVariable(DCM_RULE_ANCHOR(),(str)) ? ((call),1) : 0)

/*****************************************************************
** Define the disappearing macros for non-debug mode.
*****************************************************************/
#else

#define DCM_DEBUG_CONDX_PTR(x) NULL

#ifdef dcm_debugEntry
#undef dcm_debugEntry
#endif
#define dcm_debugEntry(dcmScope,dcmStd,dcmType,dcmName)

#ifdef dcm_debugExit
#undef dcm_debugExit
#endif
#define dcm_debugExit(dcmScope,dcmStd,dcmType,dcmName,dcmrc)

#ifdef dcm_debugExitWithDump
#undef dcm_debugExitWithDump
#endif
#define dcm_debugExitWithDump(dcmScope,dcmStd,dcmType,cdNname,dcmrc,\
                              dcmDumper,dcmValue)
```

```
#ifdef dcm_debugExitBad
#undef dcm_debugExitBad
#endif
#define dcm_debugExitBad(dcmScope,dcmStd,dcmType,dcmName,dcmrc)

#ifdef dcm_debugEntryNoStd
#undef dcm_debugEntryNoStd
#endif
#define dcm_debugEntryNoStd(dcmScope,dcmType,dcmName)

#ifdef dcm_debugExitNoStd
#undef dcm_debugExitNoStd
#endif
#define dcm_debugExitNoStd(dcmScope,dcmType,dcmName,dcmrc)

#ifdef dcm_debugExitWithDumpNoStd
#undef dcm_debugExitWithDumpNoStd
#endif
#define dcm_debugExitWithDumpNoStd(dcmScope,dcmType,cdNname,dcmrc,\
                                   dcmDumper,dcmValue)

#ifdef dcm_debugExitBadNoStd
#undef dcm_debugExitBadNoStd
#endif
#define dcm_debugExitBadNoStd(dcmScope,dcmType,dcmName,dcmrc)

#ifdef dcm_debugDefault
#undef dcm_debugDefault
#endif
#define dcm_debugDefault(dcmScope,dcmrc)

#ifdef dcm_debugDefaultCantFire
#undef dcm_debugDefaultCantFire
#endif
#define dcm_debugDefaultCantFire(dcmScope,dcmrc)

#ifdef dcm_debugModelProcInit
#undef dcm_debugModelProcInit
#endif
#define dcm_debugModelProcInit(dcmScope,dcmNodes,dcmAnyin,dcmAnyout)

#ifdef dcm_debugModelProcSimple
#undef dcm_debugModelProcSimple
#endif
#define dcm_debugModelProcSimple(dcmScope,dcmStd)

#ifdef dcm_debugStitchX
#undef dcm_debugStitchX
#endif
#define dcm_debugStitchX(dcmScope,dcmName) (0)

#ifdef dcm_debugStitchI
#undef dcm_debugStitchI
#endif
```

```
#define dcm_debugStitchI(dcmScope,dcmName) (0)

#ifdef dcm_debugStitchS
#undef dcm_debugStitchS
#endif
#define dcm_debugStitchS(dcmScope,dcmName) (0)

#ifdef dcm_debugTableLoading
#undef dcm_debugTableLoading
#endif
#define dcm_debugTableLoading(dcmScope,dcmName,dcmTable)

#ifdef dcm_debugTableDeferred
#undef dcm_debugTableDeferred
#endif
#define dcm_debugTableDeferred(dcmScope,dcmName,dcmTable)

#ifdef dcm_debugTableAdd
#undef dcm_debugTableAdd
#endif
#define dcm_debugTableAdd(dcmP1,dcmP2,dcmP3,dcmSD,dcmI1,dcmScope)

#ifdef dcm_debugTableDel
#undef dcm_debugTableDel
#endif
#define dcm_debugTableDel(dcmP1,dcmP2,dcmScope)

#ifdef dcm_debugModelSearch
#undef dcm_debugModelSearch
#endif
#define dcm_debugModelSearch(dcmScope,dcmStd,dcmbtr,dcmbhc,dcmbrc)

#ifdef dcm_debugModelSearchEnd
#undef dcm_debugModelSearchEnd
#endif
#define dcm_debugModelSearchEnd()

#ifdef dcmModelIndentSaver
#undef dcmModelIndentSaver
#endif
#define dcmModelIndentSaver()

#ifdef dcmModelIndentRestorer
#undef dcmModelIndentRestorer
#endif
#define dcmModelIndentRestorer()

#ifdef dcm_debugPathPins
#undef dcm_debugPathPins
#endif
#define dcm_debugPathPins(dcmScope,dcmKind,dcmList)
```

```
#ifdef dcm_debugNameList
#undef dcm_debugNameList
#endif
#define dcm_debugNameList(dcmScope,dcmKind,dcmList)

#ifdef dcm_dumpMPTree
#undef dcm_dumpMPTree
#endif
#define dcm_dumpMPTree(dcmScope, dcmTree, dcmString)

#ifdef dcm_dumpPinList
#undef dcm_dumpPinList
#endif
#define dcm_dumpPinList(dcmScope, dcmList)

#ifdef dcm_dump_STD_STRUCT
#undef dcm_dump_STD_STRUCT
#endif
#define dcm_dump_STD_STRUCT(dcmScope, dcmStd)

#ifdef dcm_debugInit
#undef dcm_debugInit
#endif
#define dcm_debugInit(dcmData)

#ifdef dcm_debugTerm
#undef dcm_debugTerm
#endif
#define dcm_debugTerm(dcmData)

#ifdef dcm_debugSubruleLoad
#undef dcm_debugSubruleLoad
#endif
#define dcm_debugSubruleLoad(dcmScope, dcmSubRule)

#ifdef dcm_debugSubrulePreLoad
#undef dcm_debugSubrulePreLoad
#endif
#define dcm_debugSubrulePreLoad(dcmScope, dcmSubRule)

#ifdef dcm_debugSegData
#undef dcm_debugSegData
#endif
#define dcm_debugSegData(dcmScope, dcmPC)

#ifdef dcm_debugNodeCall
#undef dcm_debugNodeCall
#endif
#define dcm_debugNodeCall(dcmScope, dcmName)

#ifdef dcm_debugStoreCall
#undef dcm_debugStoreCall
#endif
#define dcm_debugStoreCall(dcmScope, dcmName)
```

```
#ifdef dcm_debugSlotStoreCall
#undef dcm_debugSlotStoreCall
#endif
#define dcm_debugSlotStoreCall(dcmScope, dcmName)

#ifdef dcm_debugDelayMatrix
#undef dcm_debugDelayMatrix
#endif
#define dcm_debugDelayMatrix(dcmScope,dcmS1,dcmS2,dcmS3,dcmS4,
                             dcmS5,dcmS6)

#ifdef dcm_debugMethodsClause
#undef dcm_debugMethodsClause
#endif
#define dcm_debugMethodsClause(dcmScope,dcmS1,dcmS2)

#ifdef dcm_debugTestMatrix
#undef dcm_debugTestMatrix
#endif
#define dcm_debugTestMatrix(dcmScope,dcmS1,dcmS2,dcmS3,dcmS4,dcmS5,
                            dcmI6,dcmI7,dcmS8)

#ifdef dcm_debugPathSegment
#undef dcm_debugPathSegment
#endif
#define dcm_debugPathSegment(dcmScope,dcmStd,dcmI1)

#ifdef dcm_debugTestSegment
#undef dcm_debugTestSegment
#endif
#define dcm_debugTestSegment(dcmScope,dcmStd,dcmI1)

#ifdef dcm_debugNetSegment
#undef dcm_debugNetSegment
#endif
#define dcm_debugNetSegment(dcmScope,dcmStd,dcmS1,dcmS2,dcmI3)

#ifdef SET_DCM_DEBUG_LEVEL
#undef SET_DCM_DEBUG_LEVEL
#endif
#define SET_DCM_DEBUG_LEVEL(dcmLevel) (DCM_OFF)

#ifdef SET_DCM_DEBUG_LEVEL_VOID
#undef SET_DCM_DEBUG_LEVEL_VOID
#endif
#define SET_DCM_DEBUG_LEVEL_VOID(dcmLevel)

#ifdef GET_DCM_DEBUG_LEVEL
#undef GET_DCM_DEBUG_LEVEL
#endif
#define GET_DCM_DEBUG_LEVEL() (DCM_OFF)
```

```
#ifdef dcm_debugIntLoop
#undef dcm_debugIntLoop
#endif
#define dcm_debugIntLoop(dcmScope, dcmS1, dcmI1, dcmI2, dcmI3)

#ifdef dcm_debugLoopResult
#undef dcm_debugLoopResult
#endif
#define dcm_debugLoopResult(dcmScope, dcmd)

#ifdef dcm_debugLoopStart
#undef dcm_debugLoopStart
#endif
#define dcm_debugLoopStart(dcmScope, dcmd)

#ifdef dcm_debugLoopEnd
#undef dcm_debugLoopEnd
#endif
#define dcm_debugLoopEnd(dcmScope, dcmd)

#ifdef dcm_debugPinLoop
#undef dcm_debugPinLoop
#endif
#define dcm_debugPinLoop(dcmScope, dcmS1, dcmP1)

#ifdef dcm_debugExposeChainForward
#undef dcm_debugExposeChainForward
#endif
#define dcm_debugExposeChainForward(dcmScope,dcmName,dcmrc)

#ifdef dcm_debugExposeChainStops
#undef dcm_debugExposeChainStops
#endif
#define dcm_debugExposeChainStops(dcmScope,dcmName,dcmrc)

#ifdef DCM_STRUCT_DUMPER
#undef DCM_STRUCT_DUMPER
#endif
#define DCM_STRUCT_DUMPER(a) ((DCMStructureDumper)NULL)

#ifdef dcm_checkWritableArray
#undef dcm_checkWritableArray
#endif

#ifdef dcm_checkWritableStruct
#undef dcm_checkWritableStruct
#endif

#define dcm_checkWritableArray(std,arr,vn,fn)
#define dcm_checkWritableStruct(std,str,vn,fn)

#ifdef dcmVariablePrint
#undef dcmVariablePrint
#endif
```

```
#define dcmVariablePrint(str,call)

#ifdef dcm_debugGateForLevel
#undef dcm_debugGateForLevel
#endif
#define dcm_debugGateForLevel(lvl,rs) 0

#endif                                  /* DCM_GUTS_OR_DEBUG           */

#endif                                  /* _H_DCMDEBUG                 */
```

8.23 Standard array (dcmgarray.h) file

This section lists the `dcmgarray.h` file.

