IEEE Standard for Design and Verification of Low Power Integrated Circuits

IEEE Computer Society

Sponsored by the
Design Automation Standards Committee

and the

IEEE Standards Association Corporate Advisory Group

1801™

IEEE
3 Park Avenue
New York, NY 10016-5997, USA

27 March 2009

IEEE Std 1801™-2009

IEEE Standard for Design and Verification of Low Power Integrated Circuits

Sponsor

Design Automation Standards Committee
of the
IEEE Computer Society
and the
IEEE Standards Association Corporate Advisory Group

Approved 19 March 2009

IEEE-SA Standards Board

Grateful acknowledgment is made to Accellera, Inc. for the permission to use the following source material:
Unified Power Format (UPF) Standard, Version 1.0

Abstract: The power supplied to elements in an electronic design affects the way circuits operate. Although this is obvious when stated, today's set of high-level design languages have not had a consistent way to concisely represent the regions of a design with different power provisions, nor the states of those regions or domains. This standard provides an HDL-independent way of annotating a design with power intent. In addition, the level-shifting and isolation between power domains may be described for a specific implementation, from high-level constraints to particular configurations. When the logic in a power domain receives different power supply levels, the logic state of portions of the design may be preserved with various state-retention strategies. This standard provides mechanisms for the refined and specific description of intent, effect, and implementation of various retention strategies. Incorporating components into designs is greatly assisted by the encapsulation and specification of the characteristics of the power environment of the design and the power requirements and capabilities of the components; this information encapsulation mechanism is also described in this standard. The analysis of the various power modes of a design is enabled with a combination of the description of the power modes and the collection, generation, and propagation of switching information.
Keywords: corruption semantics, interface specification, IP reuse, isolation, level-shifting, power-aware design, power intent, power domains, power modes, power states, progressive design refinement, retention, retention strategies

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

PDF: ISBN 978-0-7381-5929-4 STD95919
Print: ISBN 978-0-7381-5930-0 STDPD95919

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

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

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

The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

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

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration. A statement, written or oral, that is not processed in accordance with the IEEE-SA Standards Board Operations Manual shall not be considered the official position of IEEE or any of its committees and shall not be considered to be, nor be relied upon as, a formal interpretation of the IEEE. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that his or her views should be considered the personal views of that individual rather than the formal position, explanation, or interpretation of the IEEE.

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

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

Introduction

This introduction is not part of IEEE Std 1801-2009, IEEE Standard for Design and Verification of Low Power Integrated Circuits.

The purpose of this standard is to provide portable low power design specifications that can be used with a variety of commercial products throughout an electronic system design, analysis, verification, and implementation flow.

When the electronic design automation (EDA) industry began creating standards for use in specifying, simulating and implementing functional specifications of digital electronic circuits in the 1980s, the primary design constraint was the transistor area necessary to implement the required functionality in the prevailing process technology at that time. Power considerations were simple and easily assumed for the design as power consumption was not a major consideration and most chips operated on a single voltage for all functionality. Therefore, hardware description languages (HDLs) such as VHDL (IEC/IEEE 61691-1-1)[a] and Verilog (IEEE Std 1364™) [B2][b] provided a rich set of capabilities necessary for capturing the functional specification of electronic systems, but no capabilities for capturing the power architecture (how each element of the system is to be powered).

As the process technology for manufacturing electronic circuits continued to advance, power (as a design constraint) continually increased in importance. Even above the 90–100 nm process node size, dynamic power consumption became an important design constraint as the functional size of designs increased power consumption at the same time battery-operated mobile systems, such as cell phones and laptop computers, became a significant driver of the electronics industry. Techniques for reducing dynamic power consumption—the amount of power consumed to transition a node from a `0` to `1` state or vice versa—became commonplace. Although these techniques affected the design methodology, the changes were relatively easy to accommodate within the existing HDL-based design flow, as these techniques were primarily focused on managing the clocking for the design (more clock domains operating at different frequencies and gating of clocks when logic in a clock domain is not needed for the active operational mode). Multi-voltage power management methods were also developed. These methods did not directly impact the functionality of the design, requiring only level-shifters between different voltage domains. Multi-voltage power domains could be verified in existing design flows with additional, straight-forward extensions to the methodology.

With process technologies below 100 nm, static power consumption has become a prominent and, in many cases, dominant design constraint. Due to the physics of the smaller process nodes, power is leaked from transistors even when the circuitry is quiescent (no toggling of nodes from `0` to `1` or vice versa). New design techniques were developed to manage static power consumption. Power gating or power shut-off turns off power for a set of logic elements. Back bias techniques are used to raise the voltage threshold at which a transistor can change its state. While back bias slows the performance of the transistor, it greatly reduces leakage. These techniques are often combined with multi-voltages and require additional functionality: power management controllers, isolation cells that logically and/or electrically isolate a shutdown power domain from “powered-up” domains, level-shifters that translate signal voltages from one domain to another, and retention registers to facilitate fast transition from a power-off state to a power-on state for a domain.

The EDA industry responded with multiple vendors developing proprietary low power specification capabilities for different tools in the design and implementation flow. Although this solved the problem locally for a given tool, it was not a global solution in that the same information was often required to be specified multiple times for different tools without portability of the power specification. At the Design

[a]Information on references can be found in Clause 2.

[b]The number in brackets correspond to those of the bibliography in Annex A.

Automation Conference (DAC) in June 2006, several semiconductor/electronics companies challenged the EDA industry to define an open, portable power specification standard. The EDA industry standards incubation consortium, Accellera, answered the call by creating a Technical SubCommittee (TSC) to develop a standard. The effort was named Unified Power Format (UPF) to recognize the need of unifying the capabilities of multiple proprietary formats into a single industry standard. Accellera approved *UPF 1.0* as an Accellera standard in February 2007. In May 2007, Accellera donated UPF to the IEEE for the purposes of creating an IEEE standard. The donation was executed to the P1801 working group and, although this standard is the first version of what is formally titled the IEEE Standard for the Design and Verification of Low Power Integrated Circuits, it represents the second version of what is more colloquially referred to as UPF.

Notice to users

Laws and regulations

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

Copyrights

Updating of IEEE documents

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

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

Errata

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

Interpretations

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

Patents

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

Participants

The Unified Power Format Working Group is entity based. At the time this standard was completed, the Unified Power Format Working Group had the following membership:

Stephen Bailey, *Chair*
Gary Delp, *Vice-Chair*
Joe Daniels, *Technical Editor*

Karen Bartleson
Frank Berntsen
Jason Binney
John Biggs
Minh Chau
Marc Edwards
Ed Huijbregts
Knut Just
Juergen Karmann
Kevin Kranen
Rolf Lagerquist
Olivier Lunven
Lisa McIlwain
Arvind Narayanan
Judith Richardson
Michael Rifani
Arturo Salz
Andrew Saunders
Eike Schmidt
Jim Sproch
Yatin Trivedi

The following members of the entity balloting committee voted on this standard. Balloters may have voted for approval, disapproval, or abstention.

ARM
Accellera
IBM
Improv Systems
Intel
JEITA
LSI Corporation
Magma Design Automation
Mentor Graphics
NXP
Synopsys
Texas Instruments

When the IEEE-SA Standards Board approved this standard on 19 March 2009, it had the following membership:

Robert M. Grow, *Chair*

Steve M. Mills, *Past Chair*

Judith Gorman, *Secretary*

John Barr
Karen Bartleson
Victor Berman
Ted Burse
Richard DeBlasio
Andy Drozd
Mark Epstein
Alexander Gelman
Jim Hughes
Rich Hulett
Young Kyun Kim
Joseph L. Koepfinger*
John Kulick
David Law
Ted Olsen
Glenn Parsons
Ron Petersen
Chuck Powers
Thomas Prevost
Narayanan Ramachandran
Jon Rosdahl
Sam Sciacca

*Member Emeritus

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

Satish K. Aggarwal, *NRC Representative*
Michael Janezic, *NIST Representative*
Howard Wolfman, *TAB Representative*

Michelle Turner
IEEE Standards Program Manager, Document Development

Michael D. Kipness
IEEE Standards Program Manager, Technical Program Development

Noelle Humenick
IEEE Standards Corporate Client Manager

Contents

IEEE Standard for Design and Verification of Low Power Integrated Circuits

1. Overview

1.1 Scope

This standard establishes a format used to define the low power design intent for electronic systems and electronic intellectual property. The format provides the ability to specify the supply network, switches, isolation, retention and other aspects relevant to power management of an electronic system. The standard defines the relationship between the low power design specification and the logic design specification captured via other formats (e.g., standard hardware description languages).

1.2 Purpose

The standard provides portability of low power design specifications that can be used with a variety of commercial products throughout an electronic system design, analysis, verification and implementation flow.

1.3 Key characteristics of the Unified Power Format (UPF)

The Unified Power Format (UPF) provides the ability for electronic systems to be designed with power as a key consideration early in the process. UPF accomplishes this by allowing the specification of what was traditionally physical implementation-based power information early in the design process—at the register

transfer level (RTL) or earlier. Figure 1 shows UPF supporting the entire design flow. UPF provides a consistent format to specify power design information that may not be easily specifiable in a hardware description language (HDL) or when it is undesirable to directly specify the power semantics in an HDL, as doing so would tie the logic specification directly to a constrained power implementation. UPF specifies a set of HDL attributes and HDL packages to facilitate the expression of power intent in HDL when appropriate (see Table 1 and Annex B). UPF also defines consistent semantics across verification and implementation, i.e., what is implemented is the same as what has been verified.

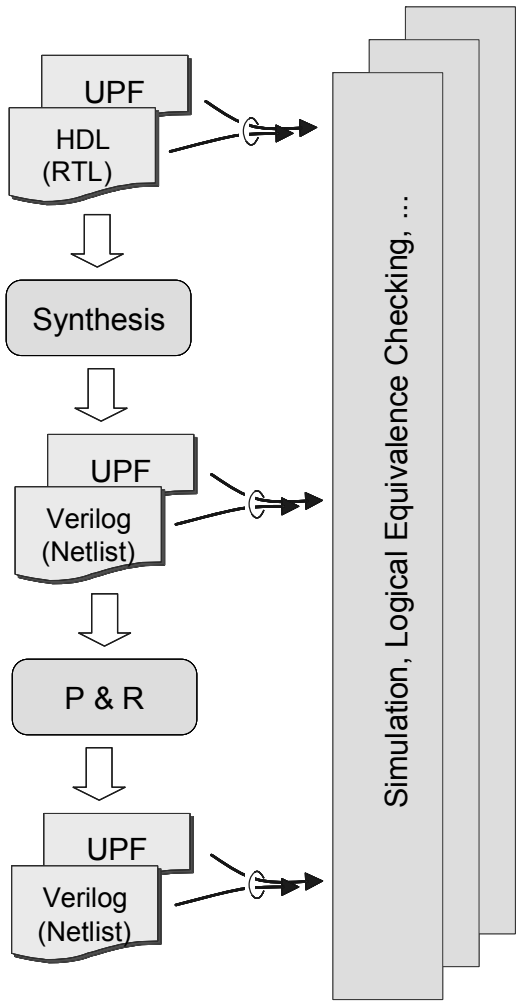

Figure 1—UPF tool flow

As indicated in Figure 1, UPF files are part of the design source. Combined with the HDL, the UPF files are used to describe the intent of the designer. This collection of source files is the input to several tools, e.g., simulation tools, synthesis tools, and formal verification tools.

— Synthesis tools can read the HDL/UPF design input files and produce a netlist. The tool or user may produce a new UPF fileset that, combined with the netlist, represents a further refined version of same design.

— In those cases where names change, a UPF file with the new names is needed. A UPF-aware logical equivalence checker can read the full design filesets and perform the checks including the results of the UPF commands to ensure equivalence.

— Place and Route tools read both the netlist and the UPF files and produce outputs, potentially including an output UPF file.

UPF is a concise power design intent specification capability. Power design intent can be easily specified over many elements in the design. A UPF specification can be included with the other deliverables of intellectual property (IP) blocks and reused along with the other delivered IP. UPF supports various methodologies through carefully defined semantics, flexibility in specification, and, when needed, defined rational limitations that facilitate automation in verification and implementation.

A UPF *specification* defines how to create a supply network to supply power to each design element, how the individual supply nets behave with respect to one another, and how the logic functionality is extended to support dynamic power switching to these logic design elements. By controlling the operating voltages of each supply net and whether the supply nets (and their connected design elements) are turned on or off, the supply network only provides power at the level the functional areas of the chip need to complete the computational task in a timely manner.

1.4 Power supply network design intent

Designing electronics to meet low power design constraints requires the specification of a power supply network that can control the distribution of that supply to minimize energy consumption. UPF supports the specification of the power supply distribution network so the supply network can be automatically implemented at a relatively abstract level.

To help manage the complexity of the supply network specification, *power domains* are defined to group elements from the logic hierarchy that share common supply needs. By default, all logic elements in a power domain use the same primary supply. Additional supplies may be defined to serve different uses in a power domain. In addition to the primary supply, UPF provides well-defined semantics for other supplies for both verification and implementation contexts.

The *supply network* consists of supply ports, switches, and supply nets. Supply network objects are defined within the logical hierarchy relative to the context of a power domain. *Supply ports* provide the supply interface to the logical hierarchy and the power domain's elements. Supply ports also provide the supply interface to switches. *Switches* control the supply distribution. *Supply nets* connect supply ports.

Although there is an obvious inference to the actual wires and ports in the implemented hardware, a UPF supply network is an abstraction of the electrical network on the chip. UPF defines no routing or layout information. As the supply network is specified apart from the logic design, the logic design specification remains independent of a specific power supply network specification.

Figure 2 shows an example supply distribution network for a hypothetical chip, `mySoC`. A top-level power domain, `mySoC_PD` defined at the `U_top` instance level in the logical hierarchy, is defined with three supply ports, `Pbat`, `Pwall`, and `GND`. These ports represent the off-chip power sources. A single switch (`S1`) controls whether the chip receives its supply from `Pbat` or `Pwall`. Supply nets connect the output from `S1` to each power domain defined within the top-level power domain. `mySoC_PD` does not contain any logic elements other than the root of the design `U_top`; its purpose is to define the interface to the off-chip power sources and provide the top-level supply network.

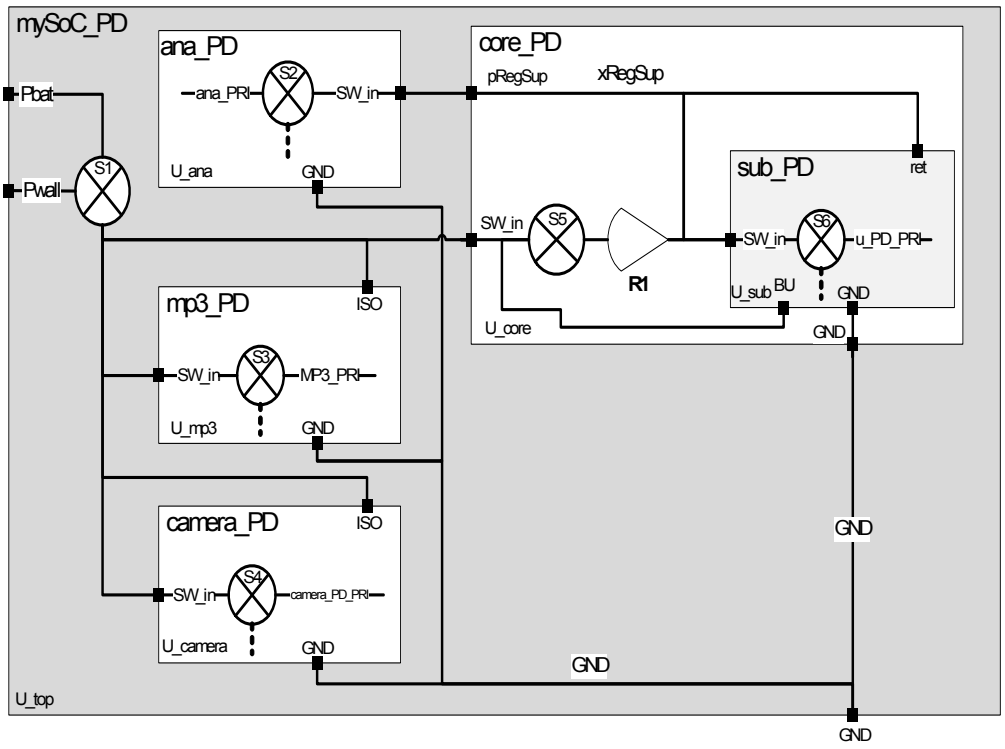

Figure 2—A supply distribution network

Four power domains are defined at the next level, `ana_PD`, `camera_PD`, `core_PD`, and `mp3_pd`. `camera_PD`, `mp3_PD`, and `ana_PD` each have a similar supply network internal to them. The primary supply is switched (`S4`, `S3`, and `S2`), which indicates power to the logic elements in the domains can be turned off. Unswitched isolation supplies are provided for `camera_PD` and `mp3_PD`.

`core_PD` contains a nested domain `sub_PD`. `core_PD`'s primary supply `xRegSup` is switched and regulated within the domain, as it is the output of a regulator that has a switch (`S5`) driving the regulator's input (`R1`). Besides being the primary supply for `core_PD`, `xRegSup` also provides a retention supply for `sub_PD`. The regulated supply is also connected to the output supply port (`pRegSup`) and, ultimately, the switch input (`S2`) whose output drives the primary supply for `ana_PD`. This indicates `ana_PD` is powered on only when `core_PD` is powered on. Within `sub_PD`, the primary supply is switched (`S6`), indicating `sub_PD` can be powered down separately from `core_PD`, but holding the relationship that if `core_PD` is powered down, so is `sub_PD`. `sub_PD` also contains two additional supplies (`ret` and `BU`). The `ret` supply (which is used as a retention supply) has the same source as `core_PD`'s primary supply (the output from switch `S5`) and the `BU` supply has the same source as the input to switch `S5`; therefore, the retention supply (`ret`) is valid only when `core_PD` is on (switch `S5` in the `core_PD` power domain). The `BU` supply is provided for any retention elements whose state needs to be saved when both `sub_PD` and `core_PD` are powered down.

This example demonstrates a methodology of locating the switches for a power domain's supplies within the power domain. UPF also supports placing the switches that can be switched off outside the power domain. If this methodology were being used, switches `S2`, `S3`, and `S4` would be located in `mySoC_PD` and additional supply nets to connect the outputs of the switches to the corresponding ports of the nested power domains' logical hierarchies would be created. Similarly, the switch `S5` and regulator `R1` would be placed in `mySoC_PD` and the switch `S6` moved up to `core_PD`. Either methodology is valid and the supply networks are equivalent.