```
/*****************************************************************
** INCLUDE NAME..... dcmgarray.h
**
** PURPOSE.......... General declares for DCL Arrays.
**
** NOTES............ These functions and types are needed in
**                       Applications that use DCL Arrays.
**                       Rules        that use DCL Arrays.
**                       Rule C-code  that uses DCL Arrays.
**
** ASSUMPTIONS.......
**
** RESTRICTIONS.....
**
** LIMITATIONS......
**
** DEVIATIONS.......
**
** AUTHOR(S)........ Peter C. Elmendorf
**
** CHANGES:
**
*****************************************************************/
#ifndef _H_DCMGARRAY
#define _H_DCMGARRAY

#define DCM_ARRAY_OBJECT 0

/*****************************************************************
** Abstract type for arrays.
*****************************************************************/
typedef void DCM_ARRAY;

/*****************************************************************
** Abstract type for functions that print arrays.
*****************************************************************/
typedef void (*DCM_ArrayFormatFunction)(const DCM_ARRAY *);
```

```
/*****************************************************************
** Abstract type for user-written functions that initialize arrays.
*****************************************************************/
typedef int (*DCM_ArrayInitUserFunction)(DCM_ARRAY *);

/*****************************************************************
** Abstract type for DCL-written functions that initialize arrays.
*****************************************************************/
typedef void (*DCM_ArrayInitDCMFunction)(DCM_ARRAY *);

/*****************************************************************
** Scalars for the type of the items in the array.
*****************************************************************/
typedef enum DCM_Array_Element_Types {
  DCM_ATYPE_ERROR,                /* OOPS!                          */
  DCM_ATYPE_Integer,              /* INTEGER                        */
  DCM_ATYPE_String,               /* STRING                         */
  DCM_ATYPE_Double,               /* DOUBLE (NUMBER)                */
  DCM_ATYPE_Float,                /* FLOAT or NUMBER in TABLEDEF DATA*/
  DCM_ATYPE_Function,             /* Function array.                */
  DCM_ATYPE_Complex,              /* Complex array.                 */
  DCM_ATYPE_Void,                 /* void array.                    */
  DCM_ATYPE_Structur              /* Array of structure (ptr).      */
  /*************************************************************
  ** Future additions will go here.
  *************************************************************/

  DCM_ATYPE_MAX                   /* Ceiling.                       */

} DCM_ATYPE;

/*****************************************************************
** Array variant types for helpful use in the standard.
*****************************************************************/
typedef DCM_INTEGER         DCM_INTEGER_ARRAY;
typedef DCM_STRING          DCM_STRING_ARRAY;
typedef DCM_DOUBLE          DCM_DOUBLE_ARRAY;
typedef DCM_FLOAT           DCM_FLOAT_ARRAY;
typedef DCM_GeneralFunction DCM_FUNCTION_ARRAY;
typedef DCM_COMPLEX         DCM_COMPLEX_ARRAY;
typedef DCM_VOID            DCM_VOID_ARRAY;

/*****************************************************************
** For back compatibility ONLY.
*****************************************************************/
#define DCM_ATYPE_Real DCM_ATYPE_Double

DCM_XC const char *dcm_array_type_string(DCM_ATYPE);

/*****************************************************************
** Array initialization option scalars.
**
** Initialization options:
**
```

```
** DCM_AINIT_doNotInitialize - do not initialize.
**
** DCM_AINIT_initAllZeroes   - initialize space to all zeroes.
**
** DCM_AINIT_initByType      - initialize space depending on type:
**                               INTEGER - MININT
**                               STRING  - NULL
**                               NUMBER  - NaNS
**                               PIN     - NULL
**                               FLOAT   - NaNS
**                               DOUBLE  - NaNS
**                               VOID    - NULL
**                               user type - call the initializer
**                                           function if present. Do
**                                           nothing if initializer
**                                           function is not present
**                                           (NULL).
**
** DCM_AINIT_useFunction     - call the initializer function if present.
**                             Do nothing if initializer function is
**                             not present (NULL).
*****************************************************************/
typedef enum DCM_Array_Initialization {
  DCM_AINIT_doNotInitialize,    /* Do not initialize at all.          */
  DCM_AINIT_initAllZeroes,      /* Init all elements to zeroes.       */
  DCM_AINIT_initByType,         /* Init by type of element.           */
  DCM_AINIT_useFunction,        /* Initialize with a function.        */
  DCM_AINIT_compilerInits,      /* For compiler-defined minimum inits.*/
  DCM_AINIT_debugInits,         /* For compiler-defined debug inits.  */
  DCM_AINIT_MAX
} DCM_AINIT;

/*****************************************************************
** Attributes for DCM_ARRAYS.
**
** Unused bits are all RESERVED for future expansion.
**
** It is important that this be a “char”.
*****************************************************************/
typedef unsigned char DCM_AATTS;

/*****************************************************************
** Attributes which can be set by the user.
*****************************************************************/

/*****************************************************************
** DCM_AATTS_DEFAULT is the only value permitted right now.
** It is symbolic, meaning “take all default values.”
*****************************************************************/
#define DCM_AATTS_DEFAULT            0xFF

/*****************************************************************
** sizeof() for a DCM_ARRAY.
** Returns size (in bytes) of array space (just the elements.)
```

```
** Includes any padding that may be present.  Just like C sizeof().
** Return zero on error.
*****************************************************************/
DCM_XC size_t dcm_sizeof_DCM_ARRAY(const DCM_ARRAY *array);

/*****************************************************************
** Given a DCM_ARRAY, return the number of dimensions.
**
** Return -1 on error.
*****************************************************************/
DCM_XC int dcm_getNumDimensions(const DCM_ARRAY *array);

/*****************************************************************
** Given a DCM_ARRAY, return the number of elements in each dimension.
** Caller supplies space to write answer into.
**
** RETURN VALUE:  answer if OK,  NULL on error.
*****************************************************************/
DCM_XC int *dcm_getNumElementsPer(const DCM_ARRAY *array, int *answer);

/*****************************************************************
** Given a DCM_ARRAY and a dimension, return the number of elements
** in that dimension.
**
** Return -1 on error.
*****************************************************************/
DCM_XC int dcm_getNumElements(const DCM_ARRAY *array, int dimension);

/*****************************************************************
** Given a DCM_ARRAY, return the type of the elements.
**
** Return DCM_Array_Element_ERROR on error.
*****************************************************************/
DCM_XC DCM_ATYPE dcm_getElementType(const DCM_ARRAY *array);

/*****************************************************************
** Function for the user to print out vital stats of a DCM_ARRAY.
*****************************************************************/
typedef enum DCM_AVS {
  DCM_AVS_DoNothing,
  DCM_AVS_JustStats,
  DCM_AVS_Describe
} DCM_AVS;

DCM_XC void dcm_print_array_stats
  (const DCM_ARRAY    *array,         /* -> array to print.           */
   DCM_AVS             options,       /* options.                     */
   /*****************************************************************
   ** Optional formatter function to print a debug dump of the array.
   ** If NULL, dcm_print_array_stats does its simple dump to stderr.
   *****************************************************************/
   DCM_ArrayFormatFunction formatter);

#endif                                 /* _H_DCMGARRAY                 */
```

8.24 DCM user array defines (dcmuarray.h) file

This section lists the `dcmuarray.h` file.

```
/*****************************************************************************
** INCLUDE NAME..... dcmuarray.h
**
** PURPOSE.......... User structures and fucntions to support arrays.
**
** NOTES.............
**
** ASSUMPTIONS.......
**
** RESTRICTIONS.....
**
** LIMITATIONS......
**
** DEVIATIONS.......
**
** AUTHOR(S)......... Peter C. Elmendorf
**
** CHANGES:
**
*****************************************************************************/
#ifndef _H_DCMUARRAY
#define _H_DCMUARRAY 1

/*****************************************************************************
** APPLICATION SERVICE to allocate space for a new array.
**
** Return NULL on error.
*****************************************************************************/

DCM_XC
DCM_ARRAY *dcm_new_DCM_ARRAY
  (int              numDims,          /* Number of dimensions.          */
   const int       *elementsPer,      /* -> # of elements per dimension.*/
   size_t           elementSize,      /* Size of an element.            */
   DCM_ATYPE        elementType,      /* Type of an element.            */
   /*****************************************************************
   ** Flags made from DCM_AATTS.
   ** Right now, value must be: DCM_AATTS_DEFAULT
   *****************************************************************/
   DCM_AATTS     attributes,
   /*****************************************************************
   ** Initialization option:
   **
   ** zero - do not initialize.
   ** 1    - initialize space to all zeroes.
   ** 2    - initialize space depending on type:
   **                INTEGER - MININT
   **                STRING  - NULL
   **                NUMBER  - NaNS
   **                PIN     - NULL
```

```
        **                FLOAT   - NaNS
        **                (user type - not supported.)
        ************************************************************/
        DCM_AINIT     initialize,
        DCM_ArrayInitUserFunction initializer);  /* Init function.        */

    /************************************************************************
    ** sizeof() for a DCM_ARRAY.
    ** Returns size (in bytes) of array space (just the elements.)
    ** Return zero on error.
    ************************************************************************/
    DCM_XC
    size_t dcm_sizeof_DCM_ARRAY(const DCM_ARRAY *array);

    /************************************************************************
    ** Application lock on a DCM_ARRAY.
    **
    ** This uses the same lock counter as ASSIGN statements etc use.
    ** If you lock, you must unlock.
    ** Locks and unlocks only the base object.
    ************************************************************************/
    DCM_XC
    int dcm_lock_DCM_ARRAY(DCM_ARRAY *array);

    DCM_XC
    int dcm_unlock_DCM_ARRAY(DCM_ARRAY *array);

    /************************************************************************
    ** Given a DCM_ARRAY, return the number of dimensions.
    **
    ** Return -1 on error.
    ************************************************************************/
    DCM_XC
    int dcm_getNumDimensions(const DCM_ARRAY *array);

    /************************************************************************
    ** Given a DCM_ARRAY, return the number of elements in each dimension.
    ** Caller supplies space to write answer into.
    **
    ** RETURN VALUE:  answer if OK,  NULL on error.
    ************************************************************************/
    DCM_XC
    int *dcm_getNumElementsPer(const DCM_ARRAY *array, int *answer);

    /************************************************************************
    ** Given a DCM_ARRAY and a dimension, return the number of elements
    ** in that dimension.
    **
    ** Return -1 on error.
    ************************************************************************/
    DCM_XC
    int dcm_getNumElements(const DCM_ARRAY *array, int dimension);
```

```
/*****************************************************************
** Given a DCM_ARRAY, return the type of the elements.
**
** Return DCM_Array_Element_ERROR on error.
*****************************************************************/
DCM_XC
DCM_ATYPE dcm_getElementType(const DCM_ARRAY *array);

/*****************************************************************
** Given a DCM_ARRAY, return 1 if it is empty, zero if it is not.
*****************************************************************/
DCM_XC
int dcm_isArrayEmpty(const DCM_ARRAY *array);

/*****************************************************************
** Operations.
*****************************************************************/

/*****************************************************************
** Array compare.
**
** Returns zero for absolutely equal, nonzero for not equal.
*****************************************************************/
DCM_XC int dcm_arraycmp(const DCM_ARRAY *a1, const DCM_ARRAY *a2);

/*****************************************************************
** Return pointer to the specified element in the array.
** Intended for development and debugging purposes.
** Not high performance.  Has high amounts of checking.
**
** Returns NULL on error.
*****************************************************************/
DCM_XC
void *dcm_index_array(const DCM_ARRAY     *array,
                      const int           *index);

#endif                                  /* _H_DCMUARRAY                 */
```

8.25 Standard platform-dependency (dcmpltfm.h) file

This section lists the `dcmpltfm.h` file.

```
#ifndef _H_DCMPLTFM
#define _H_DCMPLTFM
/*****************************************************************
** INCLUDE NAME..... dcmpltfm.h
**
** PURPOSE.........
** This include provides platform-dependent definitions for DCM.
**
** ASSUMPTIONS.......
**
** RESTRICTIONS.....
```

```
**
** LIMITATIONS......
**
** DEVIATIONS.......
**
** AUTHOR(S)......... Peter C. Elmendorf
**
** CHANGES:
** 01 Oct 93  PCE  add HP
** 11 Feb 98  Unmesh Ballal added INTEL specific defines.
**
*****************************************************************************/

/*****************************************************************************
** Make sure a platform is chosen.
*****************************************************************************/
#if !defined(_IBMRS) && !defined(_SUN) && !defined(_HP)
    && !defined(_SOL) && !defined(INTEL)
#  if !defined(_IBM_DCM_PLATFORM_OVERRIDE)

    /*************************************************************************
    ** This deliberate syntax error is fed to the or C compiler
    ** when the user has failed to define a platform at compile time.
    **
    ** We must catch this omission, lest inappropriate code be
    ** accidentally compiled without any mention of this situation.
    *************************************************************************/
    One_platform_must_be_chosen_when_compiling_DCM_code
             Choose_one_of   -D_IBMRS  -D_SUN  -D_HP -D_SOL

#  endif
#endif

/*****************************************************************************
** System includes.
*****************************************************************************/
#ifdef _IBMRS

#ifndef  _POSIX_SOURCE
#  define _POSIX_SOURCE
#endif

#ifndef _ALL_SOURCE
#  define _ALL_SOURCE
#endif

#include <sys/types.h>
#endif

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <math.h>
```

```
#if defined(__cplusplus)
    && !defined(_IBM_DCL_SPECIAL_OVERRIDE_XC_FACTOR)

#define DCM_XC extern “C”
#define DCM_XCX extern “C”
#define DCM_XC_OPEN extern “C” {
#define DCM_XC_CLOSE }

#else

#define DCM_XC
#define DCM_XCX extern
#define DCM_XC_OPEN
#define DCM_XC_CLOSE

#endif

/*****************************************************************
** DCL_DEBUG and DCL_DEBUG_INIT same as their NDCL counterparts.
*****************************************************************/
#if defined(DCL_DEBUG) && !defined(NDCL_DEBUG)
#define NDCL_DEBUG 1
#endif

#if defined(DCL_DEBUG_INIT) && !defined(NDCL_DEBUG_INIT)
#define NDCL_DEBUG_INIT 1
#endif

#if defined(NDCL_DEBUG) && !defined(DCL_DEBUG)
#define DCL_DEBUG 1
#endif

#if defined(NDCL_DEBUG_INIT) && !defined(DCL_DEBUG_INIT)
#define DCL_DEBUG_INIT 1
#endif

/*****************************************************************
** DCL_DEBUG_INIT implies DCL_DEBUG
*****************************************************************/
#ifdef DCL_DEBUG_INIT
#  ifndef DCL_DEBUG
#    define DCL_DEBUG 1
#  endif
#endif

/*****************************************************************
** Internal compile or rule in debug mode?
*****************************************************************/
#ifdef DCM_GUTS
#define DCM_GUTS_OR_DEBUG 1
#else
#  ifdef DCL_DEBUG
#  define DCM_GUTS_OR_DEBUG 1
```

```
   #endif
#endif

/*********************************************************************
** The #define NULL statment is meant to override the
** #define NULL ((void *)0) declaration found in stdlib.h
*********************************************************************/
#undef NULL
#define NULL 0

#ifdef _IBM_DCM_PLATFORM_OVERRIDE
#include <dcmpltov.h>
#endif

/*===================================================================*/

/*********************************************************************
** Special includes so DCM will work on the RS.
*********************************************************************/
#ifdef _IBMRS

#define DCM_DEV_NULL_FILE_NAME "/dev/null"

#define _IBMAIX 1                       /* A unix-ish system.        */
#define DCM_DYNAMIC_LOAD_AVAILABLE 1    /* Dynamic load exists.      */

#ifndef _ALL_SOURCE
#  define _ALL_SOURCE
#endif

#include <float.h>                      /* Floating point.           */

#endif                                  /* _IBMRS                    */

/*===================================================================*/

/*********************************************************************
** Special includes so DCM will work on SOLARIS.
*********************************************************************/
#ifdef _SOL
#  ifndef _SUN
#    define _SUN 1
#  endif
#endif

/*********************************************************************
** Special includes so DCM will work on the SUN.
*********************************************************************/
#ifdef _SUN

#define DCM_DEV_NULL_FILE_NAME "/dev/null"

#define _IBMAIX 1                       /* A unix-ish system.        */
#define DCM_DYNAMIC_LOAD_AVAILABLE 1    /* Dynamic load exists.      */
```

```
typedef unsigned long ulong;           /* A BSD type.                  */

typedef struct dcm_2int { int a; int b; } dcm_2int;
/*****************************************************************
** Define the Float Signalling Not a Number.
*****************************************************************/
static       int  dcm_SNANFV = 0x7f855555;
static const float * const dcm_SNANFP = (float *)&dcm_SNANFV;
#define FLT_SNAN (*dcm_SNANFP)

/*****************************************************************
** Define the Float Quiet Not a Number.
*****************************************************************/
static       int  dcm_QNANFV = 0x7fc00000;
static const float * const dcm_QNANFP = (float *)&dcm_QNANFV;
#define FLT_QNAN (*dcm_QNANFP)

/*****************************************************************
** Define the Dbl Signalling Not a Number.
*****************************************************************/
static       dcm_2int  dcm_SNANDV = { 0x7ff55555, 0x55555555 };
static const double * const dcm_SNANDP = (double *)&dcm_SNANDV;
#define DBL_SNAN (*dcm_SNANDP)

/*****************************************************************
** Define the DBL Quiet Not a Number.
*****************************************************************/
static       dcm_2int  dcm_QNANDV = { 0x7ff80000, 0x00000000 };
static const double * const dcm_QNANDP = (double *)&dcm_QNANDV;
#define DBL_QNAN (*dcm_QNANDP)

#ifndef _SOL
#  ifdef __cplusplus
     class DCM_RuleScope_Z;
#    define DCM_RuleScope DCM_RuleScope_Z
     class DCM_STD_STRUCT_Z;
#    define DCM_STD_STRUCT DCM_STD_STRUCT_Z
#  endif
#endif

#endif                                 /* _SUN                         */

/*===============================================================*/

/*****************************************************************
** Special includes so DCM will work on the HP.
*****************************************************************/
#ifdef _HP

#define DCM_DEV_NULL_FILE_NAME "/dev/null"

#define _IBMAIX 1                      /* A unix-ish system.           */
#define DCM_DYNAMIC_LOAD_AVAILABLE 1   /* Dynamic load exists.         */
```

```
typedef struct dcm_2int  { int  a; int  b; } dcm_2int ;
/*****************************************************************
** Define the Float Signalling Not a Number.
*****************************************************************/
static       int  dcm_SNANFV = 0x7f855555;
static const float * const dcm_SNANFP = (float *)&dcm_SNANFV;
#define FLT_SNAN (*dcm_SNANFP)

/*****************************************************************
** Define the Float Quiet Not a Number.
*****************************************************************/
static       int  dcm_QNANFV = 0x7fc00000;
static const float * const dcm_QNANFP = (float *)&dcm_QNANFV;
#define FLT_QNAN (*dcm_QNANFP)

/*****************************************************************
** Define the Dbl Signalling Not a Number.
*****************************************************************/
static       dcm_2int  dcm_SNANDV = { 0x7ff55555, 0x55555555 };
static const double * const dcm_SNANDP = (double *)&dcm_SNANDV;
#define DBL_SNAN (*dcm_SNANDP)

/*****************************************************************
** Define the DBL Quiet Not a Number.
*****************************************************************/
static       dcm_2int  dcm_QNANDV = { 0x7ff80000, 0x00000000 };
static const double * const dcm_QNANDP = (double *)&dcm_QNANDV;
#define DBL_QNAN (*dcm_QNANDP)

#  ifdef __cplusplus
     class DCM_RuleScope_Z;
#    define DCM_RuleScope DCM_RuleScope_Z
     class DCM_STD_STRUCT_Z;
#    define DCM_STD_STRUCT DCM_STD_STRUCT_Z
#  endif
#endif                                  /* _HP                          */

#ifdef INTEL

#define DCM_DEV_NULL_FILE_NAME “nul”

#define DCM_DYNAMIC_LOAD_AVAILABLE 1   /* Dynamic load exists.         */

typedef struct dcm_2int  { int  a; int  b; } dcm_2int ;
/*****************************************************************
** Define the Float Signalling Not a Number.
*****************************************************************/
static       int  dcm_SNANFV = 0x7f855555;
static const float * const dcm_SNANFP = (float *)&dcm_SNANFV;
#define FLT_SNAN (*dcm_SNANFP)

/*****************************************************************
** Define the Float Quiet Not a Number.
*****************************************************************/
```

```
static       int  dcm_QNANFV = 0x7fc00000;
static const float * const dcm_QNANFP = (float *)&dcm_QNANFV;
#define FLT_QNAN (*dcm_QNANFP)

/*****************************************************************
** Define the Dbl Signalling Not a Number.
*****************************************************************/
static       dcm_2int  dcm_SNANDV = { 0x7ff55555, 0x55555555 };
static const double * const dcm_SNANDP = (double *)&dcm_SNANDV;
#define DBL_SNAN (*dcm_SNANDP)

/*****************************************************************
** Define the DBL Quiet Not a Number.
*****************************************************************/
static       dcm_2int  dcm_QNANDV = { 0x7ff80000, 0x00000000 };
static const double * const dcm_QNANDP = (double *)&dcm_QNANDV;
#define DBL_QNAN (*dcm_QNANDP)

#  ifdef __cplusplus
     class DCM_RuleScope_Z;
#    define DCM_RuleScope DCM_RuleScope_Z
     class DCM_STD_STRUCT_Z;
#    define DCM_STD_STRUCT DCM_STD_STRUCT_Z
#  endif
#endif                                  /* INTEL                 */
/*===============================================================*/
/*===============================================================*/
/*              GENERIC  DEFINITIONS!                            */
/*===============================================================*/
/*===============================================================*/

/*****************************************************************
** Here are the GENERIC PLATFORM DEFINITIONS.
** These definitions are not specific to a platform, but are common
** across a family of platforms.
*****************************************************************/

/*****************************************************************
**          AIX and other IX style platforms.
*****************************************************************/
#ifdef _IBMAIX
#define DCM_MIN_INT ((int )0x80000000) /* Smallest integer.      */

#endif

#ifdef INTEL
#ifndef INT_MIN
#include <limits.h>
#endif

#define DCM_MIN_INT (INT_MIN)          /* Smallest integer.      */
#endif
/*===============================================================*/
#endif                                  /* _H_DCMPLTFM           */
```

8.26 Standard state variables (dcmstate.h) file

This section lists the `dcmstate.h` file.

```
#ifndef _H_DCMSTATE
#define _H_DCMSTATE 1
/*****************************************************************
** INCLUDE NAME..... dcmstate.h
**
** PURPOSE.........
** Add the DCM state variables.
**
** NOTES.............
**
** ASSUMPTIONS.......
**
** RESTRICTIONS.....
**   THE DECLARATIONS INCLUDED BELOW ARE NOT TO BE CONSIDERED
**   AS PART OF THE INTERFACE.  DO NOT REFERENCE THESE SYMBOLS IN
**   APPLICATION CODE.  THEY MAY CHANGE OVER TIME.  ANY CHANGE
**   TO THIS DECLARATION WILL NOT BE CONSIDERED AS GROUNDS FOR
**   AN APAR OR ANY OTHER KIND OR FORM OF COMPLAINT OR
**   SUGGESTION!
**
** LIMITATIONS..... DO NOT TOUCH!
**
** DEVIATIONS.......
**
** AUTHOR(S)......... Peter C. Elmendorf
**
** CHANGES:
**
*****************************************************************/

/*****************************************************************
** The internal state block.
*****************************************************************/
typedef struct DCM_StateBlock {        /* Hands OFF!                */
  void *dcmNodeTree;                   /* Hands OFF!                */
  void *dcmAnyinTree;                  /* Hands OFF!                */
  void *dcmAnyoutTree;                 /* Hands OFF!                */
  void **dcm_cb_data;                  /* Hands OFF!                */
  void *dcm_lscope;                    /* Hands OFF!                */
  void *reserved3;                     /* Hands OFF!                */
  short dcm_tf;                        /* Hands OFF!                */
  unsigned short stateFlags;           /* Hands OFF!                */
  unsigned int  dcmDepth;              /* Hands OFF!                */
  void *olist;                         /* Hands OFF!                */
  void **dcm_cdb_data;                 /* Hands OFF!                */
  void *reserved6;                     /* Hands OFF!                */
  void *reserved7;                     /* Hands OFF!                */
  void *reserved8;                     /* Hands OFF!                */
} DCM_StateBlock;                      /* Hands OFF!                */
```

```
/*************************************************************************
** Macros for ** OFFICIAL DCM CODE ONLY ** to access magic fields.
*************************************************************************/
#define DCM_GET_NODE_TREE() \
  ((DCM_tree   *)(std_struct->dcmStates->dcmNodeTree))


#define DCM_GET_ANYIN_TREE() \
  ((DCM_tree   *)(std_struct->dcmStates->dcmAnyinTree))

#define DCM_GET_ANYOUT_TREE() \
  ((DCM_tree   *)(std_struct->dcmStates->dcmAnyoutTree))

#define DCM_SET_NODE_TREE(t) \
  (std_struct->dcmStates->dcmNodeTree=(void *)(t))

#define DCM_SET_ANYIN_TREE(t) \
  (std_struct->dcmStates->dcmAnyinTree=(void *)(t))

#define DCM_SET_ANYOUT_TREE(t) \
  (std_struct->dcmStates->dcmAnyoutTree=(void *)(t))

#define DCM_CB_PTR() \
  ((DCM_ConsistencyBlock **)(std_struct->dcmStates->dcm_cb_data))

#define DCM_CB() \
  (*(DCM_ConsistencyBlock **)(std_struct->dcmStates->dcm_cb_data))

#define DCM_CDB_PTR() \
  ((DCM_CellDataBlock **)(std_struct->dcmStates->dcm_cdb_data))

#define DCM_CDB() \
  (*(DCM_CellDataBlock **)(std_struct->dcmStates->dcm_cdb_data))

#define DCM_GET_LSCOPE(std) \
  ((DCM_LoadScope *)((std)->dcmStates->dcm_lscope))

#define DCM_SET_LSCOPE(std,x) \
  ((std)->dcmStates->dcm_lscope = (void *)(x))

#define DCM_GET_TF(std) \
  ((std)->dcmStates->dcm_tf)

#define DCM_SET_TF(std,x) \
  ((std)->dcmStates->dcm_tf = (x))

#define DCM_STATE_FLAGS(std) \
  ((std)->dcmStates->stateFlags)

#define DCM_DEPTH(std) ((std)->dcmStates->dcmDepth)

#define DCM_GET_OLIST(std) ((DCM_list *)((std)->dcmStates->olist))
#define DCM_SET_OLIST(std,l) (((std)->dcmStates->olist)=(void *)(l))
#endif                                  /* _H_DCMSTATE.                  */
```

8.27 Standard table descriptor(dcmutab.h)

This section lists the dcmutab.h file.