UPF provides the flexibility to specify the network in whatever way makes sense for a given design or methodology, including what can be switched. In this example, only the power supplies are switched; however, switches can also be placed on grounds, bias nets, and other named supply functions. This provides complete flexibility in how the supply distribution is controlled.

A UPF supply network defines a directed acyclic graph (DAG) when inout supply ports are not used. Limiting the supply network to a DAG representation simplifies the evaluation of the supply network state. This simplification enables efficient verification of the logic design and supply network within a digital simulation environment.

The state of the supply network is simply the state of each control element (switch) in the network and the values of any root supply driver at any given point in time. Absolute relationships are easy to infer from the supply network specification (e.g., `core_PD` is always on when `sub_PD` and `ana_PD` are on). To capture situations that are never intended to occur, UPF can also define legal and illegal power states (see Clause 4). The named power states and the legality of a named or unnamed power state can be used to optimize the design implementation and to facilitate design verification.

1.5 Extending logic specification

UPF defines extensions of the logic design with power-specific capabilities and constraints without modifying the original logic specification. UPF provides designers with the guarantee they intuitively expect, i.e., by adding a UPF power specification to a logic design or by changing or replacing an existing UPF specification for a logic design, designers do not need to touch the original logic specification or re-verify the logic functionality independent of the power specification. The UPF standard also facilitates the reuse of the logic specification in contexts where an explicit power-design intent is not required or where the power-design intent can change from one implementation of the design to another. Figure 3 illustrates how UPF extends the logic design by adding power functionality while leaving the original functionality unmodified.

Figure 3 also demonstrates the addition of retention and isolation functionality to the logic design. Isolation is required to ensure undefined outputs from powered-down design elements do not drain power from those design elements that are not powered down. Isolation also ensures a specific logic value is driven from the power domain's outputs. Assuming the output `CNT` in Figure 3 can serve as an interrupt or reset indicator to another logic element, if the isolation of `CNT` did not take into account the active level of the interrupt or reset interpretation of its value, then powering down the domain containing this counter could result in undesired side-effects, e.g., logic in another power domain entering and remaining in a reset or interrupt-handling mode. UPF supports the specification of isolation strategies that provide information on the clamp values and location of the isolation logic.

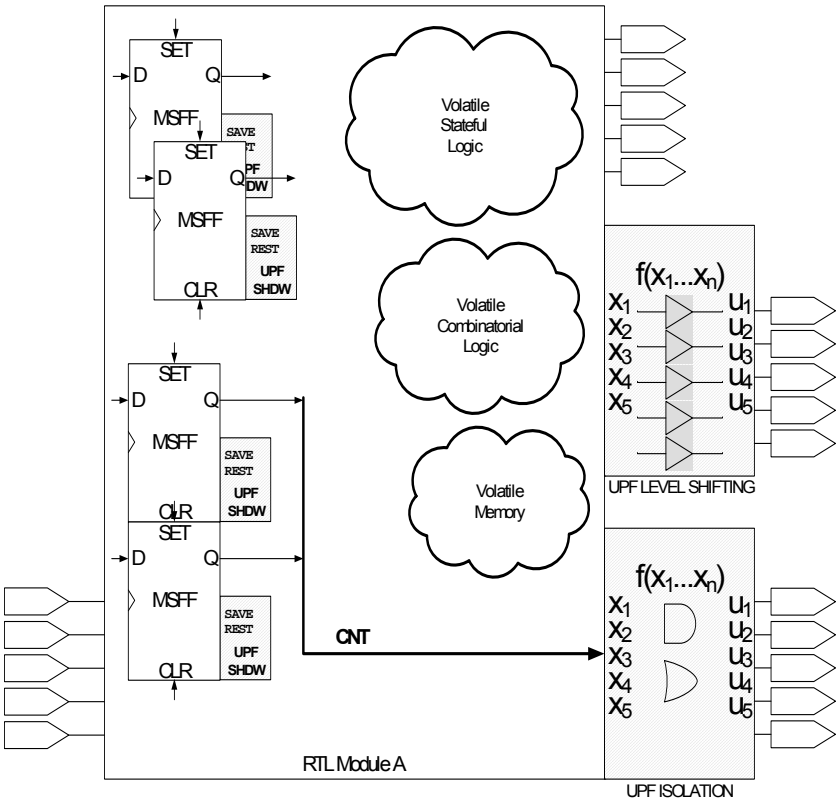

Figure 3—UPF extends the logic design

Retention is the ability to save the value of a sequential element in a power domain prior to switching off the power to that element and then later restoring its value after power has been enabled for the element. Behavioral and implementation retention semantics can be specified for sequential elements that require save and restore capabilities.

Isolation and retention both demonstrate UPF's concise specification characteristics and its flexibility in specification. A general strategy can be defined for both isolation and retention. This strategy is then applied to all ports or sequential elements governed by the scope of the strategy. Specific overriding of a general strategy allows for the management of exceptional situations. Flexibility is realized through the recognition that isolation and retention behavior, for any given design or implementation, may be more complex or require a different connectivity than the predefined general behaviors supported by UPF. In these situations, mapping of the isolation or retention to specific verification (functional behavior) and implementation models is supported.

1.6 Conventions used

Each clause that details any UPF commands defines it own conventions and meta-syntax as needed (see also Clause 6 through Clause 8).

1.7 Use of color in this standard

This standard uses a minimal amount of color to enhance readability. The coloring is not essential and does not affect the accuracy of this standard when viewed in pure black and white. The places where color is used are the following:

— Cross references that are hyperlinked to other portions of this standard are shown in underlined-blue text (hyperlinking works when this standard is viewed interactively as a PDF file).

— Syntactic keywords and tokens in the formal language definitions are shown in **boldface-red text**.

— Command arguments that can be layered are shown in **boldface-green text**. See also 6.4.

1.8 Contents of this standard

The organization of the remainder of this standard is as follows:

— Clause 2 provides references to other applicable standards that are presumed or required for this standard.

— Clause 3 defines terms and acronyms used throughout the different specifications contained in this standard.

— Clause 4 describes power domains and specifies the conventions for naming objects, their name space resolution, and conflict resolution.

— Clause 5 defines simulation semantics for various UPF commands.

— Clause 6 details the syntax and semantics for each UPF command.

— Clause 7 details the syntax and semantics for each UPF query and the **find_objects** command.

— Clause 8 describes the syntax and semantics of the Switching Activity Interchange Format (SAIF).

— Annexes. Following Clause 8 are a series of annexes.

NOTE—The normative portion of this document is organized in two major sections. The first, Clause 1 through Clause 7, defines the UPF capabilities for capturing low power design intent. The second, Clause 8, defines a format for capturing switching activity information, which can be used for analysis and estimating power consumption. While designing electronics to minimize power would benefit from tools supporting both sets of capabilities, the two sections hold no dependencies between one another.[1]

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

2. Normative references

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

IEC/IEEE 61691-1-1, Behavioural languages—Part 1: VHDL language reference manual.[2, 3]

IEEE Std 1800™, IEEE Standard for SystemVerilog Unified Hardware Design, Specification, and Verification Language.[4]

3. Definitions, acronyms, and abbreviations

For the purposes of this document, the following terms and definitions apply. *The Authoritative Dictionary of IEEE Standards Terms* [B1][5] should be referenced for terms not defined in this clause.

3.1 Definitions

3.1.1 anonymous object: An object that is not named in the context of UPF. Implementations may assign a legal name, but such names are not visible in the UPF context.

3.1.2 active scope: The **instance** specified by the **set_scope** command (see 6.50).

3.1.3 activity: Any change of the inputs of an **element**, regardless of whether there are any changes on the output.

3.1.4 ancestor: Any **instance** between the **active scope** in the **logic hierarchy** and its root, including the root. However, when the **active scope** is the root, it does not have any ancestors.

3.1.5 automatic supply net connection: A supply net connection created as a result of the association of a **supply net** of a **supply set** identified by the function the net performs for the set and the **pg_type** attribute on a **supply port**. *See also* **explicit supply net connection** and **implicit supply set connection**.

3.1.6 command: A Tcl procedure [B5] defined to specify UPF power design intent with one or more **design object**s as its target.

3.1.7 composite domain: A container comprising a set of **power domains** called **subdomains**. All **subdomains** in the composite domain share the same **primary supply set**. Any operation performed on the **composite domain** has the same effect as performing the operation on each of the **subdomains**.

3.1.8 connection: The attachment of a **port** to a **net**. *See also* **HighConn** and **LowConn**.

[2]IEC publications are available from the Sales Department of the International Electrotechnical Commission, Case Postale 131, 3, rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse (http://www.iec.ch/). IEC publications are also available in the United States from the Sales Department, American National Standards Institute, 25 West 43rd Street, 4th Floor, New York, NY 10036, USA (http://www.ansi.org/).

[3]IEEE publications are available from the Institute of Electrical and Electronics Engineers, Inc., 445 Hoes Lane, Piscataway, NJ 08854, USA (http://standards.ieee.org/).

[4]The IEEE standards or products referred to in this clause are trademarks of the Institute of Electrical and Electronics Engineers, Inc.

[5]The number in brackets correspond to those of the bibliography in Annex A.

3.1.9 corruption semantics: The rules defining the behavior of the logic in the context of the supply network specification, which might remove support of normal logical operation from time to time, as distinguished from the default always-on logical behavior.

3.1.10 design element: An **instance** of a SystemVerilog module (see IEEE Std 1800),[6] VHDL entity (see IEC/IEEE 61691-1-1), or library cell. The term design element is often abbreviated to **element**.

3.1.11 design object: A design object is any object that can be declared or created in the **logic hierarchy**. Design objects include (but are not limited to) wires, registers, switches, **port**s, **supply net**s, and **design element**s. The term design object is often abbreviated to **object**.

3.1.12 driver: Any **design element** that supplies a value to a **net**, either directly, or indirectly via a **port**. Ports are not drivers.

3.1.13 explicit supply net connection: A supply net connection that is explicitly specified in a **connect_supply_net** command (see 6.13). *See also* **automatic supply net connection** and **implicit supply net connection**.

3.1.14 extent: The set of **design element**s that comprise a **power domain**.

3.1.15 HighConn: This indicates the hierarchically higher (closer to the root) port connection.

3.1.16 hierarchical separator character: A special character used in composing hierarchical port, pin, net, or instance names from simple identifiers.

3.1.17 implicit supply set connection: The **supply net**s of a **supply set** are implicitly connected to those elements in the extent of a power domain, or those elements targeted by an isolation, retention, or level-shifter strategy, when no **supply port** exists on the element. *See also* **automatic supply net connection** and **explicit supply net connection**.

3.1.18 instance: A particular occurrence of a **design object**.

3.1.19 isolation: Techniques used to provide defined behavior of logic signals associated with independently switched supplies.

3.1.20 isolation cell: A **design element** that passes logic values during normal mode operation and clamps its output to some specified logic value when the control signal is asserted.

3.1.21 level-shifter: A **design element** that translates signal values from an input voltage swing to a different output voltage swing.

3.1.22 logic design: Design functionality defined using an HDL, such as SystemVerilog or VHDL.

3.1.23 logic hierarchy: The tree structure of **design element**s specified within a **logic design**.

3.1.24 LowConn: This indicates the lower (further from the root) port connection.

3.1.25 map: Substitute a specific **model** for an abstract behavior.

3.1.26 merged domain: A single **power domain** that is the union of the elements that were members of the power domains being merged. The domains that are merged into the merged domain are no longer visible.

[6]For information on references, see Clause 2.

3.1.27 model: A SystemVerilog module, VHDL entity/architecture, or Liberty cell.

3.1.28 named power state: A **power state** defined using **add_power_state** for a **supply set** or **power domain**, or the **DEFAULT_NORMAL** and **DEFAULT_CORRUPT** power states predefined for supply sets. *See also* **unnamed power state**.

3.1.29 net: The collection of interconnections between a collection of ports. A net may be named or anonymous. At a module boundary, a named net may be considered as having two parts, the **HighConn** and **LowConn** portion of the net relative to a particular module or cell boundary.

3.1.30 parent: The immediate **ancestor** of a given **instance**.

3.1.31 pg_type: Supply ports on components can be labeled with a "Power/Ground Type" name; e.g., the Liberty definition [B6] of `pg_type`.

3.1.32 port: A **connection** on the interface of a SystemVerilog module or VHDL entity.

3.1.33 power domain: A collection of **design element**s that share a **primary supply set**. A power domain may also have additional supplies, including **retention** and **isolation** supplies.

3.1.34 power state: The state of a **supply net**, **supply port**, **supply set**, or **power domain**. The power state of a **supply net** or **supply port** is the `state` and `voltage` values of that **supply net** or **supply port**. The power state of a **supply set** is determined by the state of the **supply net**s in the set or by a logic expression if the supply network has not yet been created. The power state of a **power domain** is determined by the state of **supply set**s associated with the domain and **supply set**s or **supply net**s referenced in a named power state's **-supply_expr** defined for the domain. The term power state is often abbreviated to **state**.

3.1.35 power state table: A table that captures the legal combinations of **power state**s for a set of **supply net**s.

3.1.36 primary supply set: The **supply net** connections inferred for all elements in the **power domain**, unless overridden elsewhere.

3.1.37 rail: See the Liberty definition [B6] of `power_rail`.

3.1.38 receiver: Any **design element** that uses the value of a **net** as an input, either directly, or indirectly via a **port**. Ports are not receivers.

3.1.39 regular expressions: See the Tcl definition [B4] of *regular expressions*.

3.1.40 regulator: A **design element** that takes a set of input **supply net**s and provides the source for a set of output **supply net**s. The output voltage is a function of combining the input voltage and the logical state of any control signals.

3.1.41 retention: Enhanced functionality associated with selected **sequential element**s or a memory such that memory values can be preserved during the power-down state of the primary supplies.

3.1.42 retention register: A register that extends the functionality of a **sequential element** with the ability to retain its memory value during the power-down state following the application of a suitable save and restore protocol sequence.

3.1.43 rooted name: The name of a **design object** in the active scope or one of its descendants. It may include a **hierarchical separator character**, but may not start with the **hierarchical separator character**.

3.1.44 root supply driver: The origin of a supply, e.g., an on-system voltage regulator, bias generator modeled in HDL, or an off-chip supply source. *See also* **supply source**.

3.1.45 scope: A particular **design element** in the **logic hierarchy**.

3.1.46 sequential element: An **object** containing latch or flip-flop functionality.

3.1.47 simple name: A name in the active scope. The name does not contain any **hierarchical separator character**s.

3.1.48 simstate: The level of operational capability supported by the active state of a **supply set**.

3.1.49 source: The active device or port connection that propagates a value on a **net**.

3.1.50 state element: A storage device such as a flip-flop, latch, memory element, or stored value that would infer a **sequential element** from HDL code.

3.1.51 subdomain: A member of the set of subdomains comprising a composite power domain; each subdomain retains its original identity as well as the collective identity of the composite power domain (see 6.15).

3.1.52 supply function: The purpose of an individual **net** in a **supply set** (e.g., `power`, `ground`, etc.)

3.1.53 supply net: A **net** with **power state** semantics.

3.1.54 supply port: A **port** with **power state** semantics.

3.1.55 supply set: A collection of **supply net**s that provide a power source.

3.1.56 supply source: A supply port that propagates but does not originate a supply state and voltage value.

3.1.57 switch: A **design element** that conditionally connects one or more input **supply net**s to a single output **supply net** according to the logical state of one or more control inputs.

3.1.58 unnamed power state: A power state that is not predefined nor defined by the **add_power_state** command (see 6.8). *See also* **named power state**.

3.1.59 UPF simulation: A simulation run that depends on semantics described in this standard.

3.2 Acronyms and abbreviations

DAG directed acyclic graph

EDA electronic design automation

HDL hardware description language

IP intellectual property

PST power state table

RTL register transfer level

SAIF Switching Activity Interchange Format

UPF Unified Power Format

VCT value conversion table

VHDL VHSIC hardware description language

4. Power domains, supply sets, name spaces, and precedence

This clause provides an overview of power domains, supply sets and names in UPF.

4.1 Power domains

From a top-down view of power design specification, the fundamental object is the power domain. A *power domain* is a collection of design elements. Unless otherwise specified, elements of a power domain share a common primary supply set (see 4.3). The association of a primary supply set to all elements in a domain provides the fundamental semantics and automation opportunities in UPF as the primary supply set is implicitly connected to all elements within the domain.

The design consists of a hierarchical tree of design elements (*logic hierarchy*). The logic hierarchy level where a power domain is created is called the *scope* of the power domain. The set of design elements that belong to a power domain are said to be the *extent* of that power domain. This distinction is important—while a design element can be the scope of multiple power domains, it can be in the extent of one and only one power domain. It shall be an error if any design element is not included in a power domain and does not have supply nets connected to all supply ports of all elements after the application of UPF commands.

Each power domain exists within a scope of the logic hierarchy. A design element is a member of the power domain that includes the design element's parent instance, unless the design element has been explicitly included as an element of another power domain whose scope is the element or an ancestor of the element.

In addition to creating the power domains, UPF commands create and connect new objects that did not exist in the HDL description—e.g., switches, supply nets, supply ports, isolation elements, and level shifting elements. UPF provides the ability to specify the low power design intent. By tightly associating the power domain and other UPF created objects and connections with the HDL design, designers can design, verify, and implement complex chips complete with all power-related connections and functionality.

- For verification, every object in the power domain and its power distribution and control network exists within the logic design. This allows the designer to see these objects within the context of the design.
- For implementation, UPF provides a convenient way to manipulate and report information on groups of cells that share common power, ground, and bias supply connections.
- UPF ensures that the implementation semantics have matching simulation semantics; the results of the semantic interpretation of UPF are consistent across simulation and implementation.

A power domain can be either contiguous or non-contiguous. The power domain is *contiguous* if a connection from any object in the extent of the power domain to any other object in the extent does not require leaving that power domain; otherwise the domain is *non-contiguous*. All elements within the extent of the domain shall be within the scope of the domain or its descendants.

A net that is not driven by design elements in a particular domain is not corrupted by that domain; input ports and assignment statements are not considered drivers in this context.

Supply network objects (supply nets, supply ports, supply sets, and switches) are created within the logic hierarchy independent of the power domain definitions. This allows sharing of common components of the supply distribution network across multiple power domains independent of the power domain the object is within. Logic nets and ports created in UPF are created within the logic hierarchy independent of the power domain definitions. This allows the power control network to be created and distributed across power domains. All other UPF objects are created within the context of a power domain.

4.2 Supply nets and ports

Supply nets transport an electrical current. Supply ports provide the ability to connect a supply net to a design element (including switches). Supply nets and ports may be created in UPF or in the HDL design. If created in the HDL, the net or port shall be of the supply net type defined in the UPF SystemVerilog or VHDL package (see Annex B). When created in UPF, the supply net is created within the scope of a design element of the logic hierarchy. When created in UPF, a supply port is created on the interface of an element in the logic hierarchy. The name of the supply net or port cannot conflict with the name of an existing net or port (logic or supply) defined for that element.

Supply ports provide the ability for a design element to use a supply net that exists external to it. Supply ports consist of two halves. The first half is the `HighConn` side, which is visible to the parent of the design element whose interface contains the port. The second half is the `LowConn` side, which is visible internal to the design element whose interface contains the port.

When a supply or logic net in the active scope is connected to a supply or logic port on a child instance, the connection is made to the `HighConn` side of the port. When a supply or logic net in the active scope is connected to a port defined on the interface of the design element that is the active scope, the connection is made to the `LowConn` side of the port. If there is no net connected to a port referenced by a UPF command in a context requiring a net name, a net with the same name as the port is implicitly created in the active scope and implicitly connected to the port. If a port is referenced by a UPF command in a context requiring a net, an existing net that is already connected to the port shall be used. However, if no net has been connected to the port, a net with the same name as the port is implicitly created in the active scope and implicitly connected to the port.

A supply or logic net in the active scope can be connected to a port only if the port is directly visible in the same scope as the net or within the descendent tree of that scope. If the port is not directly visible in the same scope as the net, additional ports and nets shall be implicitly created to establish the connection from the net to the port. The implicitly created ports and nets shall have the same simple name as the net being connected unless that name conflicts with the name of an existing port or net; in which case, to avoid a name conflict, the tool shall create a name that is unique for that scope. See 6.35.

NOTE—Nets are propagated as necessary through the descendant tree and may be renamed to avoid name collision; therefore, the same simple name in different scopes may refer to nets that are independent and unconnected.

The electric current transported by a supply net is originated by a root supply driver. A *root supply driver* is the origin of a supply, e.g., an on-system voltage regulator, bias generator modeled in HDL, or an off-chip supply source. A root supply driver may be conditionally propagated by a switch (modeled in HDL or created in UPF, see 6.20). A supply net may be connected to one or more power switches or supply ports, which may be connected to one or more root supply drivers (see 6.22.1).

4.3 Supply sets

Because a single supply net by itself has no meaning relative to the power being supplied to any design element, UPF provides the ability to create supply sets. UPF predefines the following supply set handles for a domain: **primary**, **default_retention**, and **default_isolation**.

A *supply set* relates multiple supply nets as a complete power source for one or more design elements.

Each supply net in a supply set provides a function. UPF predefines the following supply net functions: **power**, **ground, pwell**, **nwell**, **deeppwell**, and **deepnwell**. A pre-defined supply set may be referenced through a supply set handle. Additional supply net functions may also be defined for a predefined supply set.

Once created, a supply set can be associated with a domain, retention strategy, or isolation strategy for a specific purpose. The supply nets of a set are implicitly "routed" to all elements that require them. Implicit "routing" includes the implicit creation of supply ports and nets required to propagate the supply net to each element where it is required.

UPF defines implicit and automatic connection semantics for the supply nets of a supply set based on the purpose of the supply set in a given domain or strategy context and the function that a supply net performs in the context of the supply set.

NOTE—A supply net may be included in more than one supply set. The function the supply net performs in one supply set is unrelated to the function it may perform in any other supply set in which it is included.

Supply nets can be connected in one of the following ways: explicitly, automatically, or implicitly.

4.3.1 Explicit connection of supply nets

A supply net may be *explicitly* connected to a port via a **connect_supply_net** command (see 6.13).

4.3.2 Automatic connection of supply nets

There are three ways supply nets can be automatically connected. They are as follows:

a) Supply nets are *automatically* connected to supply ports of design elements in a power domain based on *pg_type* specification (see 6.10).

b) Individual supply nets may be automatically connected to a port using a **connect_supply_net** command (see 6.13) with *pg_type* attributes defined.

c) The supply nets of a supply set may be automatically connected to a port using a **connect_supply_set** command (see 6.14), where the *pg_type* attribute of the supply function for a net in the supply set is matched to a design element port with a corresponding *pg_type* attribute.

   The supply nets of a supply set are automatically connected to the supply ports of elements under the following conditions:

   1) The element has a port with a *pg_type* attribute (a string-typed HDL attribute named `pg_type` or an implementation library model with a *pg_type* attribute).

      AND

   2) If the value of the *pg_type* attribute of the port matches a value specified via the **-connect** option of the **connect_supply_set** command (see 6.14), the supply net providing the function specified in the **-connect** option is automatically connected to the port.

The UPF specification is *erroneous* when it results in an incomplete set of connections for a design element.

The following precedence rules apply, from higher to lower precedence, for automatic and explicit connections of a supply net to a supply port.

connect_supply_net

connect_supply_set with **-elements** and **-connect**

connect_supply_set -connect

4.3.3 Implicit connection of supply nets

The supply nets of a supply set may be *implicitly* connected to design elements based on information in the power specification associating a supply set with a design element that is instantiated in the HDL code or implied by the power specification.

— Implicit connections provide the mechanism by which the supply nets of a supply set are connected to elements that do not have supply ports.

— The simstate simulation semantics for a supply set are applied to design elements implicitly connected to a supply set unless the **set_simstate_behavior** has been disabled (see 6.51).

The supply nets of a supply set are implicitly connected as the power source for an element when that element has *no supply ports*. Implicit connections define additional simulation semantic behavior (see Clause 5). Implicit connections are made under the following conditions:

a) Primary supply set

The supply nets of a domain's primary supply set are implicitly connected to any design element from the logic hierarchy that is within the extent of the domain if the element has no supply ports defined on its interface.

b) Retention supply set

The supply nets of a retention strategy's supply set are implicitly connected to the retention (i.e., balloon latch or shadow register) functionality that is implied for any register in the design to which the strategy applies.

c) Isolation supply set

The supply nets of a supply set for an isolation strategy are implicitly connected to the corresponding isolation functionality implied by the application of the strategy.

d) Level-shifter supply sets

The supply nets of a supply set for a level-shifting strategy are implicitly connected to the level-shifters implied by the application of the strategy.

4.3.4 Predefined supply set functions

The following names are reserved for predefined functions and may be used in any supply set definition:

a) **power** is the supply net that provides the power function of the supply set and by default shall be connected to ports having the *pg_type* **primary_power**.

b) **ground** is the supply net that provides the ground function of the supply set and by default shall be connected to ports having the *pg_type* **primary_ground**.

c) **pwell** is the supply net that provides the pwell bias of the supply set and by default shall be connected to ports having the *pg_type* **pwell**.

d) **nwell** is the supply net that provides the nwell bias of the supply set and by default shall be connected to ports having the *pg_type* **nwell**.

e) **deeppwell** is the supply net that provides the deeppwell bias of the supply set and by default shall be connected to ports having the *pg_type* **deeppwell**.

f) **deepnwell** is the supply net that provides the deepnwell bias of the supply set and by default shall be connected to ports having the *pg_type* **deepnwell**.

If a supply set is used within a domain, then a supply net shall be defined for each function required by that supply set for implementation. However, a simulator may support pre-implementation verification with only the specification that a supply set exists for the domain. A supply set that does not have supply nets defined for each of its required functions is *incompletely specified*. A reference to a supply net by its *symbolic* name is an *indirect reference*.

4.4 Naming rules

Names (identifiers) adhere to the following rules:

a) The first character of a name shall be alphabetic.

b) All other characters of a name shall be alphanumeric or the underscore character (_).

c) Names in UPF are case-sensitive.

4.5 Name space semantics

Names also need to adhere to the following name space semantics:

a) Objects created by a UPF command exist in the design; therefore, the names of those objects shall not conflict with a name that is visible within the same scope.

b) Some UPF objects are implicitly created. *Implicitly created objects* result from implied or inferred semantics and are not the direct result of creating a named UPF object. For example, supply nets are routed throughout the extent of a power domain as needed to implement the implicit and automatic connection semantics. This routing results in the creation of implicit supply ports and supply nets. UPF automatically names implicitly created objects to avoid creating a name conflict. The **name_format** command (see 6.35) can be used to provide a template for some implicitly created objects (such as isolation). Supply nets may be implicitly created and connected to supply ports and logic nets may be implicitly created and connected to logic ports (see 4.2).

c) Many command arguments require the specification of object names, such as instance names, ports, registers, nets, etc. Unless otherwise specified or unambiguous from the context, a name reference can be a simple name or the hierarchical name of an object. The names are relative to the active UPF scope. If the name is a hierarchical name, it shall be in the descendant tree of the active scope. See also 6.50.

d) Unless otherwise specified or unambiguous from the context, the terms *scope* and *UPF scope* are synonymous.

e) UPF objects may have record fields. These records comprise a name and a set of zero or more values. Record field names are in a local name space of the UPF object, e.g., a power domain may have strategies and supply set handles. Strategies themselves may also have supply set handles.

f) The following record field names are reserved in the specified context and cannot be redefined:

1) Domain record field space
   - **i) primary**
   - **ii) default_retention**
   - **iii) default_isolation**

2) Switch record field space
   - **supply**

3) Level-shifter strategy record field name space
   - **i) input_supply_set**
   - **ii) output_supply_set**
   - **iii) internal_supply_set**

4) Isolation strategy record field name space
   - **i) isolation_supply_set**
   - **ii) isolation_signal**

5) Retention strategy record field name space (see 6.33)
   - **i) retention_ref**
   - **ii) primary_ref**
   - **iii) save_signal**
   - **iv) restore_signal**
   - **v) UPF_GENERIC_CLOCK**
   - **vi) UPF_GENERIC_DATA**
   - **vii) UPF_GENERIC_ASYNC_LOAD**
   - **viii) UPF_GENERIC_OUTPUT**

g) The `.` character is the delimiter for the hierarchy of UPF record fields, e.g., `top/a/PDa.MY_SUPPLY_SET` refers to the supply set `MY_SUPPLY_SET` in power domain `PDa` in the logical scope `top/a`.

h) Name references include the refinements of complex objects such as: part-select or slice, bit-select, index-select, or field-select name (see IEEE Std 1800).

4.6 Attributes and HDLs

Hardware description languages include a mechanism for specifying properties of objects. These properties are called *attributes*. Certain UPF properties can be annotated directly in HDL source descriptions using attributes. The semantic for properties specified using HDL attributes is the same as the corresponding behavior defined by the UPF command alternative (see Clause 6). Table 1 enumerates the HDL attributes defined for UPF-compliant implementations.

Table 1—Attribute and command correspondence[a]

HDL attribute name	Attribute value specification	Equivalent UPF command arguments	See
UPF_clamp_value	<**"0"** | **"1"** | **"Z"** | **"latch"** | **"any"** | **"***value***"**>	**set_isolation -clamp_value** **set_port_attributes -clamp_value**	6.40 6.45
UPF_sink_off_clamp_value	<**"0"** | **"1"** | **"Z"** | **"latch"** | **"any"** | **"***value***"**>	**set_isolation -sink_off_clamp_value** **set_port_attributes -sink_off_clamp_value**	6.40 6.45
UPF_source_off_clamp_value	<**"0"** | **"1"** | **"Z"** | **"latch"** | **"any"** | **"***value***"**>	**set_isolation -source_off_clamp_value** **set_port_attributes -source_off_clamp_value**	6.40 6.45
UPF_pg_type	*pg_type_value* (see 4.3.4)	**set_port_attributes -pg_type**	6.45
UPF_related_power_pin	*port_name*	**set_pin_related_supply -related_power_pin** **set_port_attributes -related_power_port**	6.44 6.45
UPF_related_ground_pin	*port_name*	**set_pin_related_supply -related_ground_pin** **set_port_attributes -related_ground_port**	6.44 6.45
UPF_related_bias_pin	*port_name_list*	**set_port_attributes -related_bias_ports**	6.45
UPF_retention	<**"required"** | **"optional"**>	**set_retention_elements -retention required** **set_retention_elements -retention optional**	6.49
UPF_simstate_behavior	<**"ENABLE"** | **"DISABLE"**>	**set_simstate_behavior**	6.51
UPF_is_leaf_cell	<**"TRUE"** | **"FALSE"**>	**set_design_attributes -is_leaf_cell**	6.37

[a]Where two HDL attribute names are listed in the same table row, they are aliases for each other; where two UPF command arguments are listed in the same table row, they correspond to the same attribute.

It shall be an error if a UPF command conflicts with an attribute defined in an HDL file.

4.7 Precedence

To support concise, easily written low power specifications, UPF supports default and generic application of low power design intent. A UPF command specification can apply to a design object when specified directly on the object or an ancestor of the object. A command that applies to an element shall also apply to the descendents that match the constraints of the command. Application of multiple low power design intent specifications may result in overlapping UPF command effects.

When there is more than one power intent specification for a design object, the following precedence applies (ordered from highest to lowest priority):

a) Direct UPF commands.

   The power intent is applied through an explicit UPF command reference to a design object.

b) Power intent applied to a parent is inherited by each child and transitively applied to descendents, except when a direct UPF command applies.

c) Strategies specified with both the **-source** and **-sink** options.

d) Strategies specified with a **-source** or **-sink**.

The precedence of a command is independent of the active scope during the command processing.

It shall be an error if the precedence rules fail to uniquely identify the power intent that applies to an object.

NOTE—When *list* arguments to command options are created using **find_objects** (see 7.1), the level of precedence is based on the expanded value used as the argument, not as the pattern or regular expression used in **find_objects**.

4.8 Lexical elements

Special lexical elements (see Table 2) can be used to delimit tokens in the syntax.

Table 2—Special characters

Type	Character
logic hierarchy delimiter	/
escape character	\ (Only escapes the next character.)
bus delimiter, index operator, or within a `regex`	[]
range separator (for bus ranges)	:
record field delimiter	.

4.9 Units

The units for voltage values in UPF specifications shall be volts.

4.10 Boolean expressions

UPF expressions that are SystemVerilog Boolean expressions shall comply with the following restrictions:

— Control port names shall not conflict with any SystemVerilog keyword (see IEEE Std 1800).

— The Boolean expression syntax is limited to the tokens shown in Table 3.

Table 3—Boolean expression syntax tokens

Token	Meaning
( )	Parentheses
~ & | ^	Bit-wise negation, `AND`, `OR`, `XOR`
! && ||	Logical negation, `AND`, `OR`

5. Simulation semantics

This clause details the simulation semantics for the UPF commands (see also Clause 6).

5.1 Supply network creation

UPF supply network creation commands define the power supply network that connects power supplies to the design elements in a design. After these commands are applied, every design element in a design is connected to the power supply network. The *supply network* is a set of supply nets, supply ports, switches, and, potentially, regulators and generators. Supply sets are defined in terms of supply nets and conveniently define a complete power circuit for design elements. Supply sets simplify the management of related supply nets and facilitate connections based on the role the supply set provides for a power domain and the functions the supply nets provide within the set (see 4.3). The supply network defines how power sources are distributed to the design elements and how that distribution is controlled.

A supply port that *originates* a supply state and voltage value defines a *root supply driver*. Drivers may be created via the `create_root_supply_driver` function in the UPF packages (see Annex B). This allows models of regulators, switches, bias generators, etc. to be created in HDLs for connection to a supply network that may be defined in UPF.

A supply port that propagates but does not originate a supply state and voltage value defines a *supply source*. At any given time, a supply source can be traced through the supply network connectivity to a single root supply driver. The output port of a switch is not a root supply driver when the supply source can be traced through the switch to an input supply port (and the net connected to it) that is currently switched to the output port. HDL switch models should use the `assign_supply2supply` function to propagate the input supply to the output supply. `assign_supply2supply` propagates or maintains the trace back of the root supply driver information. Bias generators, voltage regulators, and switches modeled in HDL should create a root supply driver when the supply source originates from within the model.

Determination of the root supply driver is required for certain supply network resolution functions (see 6.22).

NOTE—Since the supply net type is defined in the UPF packages, it is possible to create the supply network entirely in HDL source.

5.2 Supply network simulation semantics

5.2.1 Supply network initialization

Simulation initialization semantics are defined by the corresponding HDL. Existing models rely on the HDL initialization semantics for operations such as initializing ROMs, etc. Therefore, UPF does not change the existing initialization semantics of HDLs. To model design power-up behavior, it is recommended that verification environments allow the simulation to initialize according to normal HDL semantics (as though the design is powered-on prior to the simulation initialization sequence). The verification environment can, after initialization, power down the design and then take the design through the power-up sequence.

The initial state of supply ports and supply nets is **UNDETERMINED** with an unspecified voltage value. The state of a supply set is **DEFAULT_CORRUPT** when at least one of the functions defined for the supply set is not associated with a supply net. To modify these states, see Annex B.

To facilitate modeling of non-inferable behavior in HDLs that can be used in both a UPF simulation and a traditional non-UPF simulation, the following are provided:

— Predefined constant of Boolean type: **UPF_POWER_AWARE**.

The value of this constant is **TRUE** in a UPF simulation, otherwise it is **FALSE**. This constant value is globally static only in a UPF simulation; i.e., its value is known at the time that SystemVerilog and VHDL `generate` statements are evaluated allowing the ability to specify logic that is conditionally generated only in a UPF simulation.

— In VHDL, a signal and, in SystemVerilog, a variable of type `power_state_simstate` can be declared within an architecture or module.

The name of this signal/variable shall be **upf_simstate**. **upf_simstate** can be used in a process's sensitivity list. It shall be an error if **upf_simstate** is assigned or connected to a port—it can only be used locally and in a read-only context. In a UPF simulation, **upf_simstate** shall represent the active simstate of the supply set that is implicitly, automatically, or explicitly connected to the design element when **simstate** behavior has been enabled for that element. If **simstate** behavior is disabled for the element, then **upf_simstate** shall remain the constant value **CORRUPT**.

5.2.2 Supply network update and evaluation

During simulation, each supply port and net maintains two pieces of information: a supply state and a voltage value. The supply state itself consists of two pieces of information: an on/off state and a full/partial state. The supply state values are **FULL_ON**, **OFF**, **PARTIAL_ON**, and **UNDETERMINED**. **PARTIAL_ON** typically represents a resolved supply net state when some switches, but not all, are **FULL_ON** or any switch is **PARTIAL_ON** (see also 6.22.2).