```
#ifndef _H_DCMUTAB
#define _H_DCMUTAB 1
#ifndef DCM_GUTS
/****************************************************************************
** INCLUDE NAME..... dcmutab.h
**
** PURPOSE.........
** This is but a partial declare for the DCMTableDescriptor.
**
** NOTES.............
** The full content of the data block is HIDDEN from the user.
** The first few words are exposed for use by DCM compiler
** generated code.
**
** ASSUMPTIONS.......
**
** RESTRICTIONS.....
** DO NOT MODIFY OR ACCESS THESE FIELDS!
**
** LIMITATIONS......
**
** DEVIATIONS.......
**
** AUTHOR(S)........ Peter C. Elmendorf
**
** CHANGES:
**
****************************************************************************/
typedef struct DCMTableDescriptor {
  void *mrr;                    /* Do not modify!                */
  int   flags;                  /* Do not modify!                */
  unsigned int    usageCount;   /* Do not modify!                */
} DCMTableDescriptor;
#endif

#endif                          /* _H_DCMUTAB.                   */
```

9. Parasitics

This section describes the parasitics used within the DPCS. The formal syntax is described using Backus-Naur Form (BNF). The following conventions are used:

a) Lowercase words, some containing embedded underscores, are used to denote syntactic categories (terminals and non-terminals), e.g.,

    name_map_entry

b) Syntactic categories have no special fonts when used in the BNF, but are italicized when defined or referred to in the text.

c) Boldface words are used to denote reserved keywords, operators, and punctuation marks as a required part of the syntax. These words appear in a larger font for distinction, e.g.,

    ***D_NET *Q :**

d) The `::=` operator separates the two parts of a 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 (e) shows the four options for a *suffix_bus_delim*.

e) A vertical bar separates alternative items (use one only) unless it appears in boldface, in which case it stands for itself, e.g.,

    suffix_bus_delim ::= **]** | **}** | **)** | **>**

f) 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.

g) 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 which may appear one or more times is listed first, followed by the item enclosed in braces, such as

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

h) Parenthesis are always part of the syntax in this section; therefore they always appear in boldface, such as:

    cnumber ::= **(**number number**)**

i) 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.

j) 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).

k) The main text uses `monospace` font for examples, file names, and references to constants such as `0`, `1`, or `x` values.

9.1 Introduction

The Standard Parasitic Exchange Format (SPEF) provides a standard medium to pass parasitic information between EDA tools during any stage in the design process. Parasitics can be represented on a net by net basis in many different levels of sophistication from a simple lumped capacitance, to a fully distributed RC tree, to a multiple pole AWE representation.

9.2 Targeted applications for SPEF

SPEF is suitable for use in many different tool combinations. Because parasitics can be represented in various levels of sophistication, *SPEF_file*s can communicate parasitic information throughout the design flow process. A design can be distributed between multiple *SPEF_file*s. The files can also communicate information such as slews and the "routing confidence" indicating at what stage of the design process and/or how the parasitics were generated. A diagram of how SPEF interfaces with various example applications is shown in Figure 9-1.

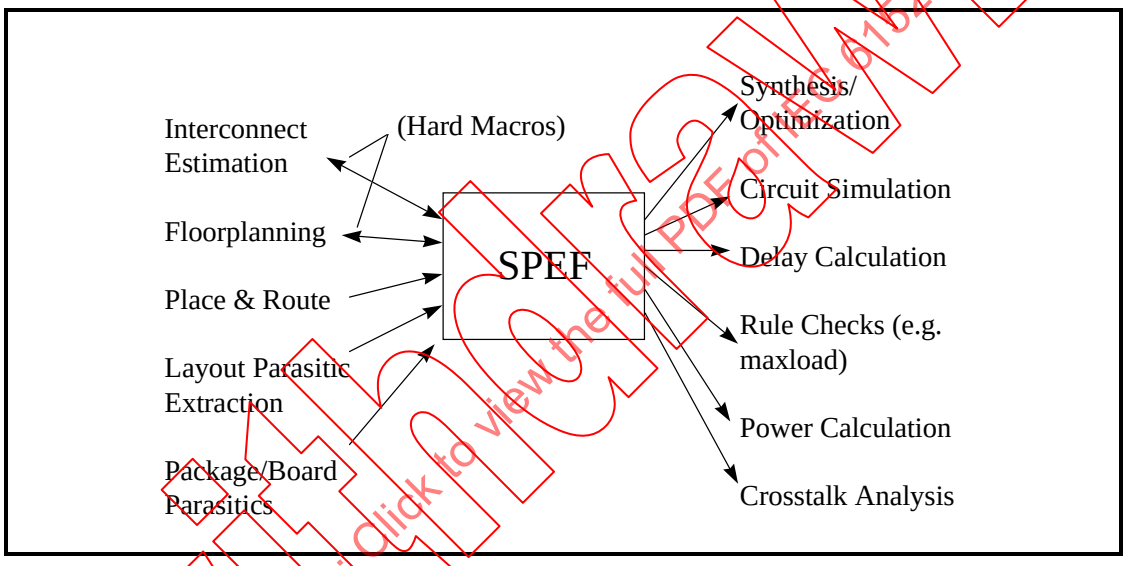


Figure 9-1 SPEF targeted applications

9.3 SPEF specification

This section details the SPEF grammar and *SPEF_file* syntax.

9.3.1 Grammar

Keywords, identifiers, characters and numbers are delimited by syntax characters, *white space*, or *newline*. Syntax characters are any nonalphanumeric characters required by the syntax. Alphanumeric characters include upper and lowercase alphabetic characters, all numbers, and the underscore (_) character. *White space* (*space* and *tab*) and *newline* may be used to separate lexical tokens, except for hierarchy separators in pathnames, pin delimiters or bus delimiters, where no *white space* or *newline* is allowed. All keywords in SPEF start with an asterisk (*) and are composed of all capital letters (for example, *SPEF).

9.3.1.1 Alphanumeric definition

The syntax for alphanumeric characters and numbers in SPEF is given in Syntax 9-1.

```
alpha ::= upper | lower
upper ::= A - Z
lower ::= a - z
digit ::= 0 - 9
sign ::= pos_sign | neg_sign
pos_sign ::= +
neg_sign ::= -
integer ::= [sign]<digit>{<digit>}
decimal ::= [sign]<digit>{<digit>}.{<digit>}
fraction ::= [sign].<digit>{<digit>}
exp ::= <radix><exp_char><integer>
radix ::= decimal | fraction
exp_char ::= E | e
float ::= decimal | fraction | exp
number ::= integer | float
pos_integer ::= <digit>{<digit>}
pos_decimal ::= <digit>{<digit>}.{<digit>}
pos_fraction ::= .<digit>{<digit>}
pos_exp ::= <pos_radix><exp_char><integer>
pos_radix ::= pos_integer | pos_decimal | pos_fraction
pos_float ::= pos_decimal | pos_fraction | pos_exp
pos_number ::= pos_integer | pos_float
```

Syntax 9-1—Syntax for alphanumeric characters

9.3.1.2 Names definition

The basic grammar for names in SPEF is given in Syntax 9-2.

```
path ::= [<hier_delim>]<bit_identifier>{<partial_path>}[<hier_delim>]
hier_delim ::= hchar
hchar ::= . | / | : | |
bit_identifier ::=
        <identifier>
      | <identifier><prefix_bus_delim><digit>{<digit>}[<suffix_bus_delim>]
identifier ::= <identifier_char>{<identifier_char>}
identifier_char ::= <escaped_char> | <alpha> | <digit> | _
escaped_char ::= \<escaped_char_set>
escaped_char_set ::= <special_char> | “
special_char ::=
      ! | # | $ | % | & | ` | ( | ) | * | + | , | - | . | / | : | ; | < | = | > | ? | @ | [ | \ | ] | ^ |
      ‘ | { | | | } | ~
prefix_bus_delim ::= [ | { | ( | < | : | .
suffix_bus_delim ::= ] | } | ) | >
partial_path ::= <hier_delim><bit_identifier>
physical_ref ::= <physical_name>{<partial_physical_ref>}
physical_name ::= name
name ::= qstring | identifier
partial_physical_ref ::= <hier_delim><physical_name>
qstring ::= “{qstring_char}”
qstring_char ::= special_char | alpha | digit | white_space | _
white_space ::= space | tab
```

Syntax 9-2—Syntax for SPEF names

9.3.2 File syntax

This section lists the syntax to use within an *SPEF_file*.

9.3.2.1 Basic file definition

The syntax for the base *SPEF_file* definition is given in Syntax 9-3.

```
SPEF_file ::=
      header_def [name_map] [power_def] [external_def] [define_def] internal_def
```

Syntax 9-3—Syntax for base SPEF_file definition

9.3.2.2 Header definition

The syntax for the *header* definition is given in Syntax 9-4.

```
header_def ::=
      SPEF_version design_name date vendor program_name program_version
      design_flow hierarchy_div_def pin_delim_def bus_delim_def unit_def
SPEF_version ::= *SPEF qstring
design_name ::= *DESIGN qstring
date ::= *DATE qstring
vendor ::= *VENDOR qstring
program_name ::= *PROGRAM qstring
program_version ::= *VERSION qstring
design_flow ::= *DESIGN_FLOW qstring {qstring}
hierarchy_div_def ::= *DIVIDER hier_delim
pin_delim_def ::= *DELIMITER pin_delim
pin_delim ::= hchar
bus_delim_def ::= *BUS_DELIMITER prefix_bus_delim [suffix_bus_delim]
unit_def ::= time_scale cap_scale res_scale induc_scale
```

Syntax 9-4—Syntax for SPEF_file header definition

The syntax for the *unit* definition is given in Syntax 9-5.

```
unit_def ::= time_scale cap_scale res_scale induc_scale
time_scale ::= *T_UNIT pos_number time_unit
time_unit ::= NS | PS
cap_scale ::= *C_UNIT pos_number cap_unit
cap_unit ::= PF | FF
res_scale ::= *R_UNIT pos_number res_unit
res_unit ::= OHM | KOHM
induc_scale ::= *L_UNIT pos_number induc_unit
induc_unit ::= HENRY | MH | UH
```

Syntax 9-5—Syntax for unit definition

9.3.2.3 Name map definition

The syntax for the *name mapping* definition is given in Syntax 9-6.

```
name_map ::= *NAME_MAP name_map_entry {name_map_entry}
name_map_entry ::= index mapped_item
index ::= *<pos_integer>
mapped_item ::= identifier | bit_identifier | path | name | physical_ref
```

Syntax 9-6—Syntax for name map definition

9.3.2.4 Power and ground nets definition

The syntax for *power nets* and *ground nets* definition is given in Syntax 9-7.