The state of a supply port and net is **FULL_ON** when it is connected to a root supply driver that is **FULL_ON**; the state is **UNDETERMINED** when there is no connected path to a root supply driver; the state is **OFF** when the root supply driver is **OFF**. By default, root supply drivers are **UNDETERMINED**; however, they may be turned **OFF**, **FULL_ON** or **PARTIAL_ON** to model the behavior and state of the root supply driver. The full/partial state represents the conductance of a switch along the supply path. The voltage value of a supply net is valid only when it is **FULL_ON** or **PARTIAL_ON**. A simulator may model the IR drop over the supply network. However, the semantics defined in this standard, such as the supply net resolution functions, presume an idealized supply network with no IR drop; the semantics for supply network resolution with modeled-IR drop are outside the scope of this standard.

The supply network is evaluated in the same step of the simulation cycle as the logic network. New root supply driver values are propagated along the connected supply nets in the same manner that logic values are propagated along the logic network.

NOTE—As no material distinction between **PARTIAL_ON** and **PARTIAL_OFF** exists, only **PARTIAL_ON** is defined.

5.3 Power switch modeling

During simulation, a power switch created with **create_power_switch** corresponds to a process that is sensitive to changes in its input port (net state and voltage value), as well as its control ports. [A general introduction to power switch behavior is described here (see 6.20 for the complete power switch semantics).] Whenever the signals on the control ports change, the corresponding on-state Boolean functions are evaluated. If an on-state function evaluates *True*, the switch is closed, which causes the state of its input port to propagate to the output port (or for a multiplexed switch, the corresponding input is switched to the output), otherwise the switch is opened—the output supply port is assigned the state **OFF** and the voltage value is unspecified. If: any of the control signals is `X` or `Z`, the input supply port is **UNDETERMINED**, the control signals match one of the error-state Boolean functions, or more than one on-state function evaluates *True*, then: the behavior of the output supply port is assigned the state **UNDETERMINED**, the voltage level shall be unspecified, and the acknowledge ports shall be driven `X`; in this case, implementations may issue a warning or an error.

Example:

Using the following **create_power_switch** command (see 6.20):

```
create_power_switch kb
-output_supply_port {outp pda_vdd}
-input_supply_port {inp1 yt}
-input_supply_port {inp2 db}
-control_port {cp1 eh}
-control_port (cp2 as}
-on_state {yt_on_kb inp1 {(cp1 && !cp2)}}
-on_state {db_on_kb inp2 {(!cp1 && cp2)}}
-off_state {kb_off {((cp1 && cp2) || (!cp1 && !cp2))}
-ack_port {ap yack {(cp1 ^ cp2)}}
```

creates an instance of an anonymous switch model that is functionally equivalent to the following SystemVerilog module definition:

```
import UPF::*;
module <anon> (
  output supply_net_type outp,
  output logic ap,
  input supply_net_type inp1, inp2,
input logic cp1, cp2 );

upf_object_handle in1H, in2H, outH;

initial begin
in1H = get_object( "inp1" );
in2H = get_object( "inp2" );
outH = get_object( "outp" );
if (!is_valid_handle( in1H ) || !is_supply_kind( in1H ) ||
   !is_valid_handle( in2H ) || !is_supply_kind( in2H ) ||
   !is_valid_handle( outH ) || !is_supply_kind( outH ))
 $display( "Invalid supply port connection on switch port" );
end

always@(cp1, cp2, inp1, inp2)
  case ({cp1, cp2})
    01 : begin
           assign_supply2supply( outp, inp2  );
           ap <= 1;
         end
    10 : begin
           assign_supply2supply( outp, inp1 );
           ap <= 1;
         end
    00 :
    11 :
         begin
           assign_supply_state( outp, OFF );
           ap <= 0;
         end
    default : begin

                assign_supply_state( outp, UNDETERMINED );
                ap <= X;
```

```
                $stop
              end
endmodule
```

The instance of the anon module is:

```
<anon> kb (.outp(pda_vdd), .inp1(yt), .inp2(db), .ap(yack), .cp1(eh),
.cp2(as));
```

5.4 Power states

Supply nets, supply ports, supply sets, and power domains have power states. The definition of the power state of each object is different.

5.4.1 Power states of supply nets and ports

Supply nets and ports form the foundation of the power distribution network specification. Each supply port and net maintains two pieces of information: a supply state and a voltage value, which together constitute the power state of the supply net or port. The power state information for supply nets and ports is defined by the **supply_net_type**; the state of being **FULL_ON**, **OFF**, **PARTIAL_ON**, or **UNDETERMINED** and the voltage level (which is relevant only for the **FULL_ON** and **PARTIAL_ON** states). A supply net may be included in a supply set. Each supply set has a reference supply. The *default reference supply* for a supply set is an implicit reference supply, defined to be `0` volts. The default reference supply can be explicitly overridden by specifying a supply net that is used as the reference supply for every supply net in the set. The voltage value of each supply net in a supply set is relative to the reference supply, which, in turn, may be at any voltage relative to the `0`-volt implicit reference supply.

5.4.2 Power states of supply sets

Continuing up from the foundation of supply nets, the next step in defining the supply network is the supply set—a collection of supply nets that together define a complete power supply. The *supply set* is composed of two or more supply nets. Therefore, the power state for a supply set is specified in terms of the supply nets that compose the set. It is the combined states of the constituent supply nets that determine the following:

— If there is current available to power an element, and

— The voltage level of the supply.

Supply set simulation semantics are applied to the elements connected to the supply set when the simstate behavior is enabled (see 6.51). The simulation behavior semantics (*simstate*) can be specified for a power state. The simstate specifies the level of operational capability supported by a supply set state. UPF defines several simstates that can be associated with supply set states. The simstates defined in UPF are an abstraction suitable for digital simulation. The following simstates are defined (from highest to lowest precedence):

a) **CORRUPT**—The power level of the supply set is either off (one or more supply nets in the set are switched off, terminating the flow of current) or at such a low level that it cannot support switching and the retention of the state of logic nets cannot be guaranteed to be maintained even in the absence of changes or activity in the elements powered by the supply.

b) **CORRUPT_ON_ACTIVITY**—The power level of the supply set is insufficient to support activity. However, the power level is sufficient that logic nets retain their state as long as there is no activity within the elements connected to the supply.

c) **CORRUPT_STATE_ON_ACTIVITY**—The power level of the supply set is sufficient to support combinational logic, but it is not sufficient to support activity inside state elements, whether that activity would result in any state change or not.

d) **CORRUPT_STATE_ON_CHANGE**—The power level of the supply set is sufficient to support combinational logic, but it is not sufficient to support a change of state for state elements.

e) **NORMAL**—The power level of the supply set is sufficient to support full and complete operational (switching) capabilities with characterized timing.

The simstate specification provides digital-simulation tools sufficient information for approximating the power-related behavior of logic implicitly connected to the supply set with sufficient accuracy.

NOTE 1—When greater accuracy is desired or required, a mixed signal or full analog simulation shall be used. Since analog simulations already incorporate power, this format provides no additional semantics for analog verification.

Simulation results reflect the implemented hardware results only to the extent the UPF simstate specification for a given power state of a supply set is correctly specified. For example, if verification is performed with simulation of a supply set in a power state specified as having a **CORRUPT_ON_ACTIVITY** simstate, but the implementation is more accurately classified as **CORRUPT_STATE_ON_CHANGE**, the simulation results will differ.

NOTE 2—In this example, the inaccuracy in simstate specification is conservative relative to the implemented hardware behavior. However, in other situations, inaccurate specifications can be optimistic resulting in errors in the implemented hardware that simulation failed to expose.

5.4.2.1 Predefined supply set power states

Every supply set has two predefined power states: **DEFAULT_NORMAL** and **DEFAULT_CORRUPT**. These power states are identical to explicitly defined power states except: It is an error if **DEFAULT_NORMAL** and **DEFAULT_CORRUPT** are used as the *object_name* in an **add_power_state** command (see 6.8). A supply set is in the **DEFAULT_NORMAL** state when all supply nets of the set are **FULL_ON** (see Table 4).

Table 4—Default supply set state with simstate == NORMAL

simstate	power	ground	pwell	nwell	deeppwell	deepnwell	All others
NORMAL	FULL_ON, any voltage	FULL_ON, any voltage	FULL_ON, any voltage	FULL_ON, any voltage	FULL_ON, any voltage	FULL_ON, any voltage	FULL_ON, any voltage

The simstate for **DEFAULT_NORMAL** is **NORMAL**. The supply set is in the **DEFAULT_CORRUPT** power state when the states of its supply nets do not match any defined power state, including the **DEFAULT_NORMAL** predefined state, for the supply set. The simstate for **DEFAULT_CORRUPT** is **CORRUPT**.

5.4.2.2 Specification of min, nom, max voltage levels

Since the power state of a supply net or port is the active supply net value, specification of the min, nom, and max voltage levels cannot be made at the supply net- or port-level. The appropriate place to specify the min, nom, and max voltage levels is in a supply set's power state specification (see 6.8).

5.4.3 Power states of power domains

The power state of a domain is determined by the state of supply sets associated with the domain. For example, the definition of a domain's `MY_DOMAIN_IS_ON` power state would logically require that the primary supply set be in a power state that is a **NORMAL** simstate (all supply nets of the primary supply set are on and the current delivered by the power circuit sufficient to support normal operation. Similarly, a

`SLEEP` mode for the domain may require the primary supply set to be in power state whose simstate is **NOT_NORMAL**, perhaps **CORRUPT** state, while appropriate retention and isolation supplies are **NORMAL**.

The state of logic elements may be a relevant aspect to the specification of a domain's power state, e.g., for a user-defined power domain called `DSP_PD`,

a) `DSP_PD` is in the state `my_on_pd_state` when:
   1) The logic signal that controls the switch for the domain's primary supply set is active. Early in the design phase, it may not be known if the power or ground net (or both) of the primary supply will be switched.
   2) The logic signal(s) enabling isolation or triggering retention save or restore are inactive.

b) `DSP_PD` is in the state `my_off_pd_state` when:
   1) The logic signal that controls the switch for the primary supply is inactive.
   2) If the isolation or retention supplies are switched, the control signals for those supplies are active (the power switch is on).
   3) Clock gating enable signals for the domain are inactive.
   4) The isolation enable(s) are active.

c) The power domain's power state may also be dependent on the clock period or similar signal interval constraint. For example, a domain in an operational bias mode needs to scale its clock frequency to a slower level to match the slower switching performance supported by the state of the primary supply set. The primary supply set's power state can include in the **-logic_expr** specification a constraint on the clock period or duty cycle interval. See 6.8.

A domain's power state can be defined directly in terms of supply nets using **-supply_expr** in addition to the **-logic_expr**.

— If a domain's power state **-logic_expr** specification includes comparison of another domain's active state to a power state defined on that domain, it is equivalent to including the **-logic_expr** and **-supply_expr** specifications for that power state of the referenced domain in the definition of the power state.

— If a domain's power state **-logic_expr** specification includes comparison of a supply set's active state to a power state defined on that supply set, it is equivalent to including the **-logic_expr** and **-supply_expr** specifications for that power state of the referenced supply set in the definition of the power state.

5.4.3.1 Incompletely specified supply sets

Prior to having the golden source (the HDL and UPF source used as input to implementation tools), the supply network may not be defined or may be partially defined. The design may have a power management block and associated power control signals that turn power switches on/off or control bias generators and voltage regulators once the supply network is fully specified. At this stage of design specification, the power domain's power states might only be defined in terms of the state of logic elements. The state of the domain's supply sets is added later and is necessary for a complete supply network specification. Through support of incremental refinement of the power state specification, early UPF simulations can be performed with only the logic net expressions defining the states. The power state definitions can be updated with **add_power_state** to incorporate supply network expressions (**-supply_expr**) or additional logic expressions (**-logic_expr**).

5.4.3.2 Simulation of power states

Each power domain and supply set is in a power state at any given point in time. A power state may be described and named via the **add_power_state** command (see 6.8). These are the named power states.

Unnamed power states may be legal or illegal (see 6.8). The power state of a supply set is determined by the state of each of the supply nets in the set. The power state of a domain is determined by the state of each of the supply sets associated with the domain and any supply nets or supply sets directly or indirectly referenced in the **-supply_expr** for any named power state for the domain.

Implementations may allow the choice of an overall mode: **min**, **nom**, or **max**. If not specified otherwise, the default shall be the **nom**inal mode. If **min** or **max** is specified, but only a single (nominal) value has been specified in the **-supply_expr** (see 6.8), that single value shall be used as the **min** or **max** value. The mode specification shall be used to determine if the **-supply_expr** of a power state evaluates *True*.

The power state of a supply set is determined as follows:

a) After the signal network is updated, including the supply network (see 5.2.2) and prior to evaluation of the power state of power domains:
   1) If any named power state of the supply set has a non-empty **-supply_expr**, then
      i) The **-supply_expr** for all named power states is evaluated.
      ii) The supply set is in a named power state if the **-supply_expr** for that named power state evaluates *True*. Zero, one, or more named power states may match (its **-supply_expr** evaluates *True*). (The predefined **DEFAULT_NORMAL** and **DEFAULT_CORRUPT** power states are named power states.)
      iii) In simulation, it shall be an error if any named power state matches and the matching state is defined as an illegal power state.

      Implementation tools may presume illegal power states do not occur.
      iv) In simulation, it shall be an error if the **-logic_expr** for any matching named power state does not evaluate *True*, see 5.4.3.3.

      Implementation tools may presume **-logic_expr** evaluates *True* for any matching named power state.
      v) In simulation, it shall be an error if the **-logic_expr** evaluates *True* for a non-matching power state (i.e., a power state for which **-supply_expr** does not evaluate *True*).

      Implementation tools may presume **-supply_expr** evaluates *True* for any non-matching named power state.
   2) Otherwise, the supply set is in a named power state if its **-logic_expr** evaluates *True*. Zero, one or more named power states may match.
b) The simstate semantics for all matching named power states are applied; the highest precedence corruption semantic prevails (see 5.4.2).
c) If zero named power states match, the supply set is in an unnamed power state. The legality of the unnamed power state is determined as specified in 6.8.

The power state of a power domain is determined as follows:

d) After the power state of supply sets is evaluated and prior to the evaluation of user-defined processes and `always` blocks, the power state of supply sets is evaluated.
   1) If any named power state of the domain has a non-empty **-supply_expr**, then
      i) The **-supply_expr** for all named power states is evaluated.
      ii) The domain is in a named power state if the **-supply_expr** for that named power state evaluates *True*. Zero, one, or more named power states may match.
      iii) In simulation, it shall be an error if any named power state matches and the matching state is defined as an illegal power state.

      Implementation tools may presume illegal power states do not occur.
      iv) In simulation, it shall be an error if the **-logic_expr** for any matching named power state does not evaluate *True*, see 5.4.3.3.

Implementation tools may presume **-logic_expr** evaluates *True* for any matching named power state.

v) In simulation, it shall be an error if the **-logic_expr** evaluates *True* for a non-matching power state (i.e., a power state for which **-supply_expr** does not evaluate *True*).

Implementation tools may presume **-supply_expr** evaluates *True* for any non-matching named power state.

2) Otherwise, the domain is in a named power state if its **-logic_expr** evaluates *True*. Zero, one, or more named power states may match.

e) If zero named power states matches, the domain is in an unnamed power state. The legality of the unnamed power state is determined as specified in 6.8.

5.4.3.3 Evaluation of the power state -supply_expr and -logic_expr expressions and reporting of mismatches

As evaluation-ordering dependencies between the update of the power control network (e.g., logic nets controlling the sources of supply nets, such as switches and bias generators) and the update of the supply network are likely, the reporting of mismatches when one expression evaluates *True* and the other does not is performed in a manner that avoids false error reporting due to any simulation-ordering dependencies within the same simulation time step.

— If a power state's **-supply_expr** or **-logic_expr** expressions do not match (one evaluates *True* when the other does not) and an error is not already scheduled for that power state, an error shall be scheduled to be reported at the end of the simulation time step (after all other activity has been completed for the time step and prior to the simulator advancing the simulation time).

— If the **-supply_expr** or **-logic_expr** is evaluated due to changes in the nets or power states referenced in the expression and both expressions evaluate *True*, the scheduled error message, if any, shall be deleted.

— At the end of the simulation time step, if a power state **-supply_expr** and **-logic_expr** expression mismatch error is scheduled, that error shall be reported.

NOTE—In SystemVerilog, the end of the simulation time step when the power state error for a **-supply_expr** and **-logic_expr** mismatch occurs is called the *reactive region*.

5.4.3.4 Incomplete supply network verification semantics

For any function of a supply set that is not associated with a supply net, an implicit supply net is created and associated with the function. Implicitly created supply nets are initialized the same as explicitly created supply nets (see 5.2.1).

Tools may provide mechanisms to change the power state of the supply set or power domain. Such mechanisms are outside the scope of this standard.

The power state of a supply set or power domain may also be changed from an HDL test bench in simulation using the `set_power_state` function defined in the UPF packages (see Annex B).

a) When a supply set's power state is changed through `set_power_state` (directly or indirectly):

1) An error is issued if the **-logic_expr** for the power state does not hold for the new state. There is no **-logic_expr** for the default supply set states: **DEFAULT_NORMAL** and **DEFAULT_CORRUPT**.

2) Any supply nets referenced in the **-supply_expr** for the new state are forced into the state specified in the **-supply_expr** specification of the new state. Any state that satisfies the **-supply_expr** is permitted. The forcing of a supply port shall result in an event on the supply net. Therefore, any change in supply nets resulting from the execution of the `set_power_state` function shall be deferred until the start of the next supply network

evaluation cycle. Forcing of a supply port is equivalent to forcing any supply net connected to the output side of the port.

NOTE—Implicitly created supply nets may be referenced through symbolic names (e.g., `primary.power`).

3) The implicitly created supply nets of the set that are not referenced in the **-supply_expr** through their symbolic names (e.g., `primary.power`), shall have their state set as follows:

   i) If the simstate of the state is **CORRUPT**: the state shall be set to **OFF** and the voltage value is unspecified.

   ii) For any other simstate: the state shall be set to **FULL_ON** and the voltage value is unspecified.

b) When a domain's power state is changed through **set_power_state** (directly or indirectly):

1) It shall be an error if the **-logic_expr** for the power state does not hold for the new state. During this check, any sub-expressions that reference the power state of an object are assumed to hold.

2) Any supply net referenced in the **-supply_expr** is forced as described in a.2) for supply sets.

3) For each domain or supply set referenced in **-logic_expr**, a recursive call to `set_power_state` shall be made to ensure the power states of those objects are set to the state required by the new state.

5.4.4 Power states of systems and subsystems

What constitutes a system is contextual. In one context, a system may be considered as complete by itself, e.g., one chip of a multi-chip or multi-board low power system. Although it might seem reasonable to define a "system" as that which is automatically implemented, UPF is not limited to that context and the verification of an entire system composed of multiple chips each with its own power design specification, as well as an overall power design specification for the board on which the chips are placed, is supported. The power states of a system or subsystem are attributed on a power domain. The use of the term *system* includes the term *subsystem*.

As all elements of a design shall be in a power domain, specification of power states for a system or subsystem are attributed to the power domain(s) that are defined for the (sub)system. As a system power state may depend on the state of more than one power domain, the power state specification for a power domain may include references to the states of domains defined on scopes in the logic hierarchy that are descendents of the "higher-level" power domain. (Here, "higher-level" refers to the location of the power domain's scope being closer to the design's top-level root design element relative to the scope of another power domain.) Therefore, UPF allows the power state for a power domain to reference the power state of any power domain, supply set, supply net or supply port that is visible in the descendent tree of the scope of the domain.

For example, assume the domain `CORE_PD` is defined on the root scope of a processor design, the power states of `CORE_PD` can reference lower-level power domains such as `CACHE_PD`, `ALU_PD`, and `FP_PD` in the specification of its power states. Thus, an example power state of `FULL_OP` for `CORE_PD` would reasonably require that its primary supply set is in a **NORMAL** simstate (all supply nets of the primary supply set are on and the voltage of the supply is sufficient for normal operations) and that the `CACHE_PD`, `ALU_PD`, and `FP_PD` are all in an equivalent fully operational mode. In contrast, a `NON_FP_OP` mode for the `CORE_PD` may be defined identically to `FULL_OP`, except the `FP_PD` may be in a `SLEEP` mode. By allowing a higher level domain to reference lower level domain's power states in the specification of its own power states, subsystem and system power states can be defined in UPF.

NOTE—Although the top-down specification of power states suggests a power domain's power states are defined in terms of the power states of supply sets and lower-level power domains, the power state of a domain can be specified entirely in terms of the state of supply sets and/or supply nets and supply ports; i.e., the hierarchical specification can be collapsed into a (relatively) flat power state specification. Top-down, hierarchical power state specification is convenient when the power design starts prior to the existence of the complete supply network and is refined into an

implementation. The flat specification of power states of domains in terms of direct references to supply nets may be faster and more concise when the power state specification is not captured until after the supply network is specified. However, flat power state specifications may be less flexible and more difficult to maintain over time and require visibility into and understanding of all aspects of the design.

5.5 Power state name spaces

Power states are attributed to specific objects in the design. The power states can be referenced by specifying the *object_name*, where *object_name* can be a hierarchical name denoting a power domain, supply set, or supply net. Power states are attributes of the object. Specifically, *power states* of a domain are attributes of the domain and not attributes of the scope of the domain. Thus, a design element may be the scope for multiple domains, each domain containing states with the same name (e.g., `sleep`) without incurring a name space collision.

The following objects may have power states attributed to them:

- Power domains
- Supply sets
- Supply nets
- Supply ports

The **add_power_state** command (see 6.8) is used to define the legal and illegal power states of power domains and supply sets. The `set_power_state` function in the UPF packages is used to set the power state of an object during simulation.

The range of possible states for supply nets and ports is defined by the type `supply_net_type` in the UPF packages. The state of supply nets and ports can be set through the `assign_supply2supply` or `assign_supply_state` functions in the UPF packages. `assign_supply2supply` propagates the association of the source supply net's root supply driver as well as the source's state and voltage values to the destination. `assign_supply_state` is used to assign a supply port that is a root supply driver.

A power state shall be defined before it can be referenced. Semantically, the transition of an object from one power state to another is a power state *event* for the object. The state of a supply net is referenced as a Boolean expression (see 4.10) in the same manner that the state of a logic net is referenced. The power state of a supply set or power domain can be referenced in an expression simply through the supply set or power domain name.

Examples

```
supply_set_li == SLEEP
-- Returns TRUE if supply_set_li is in a state consistent with state SLEEP

ALU_PD != FULL_OP
-- Returns TRUE if the ALU_PD is in a state inconsistent with FULL_OP
```

5.6 Simstate simulation semantics

Each simstate has well-defined simulation semantics, as specified in the following subclauses. Multiple power states may be defined with the same simstate specification. The simstate semantics are applied to all elements that have the supply set connected to it (including no supply net connections except those implied by the supply set connection to the element) and that have the simstate semantics implicitly or explicitly enabled. The use of the term *elements* in the rest of this subclause refers only to the elements related to the supply set as described in this paragraph.

Elements implicitly connected to a particular supply set have simstate semantics enabled by default. Elements automatically or explicitly connected to a particular supply set have simstate semantics disabled by default. Use **set_simstate_behavior** to override the default enablement of simstate semantics (see 6.51).

The supply set powering a state element or the driver for a net may be in a state that the supply is not adequate to support normal operational behavior. Under specified circumstances while in these states, the logic value of the state element or net becomes unknown. A corrupt value for a state element or net indicates the logic state of the state element or net is unknown due to the state of the supply powering the state element or driver of the net. The corrupt value of a state element or net shall be the HDL's default initial value for that object's type, except VHDL `std_ulogic` and `std_logic` typed-objects shall use `X` as the corruption value (not `U`).

NOTE—An object may be declared with an explicit initial value. This explicit initial value has no relationship to the corrupt value for the object. For example, in VHDL, the objects of `Integer` type have the default initial value of `Integer'Left` (`-2147483648` for a system using 32-bits to represent `Integer` types). A process variable inferring a state element may be declared to be of type `Integer` with an initial value of `0`. The corrupt value for the variable is `Integer'Left`, not `0`.

The following subclauses define the simulation semantics for simstates. These semantics are applied to the elements connected to the supply set with simstate behavior **ENABLED**.

5.6.1 NORMAL

This state is a normal, power-on functional state. The simulator executes the design behavior of the elements consistent with the HDL or UPF specification that defines the element.

5.6.2 CORRUPT

This state is a non-functional state. For example, this state can be used to represent a power-gated/power-off supply set state. In this power state, state elements powered by the supply set and the logic nets driven by elements powered by the supply set are corrupted. The element is disabled from evaluation while this state applies.

As long as the supply set remains in a **CORRUPT** simstate, no additional activity shall take place within the elements, i.e., all processes modeling the behavior of the element become inactive, regardless of their original sensitivity list. Events that were scheduled for elements supplied by the supply set before entering this simstate shall have no effect.

5.6.3 CORRUPT_ON_ACTIVITY

This state is a power-on state that is not dynamically functional. For example, this state can be used to represent a high-voltage threshold, (body-bias) state that does not have characterized (defined) switching performance. In this simstate, the logic state of the elements is maintained unless there is activity on any of the element's inputs. Upon activity on any input, then all state elements and logic nets driven by the element are corrupted.

5.6.4 CORRUPT_STATE_ON_CHANGE

This state is a power-on state that represents a power level sufficient to power normal functionality for combinational functionality but insufficient for powering the normal operation of a state element if the state element is written with a new value. The simulator executes the design behavior of the elements consistent with the HDL or UPF specification that defines the element, except that any change to the stored value in a state element results in the writing of a corrupt value to the state element.

5.6.5 CORRUPT_STATE_ON_ACTIVITY

This state is a power-on state that represents a power level sufficient to power normal functionality for combinational functionality but insufficient for powering the normal operation of a state element if there is any write activity on the state element. The simulator executes the design behavior of the elements consistent with the HDL or UPF specification that defines the element, except that any activity inside state elements, whether that activity would result in any state change or not, results in the writing of a corrupt value to the state element.

5.6.6 NOT_NORMAL

This is a special, placeholder state. It allows early specification of a non-operational power state while deferring the detail of whether the supply set is in the **CORRUPT**, **CORRUPT_ON_ACTIVITY**, **CORRUPT_STATE_ON_CHANGE**, or **CORRUPT_STATE_ON_ACTIVITY** simstate. If the supply set matches a power state specified with simstate **NOT_NORMAL**, the semantics of **CORRUPT** shall be applied, unless overridden by a tool-specific option. **NOT_NORMAL** semantics shall never be interpreted as **NORMAL**.

NOTE 1—Using the default interpretation of **CORRUPT** for **NOT_NORMAL** provides a conservative—the broadest corruption semantics—for simulation of the design for functional verification. However, a conservative interpretation of **NOT_NORMAL** for other tools, such as power estimation tools, might be to use a bias or lowered voltage level interpretation such as **CORRUPT_ON_ACTIVITY**.

NOTE 2—As it is possible for two or more power states of a supply set to match the state of the supply set's nets and for multiple simstate specifications to apply simultaneously, the effective result is that the simstate with the broadest corruption semantics shall apply. For example, a supply set that matches power states with simstates of **CORRUPT_STATE_ON_CHANGE** and **CORRUPT_STATE_ON_ACTIVITY** shall result in the application of **CORRUPT_STATE_ON_ACTIVITY** simstate semantics being applied.

5.7 Transitioning from one simstate state to another

The following subclauses define the simulation semantics for transitions from one simstate to another. These semantics are applied to the elements connected to the supply set with simstate behavior **ENABLED**.

5.7.1 Any state transition to CORRUPT

In this case, the nets and state elements driven by the elements connected the supply set in this simstate shall be corrupted. The elements connected to this supply set are inactive as long as the supply set is in the **CORRUPT** simstate.

5.7.2 Any state transition to CORRUPT_ON_ACTIVITY

In this case, the active state of nets and state elements driven by the element shall remain unchanged at the transition. The processes modeling the behavior of the element shall remain enabled for activation (evaluation). Any net or state element that is actively driven after transitioning to this state shall be corrupted.

Any attempt to restore a retention register's retained value while in the **CORRUPT_ON_ACTIVITY** state shall result in corruption of the register's value.

5.7.3 Any state transition to CORRUPT_STATE_ON_CHANGE

In this case, the active state of nets and state elements driven by the element shall remain unchanged at the transition. The processes modeling the behavior of the element shall be enabled for activation (evaluation).

5.7.4 Any state transition to CORRUPT_STATE_ON_ACTIVITY

In this case, the active state of nets and state elements driven by the element shall remain unchanged at the transition. The processes modeling the behavior of the element shall be enabled for activation (evaluation).

5.7.5 Any state transition to NORMAL

The corruption, if any, of all state elements and nets driven by the element concludes. Continuous assignments become sensitive to changes to their right-hand side expressions and other combinational processes (e.g., `always_comb` block in SystemVerilog) resume their normal sensitivity list operation. All continuous assignments and other combinational processes are evaluated on the transition to **NORMAL**, regardless of their sensitivity list activity, to ensure constant values and current input values are properly propagated. Level-sensitive functionality of sequential logic within the element shall be evaluated. Edge-sensitive functionality of sequential logic within the element shall not be evaluated at this transition.

5.7.6 Any state transition to NOT_NORMAL

NOT_NORMAL is simulated according to the interpretation of this placeholder simstate (see 5.6.6).

6. Commands

This clause documents the syntax for each UPF command. For details concerning the simstate semantics, see Clause 5.

6.1 Conventions used

Each command in this clause consists of a command keyword followed by one or more parameters. All parameters begin with a hyphen (`-`). The meta-syntax for the description of the syntax rules uses the conventions shown in Table 5.

Table 5—Document conventions

Visual cue	Represents
`courier`	The `courier` font indicates UPF or HDL code. For example, the following line indicates UPF code: `create_power_domain PD1`
bold	The **bold** font is used to indicate key terms, text that shall be typed exactly as it appears. For example, in the following command, the keywords "create_power_domain" shall be typed as it appears: **create_power_domain** *domain_name*
italic	The *italic* font represents user-defined UPF variables. For example, a supply net needs to be specified in the following line (after the "connect_supply_net" key term): **connect_supply_net** *net_name*
list	*list* (or *xxx_list*) indicates a Tcl list, which are denoted with curly braces **{**….**}** or as a double-quoted string of elements **"**….**"**.
time_literal	*time_literal* indicates the SystemVerilog `time_literal`.
[ ] square brackets	Square brackets indicate optional parameters. For example, the following parameter is optional: [**-elements** *element_list*]
[ ] bold square brackets	Bold square brackets are required. For example, in the following parameter, the bold square brackets (surrounding the `0`) need to be typed as they appear: *domain_name*.*isolation_name*.**[0]**
{ } curly braces	Curly braces ({ }) indicate a parameter list, which usually can be repeated. For example, the following shows one or more control ports can be specified for this command: {**-control_port** {*port_name*}}*
{ } bold curly braces	Bold curly braces are required. For example, in the following parameter, the bold (inner) curly braces need to be typed as they appear: {**-state** {*name* <*nom* | *min max* | <*min nom max*> | **off** >}}*
* asterisk	An asterisk (`*`) signifies that parameter can be repeated. For example, the following line means multiple acknowledge delays can be specified for this command: [**-ack_delay** {*port_name delay*}]*

Table 5—Document conventions (continued)

Visual cue	Represents
< > angle brackets	Angle brackets (< >) indicates a grouping, usually of alternative parameters. For example, the following line shows the "power" or "ground" key terms are possible values for the "-type" parameter: **-type <power | ground>**
| separator bar	The separator bar (|) character indicates alternative choices. For example, the following line shows the "in" or "out" key terms are possible values for the "-direction" parameter: **-direction <in | out>**

6.2 Generic UPF command semantics

— All commands are executed in the active UPF scope, except as specifically noted.

— / or . . shall not be used as the first character of any name token in a command, except in the **set_scope** command.

— All **map_*** commands specify the elements to be used in implementation. These specifications override the elements that may be inferred through a strategy. The behavior of this manual mapping may lead to an implementation that is different from the RTL specification. Therefore, it may not be possible for logical equivalence checking tools to verify the equivalence of the mapped element to its RTL specification.

6.3 effective_element_list semantics

The *effective_element_list* is the set of elements to which a command applies. The *effective_element_list* is constructed from the arguments provided to the command. The terms used in the description of this construction include: *element_list, exclude_list, aggregate_element_list, aggregate_exclude_list, prefilter_element_list,* and *effective_element_list.* The *element_list* and *exclude_list* are lists that contain the elements specified by an instance of the command. The *effective_element_list*, *aggregate_element_list*, and *aggregate_exclude_list* are associated with the named object of the command.

The following arguments can determine the *effective_element_list*:

a) **-elements** *element_list* adds the rooted names in *element_list* to the *aggregate_element_list*. It is not an error for an element to be included more than once.

b) **-exclude_elements** *exclude_list* adds the rooted names in *exclude_list* to the *aggregate_exclude_list*. It is not an error for an element to be included more than once. It is not an error for an element in the exclude list to not be in the *aggregate_element_list*.

c) When **-transitive** is **TRUE** elements (see 6.3.1) in *aggregate_element_list* that are not leaf cells are processed to include the child elements (see 6.3.2).

d) The *prefilter_element_list* comprises the *aggregate_element_list* with any matching elements from the *aggregate_exclude_list* removed (see 6.3.2).

e) The command arguments identified as filters are predicates that shall be satisfied by elements in the *effective_element_list*. The *prefilter_element_list* is filtered by the predicates to produce the *effective_element_list* (see 6.3.2).

f) The range of legal element types is command dependant for each command that uses **-elements**. Each command specifies the effect of an empty *aggregate_element_list*. An explicitly empty list may be specified with {}.

6.3.1 Transitive TRUE

The detailed semantics of transitive TRUE are described using Figure 4, Figure 5, and Figure 6. The figures are exemplary, the text provides a semantic for the validation of the result.

a) Given a design as shown in Figure 4 with a design element `A` in the active scope, where `A` has child elements `B`, `C`, and `D`; `B` has child elements `E` and `F`, `C` has child elements `G` and `H`, and `D` has child elements `I` and `J`.

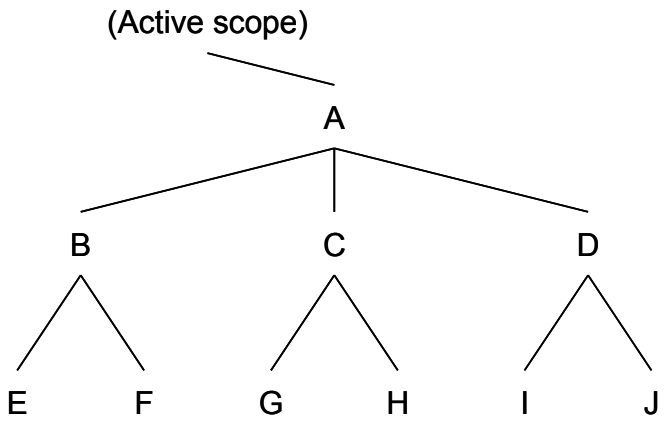

Figure 4—Element processing example design fragment

b) If the specification:

```
-elements {A A/C/H} -exclude_elements {A/C A/D} -transitive TRUE
```

is applied to the design fragment shown in Figure 4, then Figure 5 shows the four specified elements by indicating them as boxed; those specified with exclude are shown with strike-through text.

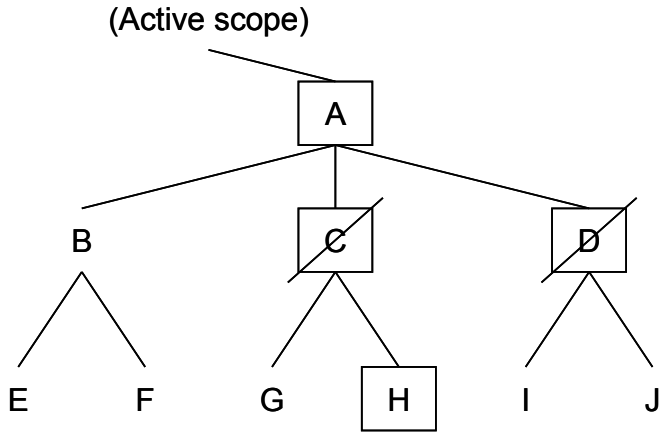

Figure 5—Element processing specification

c) Figure 6 shows the results of the *effective_element_list*. The list includes `{A A/B A/B/E A/B/F A/C/H}`

The elements included or excluded by transitivity are shown as dashed-boxes or with strike-through text, respectively.