```
power_def ::= power_net_def [ground_net_def] | ground_net_def
power_net_def ::= *POWER_NETS net_name {net_name}
net_name ::= net_ref | pnet_ref
net_ref ::= index | path
pnet_ref ::= index | physical_ref
ground_net_def ::= *GROUND_NETS net_name {net_name}
```

Syntax 9-7—Syntax for power and ground definition

9.3.2.5 External definition

The syntax for the *external* definition is given in Syntax 9-8.

```
external_def ::= port_def [physical_port_def] | physical_port_def
port_def ::= *PORTS port_entry {port_entry}
port_entry ::= port_name direction {conn_attr}
port_name ::= [<inst_name><pin_delim>]<port>
inst_name ::= index | path
port ::= index | bit_identifier
direction ::= I | B | O
conn_attr ::= coordinates | cap_load | slews | driving_cell
physical_port_def ::= *PHYSICAL_PORTS pport_entry {pport_entry}
pport_entry ::= pport_name direction {conn_attr}
pport_name ::= [<physical_inst><pin_delim>]<pport>
physical_inst ::= index | physical_ref
pport ::= index | name
```

Syntax 9-8—Syntax for external definition

The syntax for *port* (*connection*) attributes definition is given in Syntax 9-9.

```
conn_attr ::= coordinates | cap_load | slews | driving_cell
coordinates ::= *C number number
cap_load ::= *L par_value
par_value ::= float | <float>:<float>:<float>
slews ::= *S par_value par_value [threshold threshold]
threshold ::= pos_fraction | <pos_fraction>:<pos_fraction>:<pos_fraction>
driving_cell ::= *D cell_type
cell_type ::= index | name
```

Syntax 9-9—Syntax for connection attributes definition

9.3.2.6 Hierarchical SPEF (entities) definition

The syntax for the *entities* definition supporting hierarchical SPEF is given in Syntax 9-10.

```
define_def ::= define_entry {define_entry}
define_entry ::=
        *DEFINE inst_name {inst_name} entity
      | *PDEFINE physical_inst entity
entity ::= qstring
```

Syntax 9-10—Syntax for entities definition

9.3.2.7 Internal definition

The syntax for the *internal* definition is given in Syntax 9-11.

```
internal_def ::= nets {nets}
nets ::= d_net | r_net | d_pnet | r_pnet
d_net ::=
        *D_NET net_ref total_cap
        [routing_conf] [conn_sec] [cap_sec] [res_sec] [induc_sec] *END
r_net ::= *R_NET net_ref total_cap [routing_conf] {driver_reduc} *END
d_pnet ::=
        *D_PNET pnet_ref total_cap
        [routing_conf] [pconn_sec] [pcap_sec] [pres_sec] [pinduc_sec] *END
r_pnet ::= *R_PNET pnet_ref total_cap [routing_conf] {pdriver_reduc} *END
```

Syntax 9-11—Syntax for internal definition

9.3.2.7.1 d_net definition

The syntax for the *d_net* definition is given in Syntax 9-12.

```
d_net ::=
        *D_NET net_ref total_cap
        [routing_conf] [conn_sec] [cap_sec] [res_sec] [induc_sec] *END
total_cap ::= par_value
routing_conf ::= *V conf
conf ::= pos_integer
conn_sec ::= *CONN conn_def {conn_def} {internal_node_coord}
cap_sec ::= *CAP cap_elm {cap_elm}
res_sec ::= *RES res_elem {res_elem}
induc_sec ::= *INDUC induc_elem {induc_elem}
```

Syntax 9-12—Syntax for d_net definition

a) The syntax for the *d_net connectivity* section definition is given in Syntax 9-13.

```
conn_sec ::= *CONN conn_def {conn_def} {internal_node_coord}
conn_def ::=
        *P external_connection direction {conn_attr}
      | *I internal_connection direction {conn_attr}
external_connection ::= port_name | pport_name
internal_connection ::= pin_name | pnode_ref
pin_name ::= <inst_name><pin_delim><pin>
pin ::= index | bit_identifier
pnode_ref ::= <physical_inst><pin_delim><pnode>
pnode ::= index | name
internal_node_coord ::= *N internal_node_name coordinates
internal_node_name ::= <net_ref><pin_delim><pos_integer>
```

Syntax 9-13—Syntax for d_net connectivity section definition

b) The syntax for the *d_net capacitance* section definition is given in Syntax 9-14.

```
cap_sec ::= *CAP cap_elm {cap_elem}
cap_elem ::=
        cap_id node_name par_value
      | cap_id node_name node_name2 par_value
cap_id ::= pos_integer
node_name ::=
        external_connection
      | internal_connection
      | internal_node_name
      | pnode_ref
node_name2 ::=
        node_name
      | <pnet_ref><pin_delim><pos_integer>
      | <net_ref2><pin_delim><pos_integer>
net_ref2 ::= net_ref
```

Syntax 9-14—Syntax for d_net capacitance section definition

c) The syntax for the *d_net resistance* section definition is given in Syntax 9-15.

```
res_sec ::= *RES res_elem {res_elem}
res_elem ::= res_id node_name node_name par_value
res_id ::= pos_integer
```

Syntax 9-15—Syntax for d_net resistance section definition

d) The syntax for the *d_net inductance* section definition is given in Syntax 9-16.

```
induc_sec ::= *INDUC induc_elem {induc_elem}
induc_elem::= induc_id node_name node_name par_value
induc_id::= pos_integer
```

Syntax 9-16—Syntax for d_net inductance section definition

9.3.2.7.2 r_net definition

The syntax for the *r_net* definition is given in Syntax 9-17.

```
r_net ::= *R_NET net_ref total_cap [routing_conf] {driver_reduc} *END
driver_reduc ::= driver_pair driver_cell pie_model load_desc
driver_pair ::= *DRIVER pin_name
driver_cell ::= *CELL cell_type
pie_model ::= *C2_R1_C1 par_value par_value par_value
load_desc ::= *LOADS rc_desc {rc_desc}
```

Syntax 9-17—Syntax for r_net definition

The syntax for the *r_net load description* definition is given in Syntax 9-18.

```
load_desc ::= *LOADS rc_desc {rc_desc}
rc_desc ::= *RC pin_name par_value [pole_residue_desc]
pole_residue_desc ::= pole_desc residue_desc
pole_desc ::= *Q pos_integer pole {pole}
pole ::= complex_par_value
complex_par_value ::=
        cnumber
      | number
      | cnumber:cnumber:cnumber
      | number:number:number
cnumber ::= (real_component imaginary_component)
real_component ::= number
imaginary_component ::= number
residue_desc ::= *K pos_integer residue {residue}
residue ::= complex_par_value
```

Syntax 9-18—Syntax for r_net load descriptions definition

9.3.2.7.3 d_pnet definition

The syntax for the *d_pnet* definition is given in Syntax 9-19.

```
d_pnet ::=
        *D_PNET pnet_ref total_cap
        [routing_conf] [pconn_sec] [pcap_sec] [pres_sec] [pinduc_sec] *END
pconn_sec ::= *CONN pconn_def {pconn_def} {internal_pnode_coord}
pcap_sec ::= *CAP pcap_elm {pcap_elm}
pres_sec ::= *RES pres_elem {pres_elem}
pinduc_sec ::= *INDUC pinduc_elem {pinduc_elem}
```

Syntax 9-19—Syntax for d_pnet definition

a) The syntax for the *d_pnet connectivity* section definition is given in Syntax 9-20.

```
pconn_sec ::= *CONN pconn_def {pconn_def} {internal_pnode_coord}
pconn_def ::=
        *P pexternal_connection direction {conn_attr}
      | *I internal_connection direction {conn_attr}
pexternal_connection ::= pport_name
internal_pnode_coord ::= *N internal_pnode_name coordinates
internal_pnode_name ::= <pnet_ref><pin_delim><pos_integer>
```

Syntax 9-20—Syntax for d_pnet connectivity section definition

b) The syntax for the *d_pnet capacitance* section definition is given in Syntax 9-21.

```
pcap_sec ::= *CAP pcap_elm {pcap_elem}
pcap_elem ::=
        cap_id pnode_name par_value
      | cap_id pnode_name pnode_name2 par_value
pnode_name ::=
        pexternal_connection
      | internal_connection
      | internal_pnode_name
      | pnode_ref
pnode_name2 ::=
        pnode_name
      | <net_ref><pin_delim><pos_integer>
      | <pnet_ref2><pin_delim><pos_integer>
pnet_ref2 ::= pnet_ref
```

Syntax 9-21—Syntax for d_pnet capacitance section definition

c) The syntax for the *d_pnet resistance* section definition is given in Syntax 9-22.

```
pres_sec ::= *RES pres_elem {pres_elem}
pres_elem ::= res_id pnode_name pnode_name par_value
```

Syntax 9-22—Syntax for d_net resistance section definition

d) The syntax for the *d_pnet inductance* section definition is given in Syntax 9-23.

```
pinduc_sec ::= *INDUC pinduc_elem {pinduc_elem}
pinduc_elem::= induc_id pnode_name pnode_name par_value
```

Syntax 9-23—Syntax for d_pnet inductance section definition

9.3.2.7.4 r_pnet definition

The syntax for the *r_pnet* definition is given in Syntax 9-24.