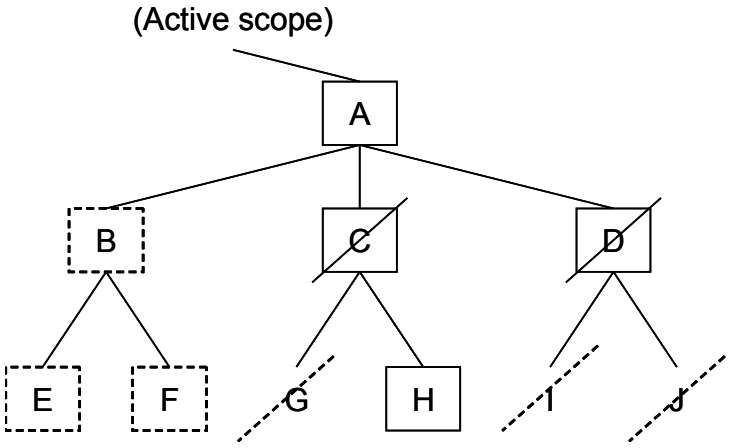

Figure 6—Element processing result

6.3.2 Result

The required result is derived as follows:

```
Begin // at the root of the active scope.
   Initialize by traversing the hierarchy and set element.mark := exclude
   For each element in the aggregate_element_list do
      set element.mark := includeP
      if (transitive = TRUE AND element NOT Leaf_Cell) then
         foreach child in element call mark_child(child, include)
      end if
   done
   For each element in the aggregate_exclude_list do
      set element.mark := excludeP
      if (transitive = TRUE AND element NOT Leaf_Cell) then
         foreach child in element call mark_child(child, exclude)
      end if
   done
   For each element in the aggregate_element_list call check_and_add(element)
done

proc mark_child(element, value)
   if (element.mark != excludeP AND element.mark != includeP ) then
      element.mark := value
      if (element NOT Leaf_Cell) then
         foreach child in element call mark_child(child, value)
      end if
   end if
end proc

proc check_and_add(element)
   if (element.mark = includeP OR element.mark = include) then
      if (for all filters filter(element) = TRUE) then
         add element to effective_element_list
         element.mark := processed
         if (transitive = TRUE AND element NOT Leaf_Cell) then
            foreach child in element call check_and_add(child)
         end if
```



```
        end if
    end if
end proc
```

NOTE—Implementations may use any data structure or algorithm that produces the same results as the method above.

6.4 Command refinement

Some UPF commands support progressive refinement. Refinable command arguments are shown in **boldface-green text** and labeled with an **R** in their respective *Arguments* listings. (See: 6.8, 6.15, 6.19, 6.24, 6.40, 6.42, and 6.47.) Refinable commands may be called on a particular named object multiple times. The first instance of the command defines the object; all mandatory arguments shall be declared in this call and any other arguments may also be included. Subsequent instances of the command for the same object shall include the **-update** option, all mandatory arguments, and any desired refinable arguments (e.g., using **-simstate** in **add_power_state** to add/refine the simstate for a particular power state [see 6.8]). The end result will be as if all of the arguments had been included in the initial instance.

Except as noted in the following list or in the command descriptions, it shall be an error to specify arguments (or HDL attributes) that contradict the power intent of previous declarations.

- **-elements** *element_list* provided to commands on successive invocations for the same object add to the *aggregate_element_list* associated with that object.
- **-exclude_elements** *exclude_list* provided to commands on successive invocations for the same object add to the *aggregate_exclude_list* associated with that object.
- The resulting *effective_element_list* is determined using the *aggregate_element_list* and *aggregate_exclude_list* associated with that object.
- UPF commands re-invoked on the same object shall use **-update** to refine the object and shall not contradict previously specified power intent.
- It shall be an error if **-update** is specified on the first invocation of any command applying to a named object.

The consequence of a set of UPF commands being applied to a particular named object is the same as if the first of the commands were modified to comprise the union of all arguments contained in all commands of the set.

Example:

This shows a multiple-part refinement for a usage of **set_isolation** (see 6.40).

a) Constraining specification

```
set_isolation demo_strategy -domain pda
    -elements {a b c d}
    -clamp_value 0
```

b) Logical configuration

```
set_isolation demo_strategy -update -domain pda
    -isolation_signal {iso_en}
    -isolation_sense {LOW}
```

c) Adding elements to the strategy

```
set_isolation demo_strategy -update
    -elements {e f g}
```

d) Supply set connection

```
set_isolation demo_strategy -update -domain pda
    -isolation_supply_set pda_isolation_supply
```

The implementation-independent part of the low power intent [see a)] could also be declared in the HDL using the following attributes:

```
(* UPF_clamp_value = "0" *) out a;
(* UPF_clamp_value = "0" *) out b;
(* UPF_clamp_value = "0" *) out c;
(* UPF_clamp_value = "0" *) out d;
```

In this case, the declaration shall have identical semantics to the equivalent UPF command.

6.5 Error handling

If an error condition occurs, e.g., an incorrect command-line option is specified, then a `TCL_ERROR` exception shall be raised. This exception can be caught using the Tcl `catch` command, so these errors can be prevented from aborting the active **load_upf** command (see 6.28). These errors shall have no impact on further commands. Processing may continue after the error is caught. Sequencing of the error `catch` and the choice of continuation is tool-dependent. The state of the design after an error is not defined. Specifically, a command that raises an error may partially complete before aborting.

In general, all commands that fail shall raise a `TCL_ERROR`. As described in the Tcl documentation, the global variables accessible after an error occurs include *errorCode* and *errorInfo*.

NOTE—The message string returned by the Tcl `catch` command is not specified in this standard.

6.5.1 errorCode

After an error has occurred, this variable contains additional information about the error in a form that is easy to process with programs. *errorCode* consists of a Tcl list with one or more elements. The first element of the list identifies a general class of errors and determines the format of the rest of the list. There are several formats for *errorCode* used by the Tcl code; see also the *Tcl command reference* [B5].

Errors defined in this standard are prefixed with `UPF_`, as shown in the following definitions. Individual applications that implement this standard may define and use additional error codes that do not start with `UPF_`. Implementations need to use errors appropriate to their application.

a) `UPF_RETURN_NOT_VISIBLE error_data`

   This error code indicates the objects referenced in the response of a query are not in the active scope. Queries return object names rooted in the active scope. Because they are called from an active scope that may be different from the scope in which all objects to be returned are visible, it shall be an error if the query cannot represent the objects to be returned as a rooted name.

   The `UPF_RETURN_NOT_VISIBLE` error may be raised in these cases where there are no other errors. When this code is returned, the `error_data` is defined to be the same as the query would have returned, but with fully qualified names for the objects not visible in the active scope.

b) `UPF_QUERY_OBJECT_NOT_DEFINED error_data`

   This error code indicates a query is called with a specific name argument and the named object is not defined in the active scope.

c) `UPF_UPDATE_CONFLICT error_data`

   This error code indicates a command has been called with arguments that conflict with previously specified values.

d) `UPF_UPDATE_MISSING error_data`

   This error code indicates a command has been called without the **-update** argument and the named object has already been defined.

e) `UPF_UPDATE_OBJECT_NOT_FOUND error_data`

This error code indicates a command has been called with the **-update** argument and the named object has not been previously defined.

f) `UPF_MERGE_FAILURE error_data`

This error code indicates a **merge_power_domains** command failed.

g) `UPF_OBJECT_NOT_FOUND error_data`

This error code indicates a object referenced in a command is not defined in the active scope.

6.5.2 errorInfo

See the *Tcl command reference* [B5].

6.6 add_domain_elements

Purpose	Add design elements to a power domain	
Syntax	**add_domain_elements** *domain_name* **-elements** *element_list*	
Arguments	*domain_name*	The power domain to modify.
	-elements *element_list*	The list of design elements to add. The elements shall be referenced relative to the active scope and are the descendents of the scope of the specified power domain.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

The **add_domain_elements** command provides the ability to separate the creation of a power domain from the specification of the elements contained within it. This is similar to only specifying the elements using the **-elements** option in the **create_power_domain** command (see 6.19). The effect of **add_domain_elements** is additive, i.e., a power domain consists of any elements specified in the **create_power_domain** command and those elements specified in any **add_domain_elements** commands.

It shall be an error if *domain_name* does not indicate a previously created power domain.

This command is semantically equivalent to

```
proc add_domain_elements {dn elements el} {
    if { string equal $elements "-elements" }{
        create_power_domain $dn -update -elements $el
        return 1
    } else {
         return -code TCL_ERROR \
              -errorcode $ecode \
              -errorinfo $einfo \
              $resulttext
    }
}
```

where any *`italicized`* arguments are implementation-defined.

Syntax example:

```
add_domain_elements U1/PD1 -elements {U1/U2/foo1 U1/U2/foo2}
```

6.7 add_port_state

Purpose	Add states to a port	
Syntax	**add_port_state** *port_name* {**-state** {*name* <*nom* | *min max* | *min nom max* | **off**>}}*	
Arguments	*port_name*	The name of the supply port. Hierarchical names are allowed.
	-state {*name* <*nom* | | *min max*| *min nom max* | **off**>}	The *name* and value for a state of the supply port. The value can be a nominal voltage; a pair specifying the minimum and maximum voltage; a triplet of values specifying the minimum, nominal, and maximum voltages; or **off**.
Return value	Return the fully qualified name (from the active scope) of the created port or raise a `TCL_ERROR` if any of the port states are not added.	

The **add_port_state** command adds state information to a supply port. If the voltage values are specified, the supply net state is **FULL_ON** and the voltage value is the single nominal value or within the range of min to max; otherwise, if **off** is specified the voltage value is **OFF**.

It shall be an error if *port_name* does not already exist.

N.B. The simulation semantics differ from UPF 1.0—the first invocation of the command for a specific supply port no longer defines the default state of the port, see Clause 5.

It shall be an error if *nom* < *min* or *max* < *nom*.

Syntax example:

```
add_port_state VN1
   -state {active_state 0.88 0.90 0.92}
   -state {off_state off}
```

6.8 add_power_state

Purpose	Attribute power state(s) to a power domain or supply set		
Syntax	**add_power_state** *object_name* {**-state** *state_name* {[**-supply_expr** {*boolean_function*}] [**-logic_expr** {*boolean_function*}] [**-simstate** *simstate*] [**-legal** | **-illegal**] [**-update**]}* [**-simstate** *simstate*][**-legal** | **-illegal**] [**-update**]		
Arguments	*object_name*	Simple name of a power domain or supply set.	
	-state *state_name*	*state_name* is the simple name of the state being defined or refined.	
	-supply_expr {*boolean_function*}	**-supply_expr** specifies a Boolean expression defined in terms of supply nets that evaluates to *True* when the object is in the state being defined.	**R**
	-logic_expr {*boolean_function*}	**-logic_expr** specifies a Boolean expression defined in terms of logic nets and supply nets that evaluates to *True* when the object is in the state being defined.	**R**
	-simstate *simstate*	**-simstate** specifies a **simstate** for the power states associated with a supply set, valid values are **NORMAL**, **CORRUPT_STATE_ON_CHANGE**, **CORRUPT_STATE_ON_ACTIVITY**, **CORRUPT_ON_ACTIVITY**, **CORRUPT**, and **NOT_NORMAL**. See 5.4.2. If a *state_name* has not been defined with a specific *simstate*, the **-simstate** can be used to set the simstate.	**R**
	-legal | **-illegal**	The legality of the state as either legal or illegal, the default is **-legal**.	
	-update	Use **-update** if the *state_name* has already been defined in the *object_name*.	
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.		

add_power_state attributes an object with a power state definition. A state name is simply an identifier; it has no semantic meaning. A power state specification is independent of any other power state specification. As a consequence there can be states with intersecting or overlapping **-supply_expr** and **-logic_expr** expressions. These states may have different legalities. A power domain or a supply set may be in a state that matches more than one power state. The refinable object for this command is *state_name*.

Multiple power states can be defined and attributed on an object in a single call to this command. simstate and legality can be defined on a per state basis (when specified within **-state**). If **-simstate** or legality are specified external to a **-state** definition then they are applied to any state listed in the command that does not otherwise specify **-simstate** or legality.

If the state has already been defined for this object, and **-update** is specified with either **-supply_expr** or **-logic_expr** then the *boolean_function* of **-supply_expr** or **-logic_expr** is concatenated with the existing state definition via a logical `AND`ing of this expression with the existing definition:

logic_expr' = (*previous logic_expr*) && (**logic_expr**)

supply_expr' = (*previous supply_expr*) && (**supply_expr**)

A logical contradiction exists when a logic or supply net is specified to be exactly more than one value for the state, e.g., `(enable == '1')` and `(enable == '0')`. A power state specification is *erroneous* if it contains logical contradiction(s).

The **-logic_expr** *boolean_function* shall be a SystemVerilog Boolean expression (see 4.10). A supply set or power domain name may be referenced as one operand with the name of a power state attributed on the supply set or power domain as the other operand in an equality or inequality subexpression of **-logic_expr**. For example:

```
-logic_expr {core_pd == turbo && ram_pd != sleep}
```

The **-logic_expr** *boolean_function* expression may contain a reference to an **interval**(*signal_name* [*edge edge*]). The **interval** function returns the time difference between two edges of *signal_name*. *edge* shall be one of **posedge** or **negedge**. If the edges are not specified, **posedge** is the default for both. If the second *edge* specification is identical to the first *edge* specification, the interval is the time measured between the occurrence to two successive identical *edge*s. The value returned by **interval** shall be compared against a SystemVerilog time literal value. At the beginning of simulation prior to the occurrence of the *edge*s, the **interval** function shall return a value larger than any time literal that can be compared with it.

The **-supply_expr** *boolean* shall be a SystemVerilog expression referencing supply nets and supply ports. The supply net type value can be expressed as an aggregate value in the following ways:

— *supply_net* == *state*
This syntactic form is not standard SystemVerilog, but it allows a convenient way to specify the condition is true when *supply_net*.state = *state*. (*supply_net*.voltage can be any value.)

— *supply_net* == ‘{*state*, ‘{*voltage_1* [, *voltage_2* [, *voltage_3*]]}}
This syntactic form is not standard SystemVerilog, but it allows a single method (see 5.4.3.2) to specify either a single voltage value, a voltage range when two values are specified, or a min-nom-max tuple when three values are specified. When only *voltage_1* is specified, *min*, *nom*, and *max* are set to that value. If both *voltage_1* and *voltage_2* are specified, *min* is set to *voltage_1*, *max* is set to *voltage_2*, and *nom* is set to (*voltage_1* + *voltage_2*)/2. When all three values are specified, *min* is set to *voltage_1*, *nom* is set to *voltage_2*, and *max* is set to *voltage_3*. The **-supply_expr** *boolean* evaluates *True* when the *supply_net*.state = *state* and *voltage_1* <= *supply_net*.voltage <= *voltage_3*.

It shall be an error if *voltage_2* < *voltage_1* or *voltage_3* < *voltage_2*.

NOTE 1—Simulation and timing verification can differentiate power states based on differences in voltage. It is unknown if implementation and formal verification tools can differentiate power states based on voltage differences.

-simstate shall only be specified for power states attributed on a supply set. It shall be an error if the power state is associated with any other type of object (see 6.51). See also 5.6.

When **-simstate**:

— Is declared for a named state the first time, any of the arguments are legal.

— Is declared **NOT_NORMAL**, the effect shall be the same as if **CORRUPT** had been declared.

— Has previously been declared **NOT_NORMAL**, the definition may be subsequently refined to any simstate other than **NORMAL**.

— Has previously been declared **NORMAL**, **CORRUPT**, **CORRUPT_ON_ACTIVITY**, **CORRUPT_STATE_ON_CHANGE**, or **CORRUPT_STATE_ON_ACTIVITY**, it shall be an error to specify any state other than that originally declared (e.g., once **CORRUPT** has been declared for a particular state, it needs to remain as **CORRUPT** in any subsequent declarations for that state).

Legality specifies if it is permissible for the object to be in that state. Undefined states are illegal unless specified otherwise. Undefined states of a particular object are legal if **add_power_state** is invoked for that object without specification of **-illegal** or **-state**. Defined states are legal unless **-illegal** is specified.

If neither **-legal** or **-illegal** are specified in an **add_power_state** command, the default is **-legal**.

a) **-legal** means this state is legal and the object may be in that state.

b) **-illegal** means the object shall never be in that state.

Verification tools shall emit an error when an illegal state occurs.

NOTE 2—The choice of state name has no simstate implications.

NOTE 3—Implementation tools may optimize a design based on the presumption illegal states never occur.

Syntax examples:

```
add_power_state PdA.primary
-state GO_MODE {-logic_expr SW_ON -simstate NORMAL
    -supply_expr {{power == `{FULL_ON, 0.8}}
    && {ground == `{FULL_ON, 0}}
    && {nwell == `{FULL_ON,0.8}}}
-state OFF_MODE {-logic_expr !SW_ON
    -supply_expr {power == `{OFF}}
    -simstate CORRUPT}
-state SLEEP_MODE {-logic_expr {SW_ON && interval(clk_dyn posedge negedge)
    >= 100ns}
    -supply_expr {{power == `{FULL_ON, 0.8}}
    && {ground == `{FULL_ON, 0}}
    && {nwell == `{FULL_ON,0.6}}}
    -simstate CORRUPT_STATE_ON_CHANGE}
-legal

add_power_state PdTOP -legal
    -state GOGO {-logic_expr {u1/PdA == GO_MODE}}

add_power_state GARY -legal
```

6.9 add_pst_state

Purpose	Define the states of each of the supply nets for one possible state of the design	
Syntax	**add_pst_state** *state_name* **-pst** *table_name* **-state** *supply_states*	
Arguments	*state_name*	The simple name of the state being defined.
	-pst *pst_name*	The power state table (PST) to which this state applies.
	-state *supply_states*	The list of supply net state names (see 6.22), listed in the corresponding order of the **-supplies** listing in the **create_pst** command (see 6.21). A ***** in place of a state name indicates this is a "don't care" for that supply.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

The **add_pst_state** command defines the name for a specific state of the supply nets defined for the PST *table_name*.

It shall be an error if

— The number of *supply_states* is different than the number of supply nets within the PST.

— A *state_name* is defined more than once for the same PST.