```
r_pnet ::= *R_NET pnet_ref total_cap [routing_conf] {pdriver_reduc} *END
pdriver_reduc ::= pdriver_pair driver_cell pie_model load_desc
pdriver_pair ::= *DRIVER internal_connection
```

Syntax 9-24—Syntax for r_pnet definition

9.3.3 Escaping rules

This section gives the escaping rules for identifiers in SPEF.

9.3.3.1 Special characters

Any character other than alphanumerics and underscore (_) shall be escaped when used in an identifier in a *SPEF_file*. These special characters are legal to use without escaping within a *qstring* but shall be escaped to use in an *identifier* or *bit_identifier*:

! # $ % & ` ( ) * + , -. / : ; < = > ? @ [\ ] ^ ‘ { | } ~

The quote (**“**) is not allowed within a *qstring*, but it can be used within an *identifier* or *bit_identifier* when escaped. Exceptions to the escaping rules are:

— the *pin_delim* between an instance and pin name, such as **:** in `I\$481:X`
— the *hier_delim* character, such as / in `/top/coreblk/cpu1/inreg0/I\$481`
— a *prefix_bus_delim* or *suffix_bus_delim* being used to denote a bit of a logical bus or an arrayed instance, such as `DATAOUT[12]`

9.3.3.2 Character escaping mechanism for identifiers in SPEF

Escape a character by preceding it with a backslash (\). An alphanumeric or underscore preceded with a backslash is legal, and merely maps to the unescaped character. For example, \A and A in an *SPEF_file* are both interpreted by an SPEF reader as A. Some characters shall not be included in an identifier under any circumstances, such as *whitespace* like *space* or *tab*, and control characters such as *newline*, carriage return, formfeed, etc.

9.3.4 Comments

// begins a single-line comment anywhere on the line, which is terminated by a newline. /* begins a multi-line comment, terminated by */. No nesting of comments is allowed; // appearing between /* and */ is treated as characters within the multi-line comment.

An application may ignore comments and is not required to pass them forward.

9.3.5 File semantics

This section describes the semantic intent underlying each construct.

a) *SPEF_file* ::= header_def [name_map] [power_def] [external_def] [define_def] internal_def

Each *SPEF_file* consists of these items and sections.

b) *header_def* ::= SPEF_version design_name date vendor program_name program_version design_flow hierarchy_div_def pin_delim_def bus_delim_def unit_def

This section contains basic global information about the design and/or how this *SPEF_file* was created.

c) *SPEF_version* ::= ***SPEF** qstring

qstring represents the SPEF version. The version described herein is “IEEE 1481-1998”.

d) *design_name* ::= ***DESIGN** qstring

qstring represents the name of the design for which this *SPEF_file* was generated.

e) *date* ::= ***DATE** qstring

qstring represents the date and time when this *SPEF_file* was generated.

f) *vendor* ::= ***VENDOR** qstring

qstring represents the name of the vendor of the program used to generate this *SPEF_file*.

g) *program_name* ::= ***PROGRAM** qstring

qstring represents the name of the program used to generate this *SPEF_file*.

h) *program_version* ::= ***VERSION** qstring

qstring represents the version number of the program used to generate this *SPEF_file*.

i) *design_flow* ::= ***DESIGN_FLOW** qstring {qstring}

This construct gives information about the content of this *SPEF_file* and/or at which stage in the design flow this *SPEF_file* was generated. It may save processing time to not have to determine if the *SPEF_file* contains certain information. The construct can identify where certain information being absent from the *SPEF_file* carries meaning.

The application reading the *SPEF_file* is thus able to correctly interpret the *SPEF_file* and determine whether its content is appropriate for the design flow being used or for this stage in the design flow.

New *qstring* values may be defined for specific design flows. The predefined values shown in Table 9-1 provide standard interpretation of common design flow information.

Table 9-1—Design flow values

Value	Definition
EXTERNAL_LOADS	External loads, if any, are fully specified in the *SPEF_file*. Absence of `EXTERNAL_LOADS` means they either are not specified or are not fully specified in this *SPEF_file*; if the application requires this information, the application shall have some other means of getting external load information, or if none is available, shall issue an error message and exit.
EXTERNAL_SLEWS	External slews, if any, are fully specified in the *SPEF_file*. Absence of `EXTERNAL_SLEWS` means they either are not specified or are not fully specified in this *SPEF_file*; if the application requires this information, the application shall have some other means of getting external slew information, or if none is available, shall issue an error message and exit.
FULL_CONNECTIVITY	All connectivity corresponding to the logical netlist for the design is included in the *SPEF_file*. Absence of `FULL_CONNECTIVITY` means if the application needs complete connectivity information, the application shall obtain it from another source, or if none is available, shall issue an error message and exit. Presence or absence of pnets (nets not corresponding to the logical netlist) or power and/or ground nets does not affect `FULL_CONNECTIVITY`; the value pertains to nets related to the design function.
MISSING_NETS	Some logical nets in the design are or may be missing from the *SPEF_file*. The application shall determine and fill in the missing information, such as merging missing logical net parasitics from another source or reading the netlist and estimating the missing parasitics. Absence of `MISSING_NETS` means the *SPEF_file* contains entries for all logical nets in the design. If an application requires all logical connectivity information to be present in the *SPEF_file*, it shall issue an error message and exit if `MISSING_NETS` is present. It shall be a semantic error for both `FULL_CONNECTIVITY` and `MISSING_NETS` to be listed in the same *SPEF_file*. Presence or absence of physical nets (pnets) and power and/or ground nets does not affect `MISSING_NETS`; the value pertains to logical nets related to the design function
NETLIST_TYPE_VERILOG	The *SPEF_file* uses *Verilog* type naming conventions. It shall be a semantic error for more than one netlist type to be listed for the same *SPEF_file*.
NETLIST_TYPE_VHDL87	The *SPEF_file* uses *VHDL87* naming conventions. It shall be a semantic error for more than one netlist type to be listed for the same *SPEF_file*.
NETLIST_TYPE_VHDL93	The *SPEF_file* uses *VHDL93* naming conventions. It shall be a semantic error for more than one netlist type to be listed for the same *SPEF_file*.
NETLIST_TYPE_EDIF	The *SPEF_file* uses *Edif* type naming conventions. It shall be a semantic error for more than one netlist type to be listed for the same *SPEF_file*.

Table 9-1—Design flow values *(continued)*

Value	Definition
ROUTING_CONFIDENCE *conf*	This specifies a default routing confidence value for all the nets contained in the *SPEF_file*. `ROUTING_CONFIDENCE` is further described under *d_net* semantics. *conf* ::= pos_integer
ROUTING_CONFIDENCE_ENTRY *conf string*	This `DESIGN_FLOW` construct defines routing confidence values to supplement the predefined routing confidence values. The *conf* value shall be consistent with predefined values described in the *d_net* semantics. *conf* ::= pos_integer *string* ::= <qstring_char>{<qstring_char>}
NAME_SCOPE *scope*	This `DESIGN_FLOW` construct specifies whether the *path*(s) contained in this *SPEF_file* are `LOCAL` (relative to this *SPEF_file*) or `FLAT` (relative to the top of the complete design). See "define_def ::= define_entry {define_entry}" on page 342. The default is `LOCAL`. This construct has no meaning in a top level *SPEF_file*. This construct is required if this *SPEF_file* is a physical instance not corresponding to a logical instance or it contains `internal_def` entries for pnets, since the value in this case shall be `FLAT`. *scope* ::= **LOCAL** | **FLAT**
SLEW_THRESHOLDS *threshold threshold*	This construct specifies default input slew thresholds for the design, where the first *threshold* is the low input threshold as a percentage of the voltage level for the input pin, and the second *threshold* is the high input threshold as a percentage of the voltage level for the input pin. *threshold* is a single or triplet *pos_fraction*.
PIN_CAP *cap_calc_method*	This construct specifies what type of pin capacitances are included in the *total_cap* entries for all nets in the *SPEF_file*. `NONE` designates no pin capacitances are included in the *total_cap* entries, `INPUT_OUTPUT` (the default value) designates both input and output pin capacitances are included, and `INPUT_ONLY` designates only input pin capacitances (no output pin capacitances) are included. *cap_calc_method* ::= **NONE** | **INPUT_OUTPUT** | **INPUT_ONLY**

j) *hierarchy_div_def* ::= ***DIVIDER** hier_delim

hier_delim is the hierarchy delimiter. Legal characters are the *hchar* set: **.**, **/**, **:**, or **|**.

k) *pin_delim_def* ::= ***DELIMITER** pin_delim

pin_delim is the delimiter between an instance name and pin name. Legal characters are the *hchar* set: **.**, **/**, **:**, or **|**. To allow for naming conventions having the *pin_delim* being the same character as the *hier_delim*, the application reading the *SPEF_file* needs to distinguish between the two based on context.

l) *bus_delim_def* ::= ***BUS_DELIMITER** prefix_bus_delim [suffix_bus_delim]

prefix_bus_delim denotes the opening of designation of a bus bit or arrayed instance number; *suffix_bus_delim* denotes the closing of the designation of the bus bit or arrayed instance, such as `[` and `]` in `DATA[2]`. It shall be a semantic error if a bus delimiter character is the same as the *hier_delim* or *pin_delim*. Legal characters for *prefix_bus_delim* are **[**, **{**, **(**, **<**, **:**, or **.**, while legal characters for *suffix_bus_delim* are **]**, **}**, **)**, or **>**. It shall be a semantic error if the *suffix_bus_delim* is not

the corresponding closing character for a *prefix_bus_delim* of **[**, **{**, **(**, or **<**. There normally is not a *suffix_bus_delim* if the *prefix_bus_delim* is **:** or **.**

Examples

A *SPEF_file* produced from a Verilog netlist would normally specify

```
*BUS_DELIMITER [ ]
```

so `dummy[21]` would be a legal SPEF bit_identifier with no escaping. If

```
*BUS_DELIMITER :
```

is used instead, then bit identifier `dummy:21` is legal.

m) *unit_def* ::= time_scale cap_scale res_scale induc_scale

This section defines the units for this *SPEF_file*.

n) *time_scale* ::= ***T_UNIT** pos_number time_unit

This specifies the time unit used throughout the *SPEF_file*. *pos_number* is a floating point number and *time_unit* is **NS** for nanoseconds or **PS** for picoseconds.

o) *cap_scale* ::= ***C_UNIT** pos_number cap_unit

This specifies the capacitive unit used throughout the *SPEF_file*. *pos_number* is a floating point number and *cap_unit* is **FF** for femtofarad or **PF** for picofarad.

p) *res_scale* ::= ***R_UNIT** pos_number res_unit

This specifies the resistive unit used throughout the *SPEF_file*. *pos_number* is a floating point number and *res_unit* is **OHM** for ohm or **KOHM** for kilo-ohm.

q) *induc_scale* ::= ***L_UNIT** pos_number induc_unit

This specifies the inductance unit used throughout the *SPEF_file*. *pos_number* is a floating point number and *induc_unit* is **HENRY** for henry, **MH** for milli-henry, or **UH** for micro-henry.

r) *name_map* ::= ***NAME_MAP** name_map_entry {name_map_entry}

A name map is an optional capability of SPEF to reduce file space by mapping a name which may be used multiple times in the *SPEF_file*. The first part of a name map entry is the *index* (hash id) used throughout the *SPEF_file* to represent the name and the second part is the name being mapped, as follows:

name_map_entry ::= index mapped_item

where

index ::= *****<pos_integer>
mapped_item ::= identifier | bit_identifier | path | name | physical_ref

s) *power_def* ::= power_net_def [ground_net_def] | ground_net_def

This optional section identifies logical and physical net name(s) which are power or ground nets. These logical and physical nets may or may not have `internal_def` entries and may or may not be captured in the design's netlist. These net names are commonly used in logical netlists to tie cell inputs high (power) or low (ground).

t) *power_net_def* ::= ***POWER_NETS** net_name {net_name}

This specifies the power net name(s) used in the *SPEF_file*. *net_name* can be a reference to a net or pnet,

where

net_name ::= net_ref | pnet_ref

net_ref ::= index | path

is a path, or an index to a path, which represents the logical hierarchy name of, or reference to, a logical net.

pnet_ref ::= index | physical_ref

is a physical reference, or an index to a physical reference, which represents the physical hierarchy name of, or reference to, a physical net.

Example