Syntax example:

```
create_pst            pt -supplies { PN1   PN2   SOC/OTC/PN3 }
add_pst_state s1 -pst pt -state    { s08   s08   s08         }
add_pst_state s2 -pst pt -state    { s08   s08   off         }
add_pst_state s3 -pst pt -state    { s08   s09   off         }
```

6.10 associate_supply_set

Purpose	Associate a supply set or *supply_set_ref* to a power domain, power switch, or strategy *supply_set_handle*	
Syntax	**associate_supply_set** *supply_set_ref* **-handle** *supply_set_handle*	
Arguments	*supply_set_ref*	The rooted name of the supply set or a *supply_set_handle* to associate.
	-handle *supply_set_handle*	The *supply_set_handle*.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

The **associate_supply_set** command associates a supply set with a domain or strategy. The supply set includes the specific nets for each of the supplies (**power**, **ground**, etc.), see 6.24. See also 6.14.

A supply set handle is constructed from a domain name or a strategy name. Handles are names rooted in the active scope. This allows the association of a power domain supply set with a supply set defined in a different scope by first setting the active scope to one in which both the supply set and the domain or strategy are visible.

Predefined *supply_set_handle*s for the power domain [*domain_name* (see 6.19)] include: *domain_name*.**primary**, *domain_name*.**default_retention**, and *domain_name*.**default_isolation**. If **default_retention** or **default_isolation** are specified on a domain, this sets the default retention or isolation supply set, respectively, to use in the retention or isolation strategies of that domain when no other supply set is specified.

a) The predefined *supply_set_handle*s for a level-shifter strategy [*level_shifter_name* (see 6.42)] are *domain_name*.*level_shifter_name*.**input** and *domain_name*.*level_shifter_name*.**output**.

b) The predefined *supply_set_handle* for a power switch [*switch_name* (see 6.20)] is *switch_name*.**supply**, e.g.,

```
create_power_switch purple_people_eater_ps
    -supply_set yellow_meanie
```

which is equivalent to

```
create_power_switch purple_people_eater_ps
associate_supply_set yellow_meanie
    -handle purple_people_eater_ps.supply
```


This power switch's supply set handle is

```
purple_people_eater_ps.supply
```

which references `yellow_meanie`, i.e.,

```
purple_people_eater_ps.yellow_meanie
```

c) The predefined *supply_set_handle* for a retention strategy [*retention_name* (see 6.47)] is *switch_name*.*retention_name*.**supply**.

d) User-specified names for *supply_set_handle* are also permitted. The *supply_set_handle* is created in the scope of the domain, switch, or strategy and the *supply_set_ref* is associated with the *supply_set_handle*.

e) For predefined *supply_set_handle*s, the *supply_set_ref* is associated with the predefined *supply_set_handle*.

f) The form of *supply_set_handle*s for the isolation cell strategies (*domain_name*.*isolation_name*.**isolation_supply_set[***index***]** (see 6.20 and 6.40), where *index starts* at `0`) are *domain_name*.*isolation_name*.**isolation_supply_set[0]**, *domain_name*.*isolation_name*.**isolation_supply_set[1]**, and so on. A name may be associated with a vector positions using the **associate_supply_set** command, e.g.,

```
associate_supply_set U1/PD1.my_iso.isolation_supply_set\[1\]
   -handle U1/PD1.my_iso.clamp
```

g) When specifying a supply net in a supply set, a handle of the form *switch_name*.*strategy*.*supply*.*function* may be used. The names are defined in the scope of the domain, switch, or strategy.

It shall be an error if

— A *supply_set_handle* is already associated with a *supply_set*.

— The associations of handles to supply sets form a loop of associations.

Syntax example:

```
associate_supply_set some_supply_set
   -handle U1/PD1.rolf_mem_ss
```

6.11 bind_checker

Purpose	Insert checker modules and bind them to design elements	
Syntax	**bind_checker** *instance_name* **-module** *checker_name* [**-elements** *element_list*] [**-bind_to** *module* [**-arch** *name*]] [**-ports** {{*port_name net_name*}*}]	
Arguments	*instance_name*	The name used to instance the checker module in each design element.
	-module *checker_name*	The name of a SystemVerilog module containing the verification code. The verification code itself shall be written in SystemVerilog, but it can be bound to either a SystemVerilog or VHDL instance.
	-elements *element_list*	The list of design elements.
	-bind_to *module* [**-arch** *name*]	The SystemVerilog module or VHDL entity/architecture for which all instances are the target of this command.
	-ports {{*port_name net_name*}*}	The association of signals to the checker's ports.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

The **bind_checker** command inserts checker modules into a design without modifying the design code or introducing functional changes. The mechanism for binding the checkers to design elements relies on the SystemVerilog `bind` directive. The `bind` directive causes one module to be instantiated within another, without having to explicitly alter the code of either. This facilitates the complete separation between the design implementation and any associated verification code.

Signals in the target instance are bound by position to inputs in the bound checker module through the port list. Thus, the bound module has access to any and all signals in the scope of the target instance, by simply adding them to the port list, which facilitates sampling of arbitrary design signals.

If **-bind_to** is specified, an instance of checker is created in every instance of the module. Otherwise, an instance of the checker is only created within the active scope.

port_name is a port defined on the interface of *checker_name* and *net_name* is a name of a net relative to the scope where *checker_name* is being instantiated.

It shall be an error if *instance_name* already exists in **-bind_to** *module*.

This command is for verification only; implementation tools shall ignore it.

Syntax example:

```
bind_checker chk_p_clks
-module assert_partial_clk
-bind_to aars
-ports {{prt1 clknet2} {port3 net4}}
```


6.12 connect_logic_net

Purpose	Connect a logic net to logic ports	
Syntax	**connect_logic_net** *net_name* **-ports** *port_list*	
Arguments	*net_name*	A simple name.
	-ports *port_list*	A list of ports on the interface of the active scope and/or on design elements that are located in the active scope and its descendants.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

The **connect_logic_net** command connects a logic net to the specified ports. The net is propagated through implicitly created ports and nets throughout the logic hierarchy in the descendant tree of the active UPF scope as required (see 4.2). The connection from *net_name* in the active UPF scope to any element in *port_list* shall not cross any power-domain boundaries.

The net and ports shall be of a compatible type. The following HDL types are compatible with each other:

— SystemVerilog `logic`
— VHDL `std_ulogic`
— SystemC `sc_bit`

It shall be an error if *net_name* does not already exist.

NOTE 1—Use **create_logic_port** (see 6.18) to create new logic ports on power domain boundaries.

NOTE 2—This command exists to allow for the propagation of signals from a power-management block. Using this command to provide non-power control connections may cause the logic function to diverge from the HDL and is strongly discouraged.

Syntax example:

```
connect_logic_net sab_chair
  -ports {s b}
```

6.13 connect_supply_net

Purpose	Connect a supply net to supply ports	
Syntax	**connect_supply_net** *net_name* [**-ports** *list*] [**-pg_type** {*pg_type_list element_list*}]* [**-vct** *vct_name*] [**-pins** *list*] [**-cells** *list*] [**-domain** *domain_name*] [**-rail_connection** *rail_type*]	
Arguments	*net_name*	A simple name.
	-ports *list*	A list of rooted port names.
	-pg_type {*pg_type_list element_list*}	An indirect connection specification via the *pg_type* on the *design_element*'s ports.
	-vct *vct_name*	A value conversion table (VCT) defining how values are mapped from UPF to an HDL model or from the HDL model to UPF.
	-pins *list*	A list of pins on cells to connect.
	-cells *list*	A list of cells to use for **-rail_connection** or **-pg_type**.
	-domain *domain_name*	The domain indicates the scope to use for **-rail_connection** or **-pg_type**.
	-rail_connection *rail_type*	The rail type (for older libraries).
Return value	Return the fully qualified name of the supply net or raise a `TCL_ERROR` if the supply net is not created.	

The **connect_supply_net** command connects a supply net to the specified ports. The net is propagated through implicitly created ports and nets throughout the logic hierarchy in the descendant tree of the active UPF scope as required (see 4.2) This explicit connection overrides (has higher precedence than) the implicit and automatic connection semantics (see 4.2) that might otherwise apply. **-domain** or **-cells** is required when the **-rail_connection** or **-pg_type** options are specified.

Use:

-ports to connect to supply ports;

-pins to connect to pins on library cells;

-cells to connect to all pins of the appropriate type (power or ground) on the specified cells;

-rail_connection to connect to pins having this (rail) type; only use this if the **-cells** or **-domain** options are specified;

-pg_type to connect to ports on the design elements that have the specified *pg_type*;

-vct to indicate for every HDL port to which the port is connected, the supply net state shall be converted if it is being propagated into the HDL port (see 6.25) or the HDL port value shall be converted if it is being propagated onto the supply net (6.16). **-vct** is ignored for any connections of the supply net to supply ports defined in UPF.

The following also apply:

- It shall be an error if any cell, domain, port, supply net, or design element specified in this command does not exist.
- It shall be an error if the value conversions specified in the VCT do not match the type of the HDL port.
- If shall be an error if at least one of **-ports**, **-pins**, **-rail_connection**, or **-pg_type** is not specified in a **connect_supply_net** command.
- The **-ports** and **-pins** options are mutually exclusive with the **-cells**, **-domain**, **-rail_connection**, and **-pg_type** options.
- The **-rail_connection** and **-pg_type** options are mutually exclusive with each other.
- Automatic propagation of a supply net throughout the extent of a power domain is determined by its usage within the domain, such as primary supply, retention supply, etc.
- It shall be an error if *net_name* has not been previously created; in this case, a `0` shall be returned.
- If **-pg_type** is specified, it shall be an error if a design element does not exist or the specified attribute does not exist on any port of the design element.

Syntax examples:

```
connect_supply_net fb
  -ports {jk jb}

connect_supply_net mc
  -ports {rl}
  -vct SV_TIED_HI
```

6.14 connect_supply_set

Purpose	Connect a supply set to particular elements	
Syntax	**connect_supply_set** *supply_set_ref* {**-connect** {*supply_function* {*pg_type_list*}}}* [**-elements** *element_list*] [**-exclude_elements** *exclude_list*] [**-transitive** <**TRUE** | **FALSE**>]	
Arguments	*supply_set_ref*	The rooted name of the supply set or a *supply_set_handle*.
	-connect {*supply_function* {*pg_type_list*}}	Define automatic connectivity for a *supply_function* of the *supply_set_ref* as ports having the specified *pg_type_list* attributes (see 6.13).
	-elements *element_list*	The list of design element names to add.
	-exclude_elements *exclude_list*	The list of design elements to exclude from the *effective_element_list*.
	-transitive <**TRUE** | **FALSE**>	When **-transitive** is **TRUE** (the default), the command applies to the descendants of the elements.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

The **connect_supply_set** command connects a supply set to the specified elements. The nets of the set are propagated through implicitly created ports and nets throughout the logic hierarchy in the descendant tree of

the active UPF scope as required (see 4.2) This explicit connection overrides (has higher precedence than) the implicit and automatic connection semantics (see 4.2) that might otherwise apply.

This command applies to elements in the *effective_element_list* (see 6.3).

a) When *supply_set_ref* refers to a handle associated with a domain, the *prefiltered_element_list* is filtered to only include elements within the extent of the domain.

b) When *supply_set_ref* refers to a handle associated with a strategy, the *prefiltered_element_list* is filtered to only include all elements connected to the strategy's supply.

c) When *supply_set_ref* refers to a handle associated with a domain and the *aggregate_element_list* is empty, all elements in the extent of the domain are added to the *aggregate_element_list*.

d) When *supply_set_ref* refers to a handle associated with a strategy and the *aggregate_element_list* is empty, all elements connected to the respective strategy supply are added to the *aggregate_element_list*.

-connect is additive, i.e., on a particular supply function, a subsequent invocation setting *pg_type_list* adds the additional *pg_type_list*.

NOTE—The *exclude_list* in **-exclude_elements** can specify elements that have not already been explicitly or implicitly specified via an explicit or implied *element_list*.

It shall be an error if

— A particular *pg_type_list* is associated with more than one supply net for any given design element in **-connect**;

— More than one supply net is connected to the same port in a design element, even if the connection is the result of more than one command that connects supply nets, e.g, **connect_supply_set**, **connect_supply_net**, etc.

— Any element of *element_list* or *exclude_list* is not in a specified domain or strategy referenced in the *supply_set_handle*.

Syntax example:

```
connect_supply_set some_supply_set
    -elements {U1/U_minh_mem}
    -connect {mem_array_power {rolf_pg_type minh_pg_type}}
    -connect {power {primary_power}}
    -connect {ground {primary_ground}}
```


6.15 create_composite_domain

Purpose	Define a composite domain comprised of one or more subdomains		
Syntax	**create_composite_domain** *composite_domain_name* [**-subdomains** *subdomain_list*] [**-supply** {*supply_set_handle* [*supply_set_ref*]}]* [**-update**]		
Arguments	*composite_domain_name*	The name of the composite domain; this shall be a simple name.	
	-subdomains *subdomain_list*	The **-subdomains** option specifies a list of rooted domain names, including any previously created composite domains.	**R**
	-supply {*supply_set_handle* [*supply_set_ref*]}	The **-supply** option specifies the *supply_set_handle* for *composite_domain_name*. If *supply_set_ref* is also specified, the domain *supply_set_handle* is associated with the specified *supply_set_ref*. The *supply_set_ref* may be any supply set or *supply_set_handle* visible in the active scope. The predefined *supply_set_handle*s are: **primary**, **default_retention**, and **default_isolation**. See also 6.10.	**R**
	-update	Use **-update** if the *composite_domain_name* has already been defined.	
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.		

A *composite power domain* is a simple container for a set of power domains. Operations performed on the composite domain are transitively applied to each subdomain, e.g., creating a *supply_set_handle*, associating a supply set to the domain, or defining a strategy.

A composite domain has no attributes except power states and the primary *supply_set_handle*. A command applied to a composite domain is transitively applied to each subdomain if and only if the application of that command does not result in an error in any subdomain. It is valid to refer to the primary supply of a composite domain because there is exactly one primary supply common to all subdomains. It is not valid to refer to other *supply_set_handle*s or strategies in the composite domain because they are not necessarily common to all sub domains.

a) Composite power domains can be used as a subdomain of other composite power domains.

b) Since a composite domain is simply a container, commands can still be applied to subdomains after merging.

c) For each subdomain before merging: If a supply set is associated with the primary *supply_set_handle* of a subdomain, that supply set shall be the same as the supply set of the composite domain.

d) The following restrictions apply to the composite domain and subdomains after the merge has been completed:

   1) It shall be an error to associate a supply set to the primary *supply_set_handle* of a subdomain.

   2) Commands applied to a subdomain do not affect any other subdomain or the composite domain.

e) Subdomains of a composite domain can still be referenced after merging, in the sense their elements lists are valid after merging, and all aspects of the subdomain (e.g., strategies defined on them) can be referenced.

A domain *supply_set_handle* may be defined without an association to a *supply_set_ref* (see 6.19). The association can be completed separately (see 6.10).

When a *supply_set_handle* and a *supply_set_ref* are specified in **-supply**, it is equivalent to the following:

```
associate_supply_set supply_set_ref
   -handle composite_domain_name.supply_set_handle
```

If **default_retention** or **default_isolation** are specified on **-supply**, this specifies the default retention or isolation supply set, respectively, to use in the retention or isolation strategies of that domain when no other supply set is specified. User-defined *supply_set_handle*s are also permitted.

Syntax example:

```
create_composite_domain my_combo_domain_name
   -subdomains {a/pd1 b/pd2}
   -supply {default_isolation could_be_on_ss}
```

6.16 create_hdl2upf_vct

Purpose	Define a value conversion table that can be used in converting HDL logic values into `state` type values	
Syntax	**create_hdl2upf_vct** *vct_name* **-hdl_type** {<**vhdl** | **sv**> [*typename*]} **-table** {{*from_value to_value*}*}	
Arguments	*vct_name*	The value conversion table (VCT) name.
	-hdl_type {<**vhdl** | **sv**> [*typename*]}	The HDL type for which the value conversions are defined.
	-table {{*from_value to_value*}*}	A list of the values of the HDL type to map to UPF `state` type values.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

The **create_hdl2upf_vct** command defines a value conversion table from an HDL logic type to the `state` type of the supply net value (see Annex B) when that value is propagated from HDL port to a UPF supply net. It shall provide a conversion for each possible logic value that the HDL port can have. **create_upf2hdl_vct** does not check that the set of HDL values are complete or compatible with any HDL port type.

vct_name provides a name for the value conversion table for later use with the **connect_supply_net** command (see 6.13). There is a single global name space for *vct_name*s. The predefined VCTs are shown in Annex C.

-hdl_type specifies the HDL type for which the value conversions are defined. This information allows a tool to provide completeness and compatibility checks. If the *typename* is not one of the language's predefined types or one of the types specified in the next paragraph, then it shall be of the form *library*.*pkg*.*type*.

The following HDL types shall be the minimum set of types supported. An implementation tool may support additional HDL types:

a) VHDL

   1) `Bit`, `std_[u]logic`, `Boolean`

   2) Subtypes of `std_[u]logic`

b) SystemVerilog

   `reg`/`wire`, `Bit`, `Logic`

-table defines the 1:1 conversion from HDL logic value to the UPF partially on and on/off states. The values are consistent with the HDL type values.

For example:

— When converting from SystemVerilog *logic type*, the legal values are `0`, `1`, `X`, and `Z`.

— When converting from SystemVerilog or VHDL `bit`, the legal values are `0` or `1`.

— When converting from VHDL `std_[u]logic`, the legal values are `U`, `X`, `0`, `1`, `Z`, `W`, `L`, `H`, and `-`.

The conversion values have no semantic meaning in UPF. The meaning of the conversion value is relevant to the HDL model to which the supply net is connected.

It shall be an error if *vct_name* already exists in the active UPF scope.

Syntax examples:

```
create_hdl2upf_vct
   vlog2upf_vss
   -hdl_type {sv reg}
   -table {{X OFF} {0 FULL_ON} {1 OFF} {Z PARTIAL_ON}}
create_hdl2upf_vct
   stdlogic2upf_vss
   -hdl_type {vhdl std_logic}
   -table {{'U' OFF}
       {'X' OFF}
       {'0' OFF}
       {'1' FULL_ON}
       {'Z' PARTIAL_ON}
       {'W' OFF}
       {'L' OFF}
       {'H' FULL_ON}
       {'-' OFF}}
```

6.17 create_logic_net

Purpose	Define a logic net
Syntax	**create_logic_net** *net_name*
Arguments	*net_name* A simple name.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.

The **create_logic_net** command creates a logic net in the active scope.