```
*POWER_NETS /top/core_0/vdd2 VDD *34
```

u) *ground_net_def* ::= ***GROUND_NETS** net_name {net_name}

This specifies the ground net name(s) used in the *SPEF_file*. *net_name* can be a reference to a logical net or to a pnet.

v) *external_def* ::= port_def [physical_port_def] | physical_port_def

This section defines the logical and physical ports for a group of parasitics. Connections to the parent *SPEF_file* or to the outside world from a top-level *SPEF_file* are made through these ports. This section optionally also specifies the drive strengths, slews, and capacitive loads on the ports. If a port can be referenced by both a logical and physical name, the logical name is recommended.

w) port_def ::= ***PORTS** port_entry {port_entry}

This defines the logical ports for the *SPEF_file*. Information shown in the ***PORTS** section shall be consistent with that in the ***CONN** section of applicable *d_nets*; `conn` attributes shown in the two sections are cumulative. If an application determines required information is missing from both sections, the application shall issue an error message and exit. All nets connected to ports shall be in detailed form (*d_net)* to provide for multiple *SPEF_files*; *r_net*s cannot connect to ports.

Each *port_entry* is defined as

port_entry ::= port_name direction {conn_attr}

The *port_name* is defined as

port_name ::= [<inst_name><pin_delim>]<port>

inst_name is a *path* or an *index* to a path denoting the instance of the logical entity owning the port, *pin_delim* is the *hchar* defined by ***DELIMITER** and *port* is a *bit_identifier* or an *index* to a bit_identifier denoting the name of the scalar or bus bit port of the logical entity. In a top-level *SPEF_file*, there is usually no *inst_name and pin_delim.*

The *direction* shall be **I**, **B**, or **O** (input, bidirectional, or output, respectively).

All *conn_attr* are optional and are shown in Table 9-2.

Table 9-2—conn_attr types

Type	Definition
***C** *number number*	Coordinates for the geometric location of the logical or physical port
***L** *par_value*	The capacitive load is in terms of the `C_UNIT`. The *par_value* can be a single *float* or a triplet *float*:*float*:*float*.
***S** *par_value par_value* [*threshold threshold*]	This construct defines the shape of the waveform on the logical or physical port in question. The first *par_value* is the rising slew in terms of the `T_UNIT`, while the second *par_value* is the falling slew. The threshold specification is optional and overrides for this port the default specified by the optional `SLEW_THRESHOLDS` value for `DESIGN_FLOW`. The first *threshold* is the percentage expressed as a *pos_fraction* of the input voltage defining the low input threshold, and the second is the percentage defining the high input threshold. As in capacitive loads or any other numeric value in this *SPEF_file*, a *par_value* can be a single *float* or a triplet *float*:*float*:*float*. Similarly, a *threshold* can be a single *pos_fraction* or a triplet.
***D** *cell_type*	The *cell_type* is a *name* or an *index* to a name defining the type of the driving cell. The case of the driving cell type not being known is designated by the reserved cell type `UNKNOWN_DRIVER`, which is also the default value when `*D` is absent. The application shall be responsible for determining what action to take when it sees `UNKNOWN_DRIVER`, whether or not the library has a cell by this reserved name. For example, an approximation of the output characteristics of an unknown driving cell type might be inferred from `*S` information for the port.

x) *physical_port_def* ::= ***PHYSICAL_PORTS** pport_entry {pport_entry}

This defines the physical ports for the *SPEF_file*. Information shown in the ***PHYSICAL_PORTS** section shall be consistent with that in the ***CONN** section of applicable *d_pnets*; `conn` attributes shown in the two sections are cumulative. If an application determines required information is missing from both sections, the application shall issue an error message and exit. All pnets connected to ports shall be in detailed form (*d_pnet)* in order to provide for multiple *SPEF_files*; *r_pnets* cannot connect to ports.

Each *pport_entry* is defined as

port_entry ::= pport_name direction {conn_attr}

The *pport_name* is defined as

pport_name ::= [<physical_inst><pin_delim>]<pport>

physical_inst is a *physical_ref* or an *index* to a physical_ref denoting the physical instance of the entity owning the pport, *pin_delim* is the *hchar* defined by ***DELIMITER** and *pport* is a *name* or an *index* to a name denoting the port of the physical entity. In a top-level *SPEF_file*, there is usually no *physical_inst* and *pin_delim* (e.g., `*PHYSICAL_PORTS “My_Design”/”main power”`).

The *direction* shall be **I**, **B**, or **O** (input, bidirectional, or output, respectively).

All *conn_attr* are optional. They are the same as shown in Table 9-2.

y) define_def ::= define_entry {define_entry}

This optional section specifies entity instances within the current *SPEF_file* which are actually parasitic *SPEF_file* partitions described in one or more other *SPEF_file*(s). The application thus knows to read and merge the multiple *SPEF_file*s in order to find the parasitics for the overall design. Example applications include package parasitics added around a chip design, pre-routed blocks being included in a design, dividing a large design into regions for parasitic extraction, or partitioning the design into sections which are at different stages of the design, floorplanning, or physical design process.

Nesting is allowed. Each parent *SPEF_file* shall be read before its child *SPEF_file*(s). The application shall know separately the searchpaths and *SPEF_file* names for the multiple *SPEF_file*s.

There shall be a separate define_entry for each child *SPEF_file* referenced in the current parent *SPEF_file* of the form:

define_entry ::= ***DEFINE** inst_name {inst_name} entity
| ***PDEFINE** physical_inst entity

1) An *entity* is a *qstring* whose value shall correspond to the *qstring* for ***DESIGN** in the child *SPEF_file*. Logical nets within a physical partition may connect to physical ports, since those physical ports may not exist in the logical netlist. pnets can also be connected to physical ports.

   i) If the *entity* follows logical hierarchy, the ***DEFINE** keyword shall be used and each corresponding *inst_name* is a *path* or an *index* to a path. If the logical *entity* does not contain any physical objects, the *entity* may be instantiated more than once (have more than one *inst_name*), such as a pre-routed block instantiated twice in a floorplanned design. A logical entity may contain physical objects (e.g., *pnets and pports)*. If the child *SPEF_file* for a logical entity contains any physical objects (e.g., *pnets* or *pports*), all *paths* and *physical_ref*s in the child *SPEF_file* shall be relative to the top of the complete design (the `NAME_SCOPE` for the child *SPEF_file* shall be `FLAT`) and only one *inst_name* is allowed. It shall be a semantic error if physical ports for a logical entity are connected to logical nets; *pports* for a logical entity can only be connected to *pnets*. If the child *SPEF_file* represents a logical partition, logical net connections from the parent to the child partition in the parent *SPEF_file* shall be instance pins, and it shall be a semantic violation if they are pnodes. Also, logical net connections to the parent in the child *SPEF_file* shall be logical ports, and it shall be a semantic violation if they are physical ports. This allows, but does not require, correspondence between the logical and physical hierarchy for the boundary between the two files.

   ii) If the child *entity* is a physical partition not following logical hierarchy, then the ***PDEFINE** keyword shall be used, the corresponding *physical_inst* is a *physical_ref* or an *index* to a *physical_ref*, and the `NAME_SCOPE` of the child *SPEF_file* shall be `FLAT`. Logical net connections from the parent to the physical child partition in the parent *SPEF_file* may be either instance pins or pnodes. Likewise, logical net connections from the child to the parent in the child *SPEF_file* may be either logical or physical ports. This allows, but does not require, correspondence between the logical and physical hierarchy for any portion of the boundary between the two files.

2) A *pnet* within a *SPEF_file* may only connect to physical ports; it shall be a semantic error if a *pnet* is connected to a logical port. However, a *pnet* may connect to both pins and pnodes for logical and physical objects, including child SPEF entities.

   It shall be a semantic error if a logical net in one *SPEF_file* is connected to a *pnet* in the other; i.e., nets being merged across the boundary between *SPEF_file*s shall be of a consistent type, either logical or physical. Because reduction cannot be performed until all parasitics for a *net* or *pnet* are known, any logical net crossing the boundary between parent and child *SPEF_file*s

shall be in *d_net* form in both files, and any *pnet* crossing the boundary between parent and child *SPEF_file*s shall be in *d_pnet* form in both files. Any *net* or *pnet* crossing the boundary between parent and child *SPEF_file*s which is in *r_net* or *r_pnet* form shall constitute a semantic error; *r_nets* and *r_pnets* are not allowed to connect to logical or physical ports, or to pins or pnodes of child SPEF entities.

When merging *SPEF_file*s, it shall be a semantic error if the child *SPEF_file* logical or physical port and parent *SPEF_file* instance pin or pnode *directions* do not correspond; e.g., a connection between the parent and child cannot be called an output in both files or an input in both files, but a bidirectional in one *SPEF_file* may be connected to an input, output, or bidirectional in the other *SPEF_file*.

3) *total_cap* shall be recalculated by the SPEF reader for *nets* and *pnets* crossing the boundary (in accordance with the `PIN_CAP` calculation method designated in the top-level *SPEF_file*), and *conn_attr* values for *nets* and *pnets* crossing the boundary removed or adjusted as appropriate to reflect updated information from merging the files. Mapping for nets being merged, and for nets, pins, and instances within the child *SPEF_file*(s) shall be updated to reflect the overall design logical hierarchy. As the child *SPEF_file* is read, the `unit_def` section for `par_value` entries and naming conventions (including delimiters) shall be adjusted to those of the parent. Routing confidence entries shall be reconciled; nets crossing a boundary normally are demoted to the lower routing confidence value between the parent and child *SPEF_file*s. `*DESIGN_FLOW` values shall be adjusted in the merged *SPEF_file* as appropriate; e.g., if either the parent or child has `MISSING_NETS`, then the merged *SPEF_file* also has `MISSING_NETS`.

It is the responsibility of the application to determine and adjust parasitics for over the cell and over the block routing as applicable when merging multiple *SPEF_file*s.

z) *internal_def* ::= nets {nets}

The *internal_def* section is the main part of the *SPEF_file* and contains the parasitic representation for the nets in the *SPEF_file*. The two types corresponding to the logical netlist format are *d_net* (distributed nets) and *r_net* (reduced nets). The two types of physical nets having no correspondence to the logical design netlist, such as power nets in some netlist formats, are *d_pnet* (distributed physical nets) and *r_pnet* (reduced physical nets). All four types of nets can be in an *SPEF_file* in any order. Applications that only are concerned with the logical function of a design and do not utilize physical net information may ignore physical nets. It shall be a semantic error if any given net is defined more than once in the same *SPEF_file* or group of *SPEF_file*s associated by *define_def*. If a net can be referenced both logically and physically, it is recommended to use the logical name. Within a net definition, if an object can be referenced both logically and physically, it is recommended to use the logical name.

aa) *d_net* ::= ***D_NET** net_ref total_cap
[routing_conf] [conn_sec] [cap_sec] [res_sec] [induc_sec] ***END**

A *d_net* contains distributed parasitic information for a logical net. The parasitic network may be derived from an estimation, global route, extraction, or some other source, or it may be a partial reduction (using *AWE* or some other means) of a more detailed parasitic network. If the parasitic values are small enough to yield an insignificant RC delay, the *d_net* can be simplified to a lumped capacitance form.

1) The *net_ref* can either be a *path* or an *index* to a path. The *total_cap* is a par_value and is simply the total of all capacitances on the net, not an equivalent capacitance, and also includes cross-coupling capacitances and external loads (from the ***CONN** and/or ***PORTS** sections). Whether it includes pin capacitances is determined by the `*DESIGN_FLOW` value for `PIN_CAP`. Cross-coupling capacitances are assumed to be to ground for this calculation. The *total_cap* is a simple lumped capacitance if there is no *cap_sec*.

2) The *routing_conf* allows a tool to record a confidence factor specifying the accuracy of the parasitics for the net. This *routing_conf* field allows different nets in a *SPEF_file* to have different levels of accuracy, such as when some nets in a design have been extracted while others were estimated. The default parasitic confidence value for the *SPEF_file* can be set in the `DESIGN_FLOW` statement by use of the `ROUTING_CONFIDENCE` construct.

The *routing_conf* is optional for both `R_NET` and `D_NET` and follows *total_cap*

routing_conf ::= ***V** conf
conf ::= pos_integer

Predefined values for *conf* are

10	Statistical wire load model
20	Physical wire load model
30	Physical partitions with locations, no cell placement
40	Estimated cell placement with steiner tree based route
50	Estimated cell placement with global route
60	Final cell placement with Steiner route
70	Final cell placement with global route
80	Final cell placement, final route, 2d extraction
90	Final cell placement, final route, 2.5d extraction
100	Final cell placement, final route, 3d extraction

If a design flow requires one or more values for *conf* other than provided in the predefined set, special values can be defined it in the `DESIGN_FLOW` construct using the `ROUTING_CONFIDENCE_ENTRY` value. Room is provided between predefined *conf* values so that a new `ROUTING_CONFIDENCE_ENTRY` *conf* value can be consistent with predefined values; e.g., a new *conf* value of `25` can designate an estimate whose accuracy is better than physical wire load models but not as good as physical partitions with locations, less cell placement.

3) *conn_sec* is defined as

conn_sec ::= ***CONN** conn_def {conn_def} {internal_node_coord}
conn_def ::= ***P** external_connection direction {conn_attr}
| ***I** internal_connection direction {conn_attr}
external_connection ::= port_name | pport_name
internal_connection ::= pin_name | pnode_ref
internal_node_coord ::= ***N** internal_node_name coordinates

This section defines the connections on a net. A connection begins with a ***P** if it is external (a *port* or *pport*), and with a ***I** if it is internal (a pin of a logical instance or a pnode of a physical object). The *d_net* can be connected to a physical port (*pport*) only if the current *SPEF_file* describes a child physical partition; otherwise it can only connect to logical ports. Similarly, the *d_net* can be connected to a *pnode* only if the *SPEF_file* describes a parent with one or more physical partition child *SPEF_files*; otherwise it can only connect to logical pins. If the optional ***CONN** section is missing, an application which requires all connectivity information to be present in the *SPEF_file* shall issue an error message and exit.

i) The *port_name* is defined as

[<inst_name><pin_delim>]<port>

where *inst_name* is a *path* or an *index* to a path denoting the instance of the logical entity owning the port, *pin_delim* is the *hchar* defined by ***DELIMITER**, and *port* is a *bit_identifier* or an *index* to a bit_identifier denoting the name of the scalar or bus bit port of the logical entity. In a top-level *SPEF_file*, there is usually no *inst_name and pin_delim.*

ii) The *pport_name* is defined as

<physical_inst><pin_delim><pport>

where *physical_inst* is a *physical_ref* or an *index* to a physical_ref, relative to the top of the design, denoting the physical instance of the current SPEF physical partition *SPEF_file* owning the port, *pin_delim* is the *hchar* defined by ***DELIMITER**, and *pport* is a *name* or an *index* to a name for the port. It shall be a semantic error for a *d_net* in a top-level *SPEF_file* or in a logical partition child *SPEF_file* to be connected to a *pport*.

iii) The *pin_name* is defined as

<inst_name><pin_delim><pin>

where *inst_name* is a *path* or an *index* to a path denoting the logical instance of a cell type or entity, *pin_delim* is the *hchar* defined by ***DELIMITER**, and *pin* is a *bit_identifier* or an *index* to a bit_identifier denoting the name of the scalar or bus pin of the cell type or entity.

iv) The *pnode_ref* is defined as

<physical_inst><pin_delim><pnode>

where *physical_inst* is a *physical_ref* or an *index* to a physical_ref, relative to the top of the design, denoting the physical instance of the child SPEF physical partition *SPEF_file* owning the pnode, and *pnode* is a *name* or an *index* to a name denoting the physical node of the child physical partition *SPEF_file*. It shall be a semantic error for a *d_net* to be connected to a pnode of a logical partition child *SPEF_file*.

v) The *direction* shall be **I**, **B**, or **O** (input, bidirectional, or output, respectively).

vi) The *conn_attr* definitions are the same as those for a *port*.

vii) The optional *internal_node_coord* enables coordinates to be specified for internal nodes, just as they can be for other items listed in the *node_name* (described below in part 4, *cap_sec)*.

internal_node_coord ::= ***N** internal_node_name coordinates

The *internal_node_name* is defined as

internal_node_name ::= <net_ref><pin_delim><pos_integer>

Information shown in the ***CONN** section shall be consistent with that in the ***PORTS** section; `conn` attributes shown in the two sections are cumulative. If an application determines required information is missing from both sections, the application shall issue an error message and exit.

4) The *cap_sec* is defined as

cap_sec ::= ***CAP** cap_elm {cap_elm}
cap_elm ::= cap_id node_name par_value | cap_id node_name node_name2 par_value

In the first *cap_elem* definition, the capacitance is assumed to be between *node_name* and `ground`. The second definition is typically used for cross-coupling capacitance. A cross-coupling *cap_elem* shall appear in the ***CAP** sections for both nets to which it is connected, whether they are *d_nets*, *d_pnets*, or a mixture, and the value shall be the same in both locations.

i) The *cap_id* is a pos_integer used to uniquely identify the capacitor. Because the *cap_id* is unique within the scope of the current net, the same *cap_id* can be repeated in another net without collision.

ii) The *node_name* can be one of the following:

node_name ::=
external_connection | internal_connection | internal_node_name | pnode_ref

The first two definitions for *node_name* specify connections to external ports and objects internal to the *SPEF_file*, respectively, with the same restrictions for physical ports and pnodes as described earlier for the ***CONN** section. The third definition is used to specify internal nodes or junction points on the current logical net. The fourth definition is used to specify physical pins on a physical partition that have no logical correspondence or correspondence to physical partition child *SPEF_files*, such as cluster nodes from a companion PDEF file.

iii) *node_name2* can be one of the following:

node_name2 ::= node_name
| <pnet_ref><pin_delim><pos_integer>
| <net_ref2><pin_delim><pos_integer>
net_ref2 ::= net_ref

The first definition is the same as above in part ii, *node_name*. The second definition describes an internal node or junction point on a pnet connected to the other end of a coupling capacitor. The third definition is used to specify internal nodes or junction points on a logical net other than the current one connected to the other end of a coupling capacitor.

iv) The *par_value* is specified in units of capacitance defined in the `C_UNIT` definition.

5) The *res_sec* is defined as

res_sec ::= ***RES** res_elem {res_elem}
res_elem ::= res_id node_name node_name par_value

i) The *res_id* is a pos_integer used to uniquely identify the resistor. Because it is unique within the scope of the current net, the same *res_id* can be repeated in another net without collision.

ii) The *node_name* has the same definition as shown above in part 4, *cap_sec*. The *par_value* is specified in units of resistance defined in the `R_UNIT` definition.

6) The *induc_sec* is defined as follows:

induc_sec ::= ***INDUC** induc_elem {induc_elem}
induc_elem ::= induc_id node_name node_name par_value

The *induc_id* is a pos_integer used to uniquely identify the inductor. Because it is unique within the scope of the current net, the same *induc_id* can be repeated in another net without collision. The *node_name* has the same definition as in the *cap_sec*. The *par_value* is specified in units of inductance defined in the `L_UNIT` definition.

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

An *r_net* is a net which has been reduced from a distributed model to an electrical equivalent through *AWE* or some other similar method. Because all parasitics for a net shall be known before reduction, a portion of a net which crosses the boundary between a parent and child *SPEF_file* cannot be in *r_net* form and an *r_net* cannot connect to a logical or physical port or to a pnode or pin of a child *SPEF_file*. If the parasitic values are small enough to yield an insignificant RC delay, the *r_net* can be simplified to a lumped capacitance form.

There shall be one *driver_reduc* section for each driver that a net has. If a net has four different drivers, then there shall be four different *driver_reduc* sections in the *r_net* definition.

1) The *net_ref* can either be a *path* or an *index* to a path. The *total_cap* is a *par_value* and is simply the total of all capacitances on the net, not an equivalent capacitance, and also includes cross-coupling capacitances. Whether it includes pin capacitances is determined by the `*DESIGN_FLOW` value for `PIN_CAP`. Cross-coupling capacitances are assumed to be to ground for this calculation.

2) The *total_cap* is a simple lumped capacitance if there is no *driver_reduc*.

3) The *routing_conf* is defined the same as it was for a *d_net*.

4) The *driver_reduc* is defined as

   driver_reduc ::= driver_pair driver_cell pie_model load_desc

   i) driver_pair ::= ***DRIVER** pin_name

      This statement specifies the driver to which the net reduction was done. The *pin_name* is defined as

      pin_name ::= <inst_name><pin_delim><pin>

      where *inst_name* is a *path* or an *index* to a path denoting the instance of the driving cell, *pin_delim* is the *hchar* defined by ***DELIMITER** and *pin* is a *bit_identifier* or an *index* to a *bit_identifier* denoting the name of the scalar or bus pin of the cell type for the instance.

   ii) driver_cell ::= ***CELL** cell_type

      The *cell_type* is a *name* or an *index* to a name which gives the cell type of the driving cell. Because an *r_net* cannot be connected to a port, **UNKNOWN_DRIVER** is not allowed as the type of driving cell.

      `conn` attributes cannot be specified in *r_nets*. If the optional *driver_reduc* section is missing, an application which requires all connectivity information to be present in the *SPEF_file* shall issue an error message and exit.

   iii) The *pie_model* is defined as

      pie_model ::= ***C2_R1_C1** par_value par_value par_value

      The *pie_model* is an electrical description of the admittance model seen by the driving cell. The first *par_value*, `C2`, is the capacitor closest to driving cell. The third *par_value*, `C1`, is the capacitor furthest away from the driving cell. They are connected by the resistor `R1`, which is the second *par_value* in the above description.

   iv) The *load_desc* is defined as

      load_desc ::= ***LOADS** rc_desc {rc_desc}

There shall be an *rc_desc* for each load or input connection on a net. The *rc_desc* can contain a single pole Elmore delay (given as *par_value*) or a single pole Elmore delay followed by additional poles and residues.

rc_desc ::= ***RC** pin_name par_value [pole_residue_desc]
pole_residue_desc ::= pole_desc residue_desc

The *pin_name* is as described above in part i, *driver_pair*. A *pole_desc* is defined as

pole_desc ::= ***Q** pos_integer pole {pole}
pole ::= complex_par_value

The *pos_integer* specifies the number of poles to be defined. The *complex_par_value* is just like a *par_value* except the numbers can either be a *number* or a *cnumber*.

cnumber is defined as

cnumber ::= **(**number number**)**

A *cnumber* is a representation for a complex number. The first number is the real component and the second number is the imaginary component. The parenthesis are part of the syntax.

A *residue_desc* is defined as

residue_desc ::= ***K** pos_integer residue {residue}

The *pos_integer* specifies the number of residues to be defined. The *complex_par_value* is just like a *par_value* except the numbers can either be a *number* or a *cnumber*.

ac) *d_pnet* ::= ***D_PNET** pnet_ref total_cap
[routing_conf] [pconn_sec] [pcap_sec] [pres_sec] [pinduc_sec] ***END**

A *d_pnet* contains distributed parasitic information for a physical net (*pnet*). The parasitic network may be derived from an estimation, global route, extraction, or some other source, or it may be a partial reduction (using *AWE* or some other means) of a more detailed parasitic network. If the parasitic values are small enough to yield an insignificant RC delay, the *d_pnet* can be simplified to a lumped capacitance form.

1) The *pnet_ref* can either be a *physical_ref* or an *index* to a physical_ref. The *total_cap* is a *par_value* and is simply the total of all capacitances on the pnet, not an equivalent capacitance, and also includes cross-coupling capacitances and external loads (from the ***CONN** and/or ***PHYSICAL_PORTS** sections). Whether it includes pin capacitances is determined by the `*DESIGN_FLOW` value for `PIN_CAP`. Cross-coupling capacitances are assumed to be to `ground` for this calculation.

2) The *total_cap* is a simple lumped capacitance if there is no *pcap_sec*.

3) The *routing_conf* is defined the same as for *d_net*, except applying to a *pnet* rather than a logical net.

4) *pconn_sec* is defined as

pconn_sec ::= ***CONN** pconn_def {pconn_def} {internal_pnode_coord}

pconn_def ::= ***P** pexternal_connection direction {conn_attr}
| ***I** internal_connection direction {conn_attr}

pexternal_connection ::= pport_name
internal_connection ::= pin_name | pnode_ref
internal_pnode_coord ::= ***N** internal_pnode_name coordinates

This section defines the connections on a *pnet*. The connection begins with a ***P** if it is external (a pport), and with a ***I** if it is internal (a pin of a logical instance or a pnode of a physical object). The *d_pnet* can be connected only to physical ports (*pport*); it shall be a semantic violation if connected to a logical port. Internally, the *d_pnet* can be connected to either a pnode or a pin of a logical instance.

i) The *pport_name* is defined as

<physical_inst><pin_delim><pport>

where *physical_inst* is a *physical_ref* or an *index* to a *physical_ref*, relative to the top of the design, denoting the physical instance of the current SPEF physical partition *SPEF_file* owning the port, *pin_delim* is the *hchar* defined by ***DELIMITER**, and *pport* is a *name* or an *index* to a name for the port. It shall be a semantic error for a *d_pnet* to be connected to a logical port.

ii) The *pin_name* is defined as

<inst_name><pin_delim><pin>

where *inst_name* is a *path* or an *index* to a path denoting the logical instance of a cell type or entity, *pin_delim* is the *hchar* defined by ***DELIMITER**, and *pin* is a *bit_identifier* or an *index* to a bit_identifier denoting the name of the scalar or bus pin of the cell type or entity.

iii) The *pnode_ref* is defined as

<physical_inst><pin_delim><pnode>

where *physical_inst* is a *physical_ref* or an *index* to a *physical_ref*, relative to the top of the design, denoting the physical instance of a physical object within the *SPEF_file* owning the pnode, and *pnode* is a *name* or an *index* to a name denoting the physical node of the physical object.

iv) The *direction* shall be **I**, **B**, or **O** (input, bidirectional, or output, respectively).

v) The *conn_attr* definitions are the same as those for a *port*.

vi) The optional *internal_pnode_coord* enables coordinates to be specified for internal nodes, just as they can be for other items listed in the *pnode_name* described below in part 5, *pcap_sec*.

internal_pnode_coord ::= ***N** internal_pnode_name coordinates

The *internal_pnode_name* is defined as

<pnet_ref><pin_delim><pos_integer>

Information shown in the ***CONN** section shall be consistent with that in the ***PHYSICAL_PORTS** section; `conn` attributes shown in the two sections are cumulative. If an application determines required information is missing from both sections, the application shall issue an error message and exit.

5) The *pcap_sec* is defined as

pcap_sec ::= ***CAP** pcap_elm {pcap_elem}

pcap_elem ::= cap_id pnode_name par_value
| cap_id pnode_name pnode_name2 par_value

In the first *pcap_elem* definition, the capacitance is assumed to be between *pnode_name* and `ground`. The second definition is typically used for cross-coupling capacitance. A cross-coupling *pcap_elem* shall appear in the ***CAP** sections for both nets to which it is connected,