The net's type is determined by the language of the scope where it is created. If the scope is

- SystemVerilog, the type is `logic`
- VHDL, the type is `std_ulogic`
- SystemC, the type is `sc_bit`

It shall be an error if *net_name* already exists in the active scope.

Syntax example:

```
create_logic_net sab_chair
```

6.18 create_logic_port

Purpose	Define a logic port
Syntax	**create_logic_port** *port_name* [**-direction** <**in** | **out** | **inout**>]
Arguments	*port_name* A simple name. **-direction** <**in** | **out** | **inout**> The direction of the port. The default is **in**.
Return value	Return the fully qualified name (from the active scope) of the created port or raise a `TCL_ERROR` if the port is not created.

The **create_logic_port** command creates a logic port on the active scope. This logic port is not included in any existing strategies for isolation or level-shifting. However, any isolation or level-shifting strategy defined after the logic port is created shall apply to the port if it otherwise matches other criteria specified in the strategy.

The port's type is determined by the language of the scope where it is created. If the scope is

- SystemVerilog, the type is `logic`
- VHDL, the type is `std_ulogic`
- SystemC, the type is `sc_bit`

This command can be applied once to a logic port.

Syntax example:

```
create_logic_port jd_writer
-direction out
```

6.19 create_power_domain

Purpose	Define a set of design elements that generally have the same primary supply set		
Syntax	**create_power_domain** *domain_name* [**-simulation_only**] [**-elements** *element_list*] [**-exclude_elements** *exclude_list*] [**-include_scope**] [**-supply** {*supply_set_handle* [*supply_set_ref*]}*] [**-scope** *instance_name*] [**-define_func_type** {*supply_function* {*pg_type_list*}}]* [**-update**]		
Arguments	*domain_name*	The name of the power domain; this shall be a simple (non-hierarchical) name. This simple name is rooted in the active scope.	
	-simulation_only	Define a power domain for simulation purposes only.	
	-elements *element_list*	The list of design elements to add.	**R**
	-exclude_elements *exclude_list*	The list of design elements to exclude from the *effective_element_list*.	**R**
	-include_scope	Define the extent of the domain to include the active scope and, by default, all of its descendant scopes. See also 4.7.	
	-supply {*supply_set_handle* [*supply_set_ref*]}	The **-supply** option specifies the *supply_set_handle* for *domain_name*. If *supply_set_ref* is also specified, the domain *supply_set_handle* is associated with the specified *supply_set_ref*. The *supply_set_ref* may be any supply set or *supply_set_handle* visible in the active scope. The predefined *supply_set_handle*s are: **primary**, **default_retention**, and **default_isolation**. See also 6.10.	**R**
	-scope *instance_name*	Create the power domain within this scope.	
	-define_func_type {*supply_function* {*pg_type_list*}}	Define automatic connectivity for a *supply_function* of *domain_name***.primary** (see 6.10) having the specified attributes in *pg_type_list*.	**R**
	-update	Use **-update** if the *domain_name* has already been defined.	
Return value	Return the fully qualified name (from the active scope) of the created domain or raise a `TCL_ERROR` if the domain is not created.		

In addition to defining a set of design elements that share a common primary supply set, **create_power_domain** can be used to specify additional supply sets (see 6.10) used within the domain, and any isolation, level-shifting, or retention strategies. Elements and supplies may be added incrementally.

-elements specifies a set of rooted design elements contained within the power domain. This command presumes **-transitive** is always **TRUE** (see 6.3.1). The following also apply:

— *element_list* is specified respective to the active scope; it is not influenced by the **-scope** argument.

— If **-include_scope** is specified, the elements in the active scope are included as a rooted design element in the *aggregate_element_list*.

— When **-elements** and **-update** are not specified, the active scope is included as a rooted design element in the *aggregate_element_list*.

— A design element can be a member of only one power domain.

— When **-simulation_only** is specified, signal names and process labels may also be specified in *list*. **-simulation_only** specifies the domain is intended for use with behavioral non-synthesizing elements.

NOTE—When **-elements** is specified with an empty list, no elements are added to the *effective_element_list*.

A domain *supply_set_handle* may be defined without an association to a *supply_set_ref*. The association can be completed separately (see 6.10).

When a *supply_set_handle* and a *supply_set_ref* are specified in **-supply**, it is equivalent to the following:

```
associate_supply_set supply_set_ref
   -handle domain_name.supply_set_handle
```

If **default_retention** or **default_isolation** is specified as the *supply_set_handle* with **-supply**, this specifies the default retention or isolation supply set, respectively, to use in the retention or isolation strategies of that domain when no other supply set is specified. User-defined *supply_set_handle*s are also permitted.

The primary supply set is implicitly connected to design elements and logic inferred from the design elements in the power domain. However, the primary supply set shall not be implicitly connected when any of the following apply:

a) A design element has at least one supply net explicitly or automatically connected and **set_simstate_behavior** (see 6.51) has not been enabled.

b) A design element has **set_simstate_behavior** disabled.

c) A design element is created as a result of a UPF command, e.g., isolation cells, level-shifters, power switches, and the retention portion of a retention register.

Implicit connections imply simulation semantics as specified in 5.4.2.

-scope specifies the scope, i.e., the instance, where the domain shall be created. This scope is the active scope for this command; it defines the domain boundary within the logic design. If **-scope** is not specified, the power domain shall be created within the active scope.

-update may be used to add elements and supplies to a previously created domain. It shall be an error if **-update** is used during the initial creation of *domain_name*.

It shall be an error if an implementation tool encounters a **-simulation_only** power domain.

Syntax example:

```
create_power_domain PD1 -elements {top/U1}
-supply {primary PD1_Primary}
-supply {isolation PD1_ret}
-supply {retention PD1_ret}
-supply {mem_array PD1_ma}
set_scope /top/U1
create_power_domain PD2 -elements {}
```


6.20 create_power_switch

<table>
<tr><td>Purpose</td><td colspan="2">Define a switch</td></tr>
<tr><td>Syntax</td><td colspan="2">create_power_switch switch_name
  -output_supply_port {port_name [supply_net_name]}
  {-input_supply_port {port_name [supply_net_name]}}*
  {-control_port {port_name [net_name]}}*
  {-on_state {state_name input_supply_port {boolean_function}}}*
  [-off_state {state_name {boolean_function}}]*
  [-supply_set supply_set_name]
  [-on_partial_state {state_name input_supply_port {boolean_function}}]*
  [-ack_port {port_name net_name [{boolean_function}]}]*
  [-ack_delay {port_name delay}]*
  [-error_state {state_name {boolean_function}}]*
  [-domain domain_name]</td></tr>
<tr><td rowspan="11">Arguments</td><td>switch_name</td><td>The name of the switch instance to create; this shall be a simple name.</td></tr>
<tr><td>-output_supply_port {port_name [supply_net_name]}</td><td>The output supply port of the switch and, optionally, the net where this port connects.</td></tr>
<tr><td>-input_supply_port {port_name [supply_net_name]}</td><td>An input supply port of the switch and, optionally, the net where this port is connected.</td></tr>
<tr><td>-control_port {port_name [net_name]}</td><td>A control port on the switch and, optionally, the net where this control port connects.</td></tr>
<tr><td>-on_state {state_name input_supply_port {boolean_function}}</td><td>A named state, the input_supply_port for which this is defined, and its corresponding Boolean function.</td></tr>
<tr><td>-off_state {state_name {boolean_function}}</td><td>A named state and its corresponding Boolean function.</td></tr>
<tr><td>-supply_set supply_set_name</td><td>Associate a supply set with a switch.</td></tr>
<tr><td>-on_partial_state {state_name input_supply_port {boolean_function}}</td><td>A named state, the input_supply_port for which this is defined, and its corresponding Boolean function where the switch is in a current-limited state.</td></tr>
<tr><td>-ack_port {port_name net_name [{boolean_function}]}</td><td>The acknowledge port on the switch and the logic net where this port connects. A Boolean function can also be specified. If a null string is used as the net_name for -ack_port, the port and its output function are defined, but the port itself is unconnected.</td></tr>
<tr><td>-ack_delay {port_name delay}</td><td>The acknowledge port and delay on the switch where this port connects.</td></tr>
<tr><td>-error_state {state_name {boolean_function}}</td><td>Any error states, which if defined on the switch can be flagged during simulation or analysis.</td></tr>
<tr><td>-domain domain_name</td><td>If specified, the scope of the domain is the scope of the switch instance.</td></tr>
<tr><td>Return value</td><td colspan="2">Return the fully qualified name of the created switch or raise a <code>TCL_ERROR</code> if the switch is not created.</td></tr>
</table>

The **create_power_switch** command defines an instance of a power switch in the active scope or the scope of the **-domain** argument when provided.

An input supply port without a connected supply net has the value **UNDETERMINED**.

The switch's **-on_state**, **-on_partial_state**, **-off_state**, and **-error_state** are evaluated in the following order to determine the state of the output port:

a) When any **-error_state** evaluates to *True,* the state of the switch's output port is set to **UNDETERMINED** and the value of the voltage is unspecified.

b) When any **-off_state** evaluates to *True*, and any **-on_state** or **-on_partial_state** evaluates to *True,* the state of the switch's output port is set to **UNDETERMINED** and the value of the voltage is unspecified.

c) When more than one **-on_state** evaluates to *True* and the root supply drivers of the respective **-input_supply_port**s are not all the same: the state of the switch's output port is set to **UNDETERMINED**, the value of the voltage is unspecified, and it shall be an error.

d) When more than one **-on_state** evaluates to *True* and the root supply drivers of the respective **-input_supply_port**s are the same, the voltage of the output supply port is set to the voltage of the root supply driver and if any input supply port corresponding to an **-on_state** that evaluates to *True* is **PARTIAL_ON**, the output supply port's state is set to **PARTIAL_ON**; otherwise, the root supply driver is propagated to the output supply port.

e) When an **-on_state** evaluates to *True*, the value on the input supply port for that **-on_state** is propagated to the output supply port.

f) When **-off_state** has a null Boolean expression, it can not evaluate *True*; therefore, when no **-on_state** or **-on_partial_state** evaluates *True*, the state of the switch's output port is set to **UNDETERMINED**.

g) When any **-off_state** evaluate to *True*, the supply port's state is set to **OFF** and the value of the voltage is unspecified.

h) When more than one **-on_partial_state** evaluates to *True* and the root supply drivers of the respective **-input_supply_port**s are not all the same: the state of the switch's output port is set to **UNDETERMINED**, the value of the voltage is unspecified, and it shall be an error.

i) When more than one **-on_partial_state** evaluates to *True* and the root supply drivers of the respective **-input_supply_port**s are the same, the value of the root supply driver is propagated to the output supply port and degraded to **PARTIAL_ON** if it is **FULL_ON**.

j) When an **-on_partial_state** evaluates to *True*, the value on the input supply port for that **-on_partial_state** is propagated to the output supply port and degraded to **PARTIAL_ON** if the input is **FULL_ON**.

k) When all **-on_state**s and all **-on_partial_state**s evaluate to *False* and no **-off_state** is defined, the output supply port's state is set to **OFF**.

l) When all **-on_state**s, **-on_partial_state**s, **-off_state**s, and **-error_state**s evaluate to *False* and the state of all input supply ports are **OFF**, the output supply port's state is set to **UNDETERMINED**.

m) Otherwise; the state of the switch's output port is set to **UNDETERMINED** and the value of the voltage is unspecified.

An anonymous root supply driver originates the state of the output supply port when the state of the output supply port is set to **UNDETERMINED** or **OFF** as a result of the evaluating of the switch's **-control_port** [see a), b), c), f), g), h), k), l), and m)].

If a *boolean_function* is specified for **-ack_port**, the result of *boolean_function* is driven on **-ack_port**'s *port_name delay* time units after a control port transition. Otherwise, a logic `1` shall be driven on the *port_name delay* time units after an **-on_state** evaluates to *True* and a logic `0` shall be driven *delay* time units after an **-off_state** evaluates to *True*. *delay* (the default is `0`) may be specified as a unit-less natural

integer or as a SystemVerilog time unit. If specified as a natural integer, the time unit shall be the same as the simulation precision.

Any **-ack_port**, **-on_state**, **-on_partial_state**, **-off_state**, or **-error_state** *boolean_function* shall be a SystemVerilog Boolean expression (see 4.10).

If **-supply_set** is specified for a switch, it powers logic or timing control circuitry within the switch and powers any specified **-ack_port**s. When the supply set simstate is anything other than **NORMAL**, the state of the output supply port of a switch is **UNDETERMINED** and the acknowledge ports are corrupted. If a supply set is not associated with a switch, the output of the supply port implicitly operates in a **NORMAL** simstate and it shall be an error if any acknowledge ports are specified.

The following also apply:

— It shall be an error if *switch_name* already exists in the active scope.

— It shall be an error if **-on_state** has a null Boolean function.

— Any name in a *boolean_function* needs to refer to a control port of the switch.

— All states not covered by the states on, off, and error are anonymous error states.

— If the implementation of a switch can not be inferred, **map_power_switch** (see 6.32) can be used to specify it.

— If *net_name* is not specified for any of the switch's port definitions, **connect_logic_net** (see 6.12) or **connect_supply_net** (see 6.13) can be used to create the port connections.

— Each state name shall be unique for a particular switch.

— Any *port_name*s specified in this command are user-defined (e.g., `my_dogs_name`).

NOTE 1—**create_power_switch** can be used to define an abstract power switch that implementation tools may expand into multiple switches. **create_power_switch** can also be used to specify the need for a specific switch that can then be mapped to a specific switch implementation using **map_power_switch**. It is not meant to be used as a single definition representing multiple physical switches to be mapped with **map_power_switch**.

NOTE 2—**create_power_switch** provides relatively simple, general abstract functionality. HDLs can be used to model switch functionality that cannot be captured with **create_power_switch**.

Syntax example:

```
create_power_switch sw1
-output_supply_port {vout VN3}
-input_supply_port {vin1 VN1}
-input_supply_port {vin2 VN2}
-control_port {ctrl_small ON1}
-control_port {ctrl_large ON2}
-control_port {ss SUPPLY_SELECT}
-on_partial_state {partial_s1 vin1 {ctrl_small & !ctrl_large & ss}}
-on_state {full_s1 vin1 {ctrl_small & ctrl_large & ss}}
-on_partial_state {partial_s2 vin2 {ctrl_small & !ctrl_large & !ss}}
-on_state {full_s2 vin2 {ctrl_small & ctrl_large & !ss}}
-off_state {not_required {!ctrl_small & !ctrl_large}}
-error_state {no_small {!ctrl_small & ctrl_large}}
```

6.21 create_pst

Purpose	Create a power state table	
Syntax	**create_pst** *table_name* **-supplies** *supply_list*	
Arguments	*table_name*	The power state table name (PST). *table_name* is a simple name in the active scope.
	-supplies *supply_list*	The list of supply nets or ports to include in each power state of the design. The supplies are listed as rooted names in the active scope.
Return value	Return the name of the created PST or raise a `TCL_ERROR` if the PST is not created.	

The **create_pst** command defines a PST name and a set of supply nets for use in **add_pst_state** commands (see 6.9). The PST *table_name* is defined in the namespace of the active scope.

A power state table is used for implementation—specifically for synthesis, analysis, and optimization. It defines the legal combinations of states, i.e., those combinations of states that can exist at the same time during operation of the design.

create_pst can only be used with **add_pst_state** (and vice-versa). This combination and using **add_power_state** (see 6.8) are two methods for specifying power state information. Power state specifications and default state definitions form an exhaustive specification of all of the legal power states of the system.

It shall be an error if

— *table_name* conflicts with any existing name in the namespace of the active scope.

— a specified supply net or supply port specified in *supply_list* does not exist.

Syntax example:

```
create_pst MyPowerStateTable -supplies {PN1 PN2 SOC/OTC/PN3}
```

6.22 create_supply_net

Purpose	Create a supply net	
Syntax	**create_supply_net** *net_name* [**-domain** *domain_name*][**-reuse**] [**-resolve** <**unresolved** | **one_hot** | **parallel** | **parallel_one_hot**>]	
Arguments	*net_name*	A simple name.
	-domain *domain_name*	The domain in whose scope the supply net is to be created.
	-reuse	Extend *net_name* as a supply net within *domain_name*. No new nets are created.
	-resolve <**unresolved** | **one_hot** | **parallel** | **parallel_one_hot**>	A resolution mechanism that determines the state and voltage of the supply net when the net has multiple supply sources (see 6.22.2). If no option is specified, the behavior for resolution is the same as for **unresolved**.
Return value	Return the fully qualified name (from the scope in which the net is created) of the created net or raise a `TCL_ERROR` if the net is not created.	

The **create_supply_net** command creates a supply net. If **-domain** is specified, the net is created in the logic hierarchy in the same scope as *domain_name*. Otherwise, the net is created in the active scope. The net is propagated through implicitly created ports and nets throughout the logic hierarchy in the descendant tree of the scope in which the net is created as required by implicit and automatic connections of supply sets (see 6.19). This command can only be used once per net unless **-reuse** is specified.

The following also apply:

— It shall be an error if *domain_name* does not indicate a previously created power domain.

— When **-reuse** is specified, it shall be an error if *net_name* does not already exist.

— When the parameter for **-resolve** is **unresolved**, the supply net shall have only one source (see 6.22.1). For all other parameters to **-resolve**, the requirements on the drivers and sources of the net are as defined in 6.22.2.

NOTE—Use **set_scope** (see 6.50) to change the scope prior to calling this command to set the active scope to the correct scope for the net.

Syntax example:

```
create_supply_net local_vdd_3
    -resolve one_hot
```

6.22.1 Supply net resolution

Supply nets are often connected to the output of a single switch. However, certain applications, such as on-chip voltage scaling, may require the outputs of multiple switches or other supply drivers to be connected to the same supply net (either directly or via supply port connections). In these cases, a resolution mechanism is needed to determine the state and voltage of the supply net from the state and voltage values supplied by each of the supply drivers to which the net is connected.

A supply net that specifies an **unresolved** resolution cannot be connected to more than one supply source.

6.22.2 Resolutions methods

The following resolution methods shall be provided in the **create_supply_net** command (see 6.22):

a) **unresolved**

The supply net may only be connected to a single supply source (this is the default).

If the supply net has multiple sources, the net shall be resolved to **UNDETERMINED** and the voltage value is unspecified.

b) **one_hot**

Multiple supply sources, each having a unique driver, may be connected to the supply net.

A supply net with **one_hot** resolution has a deterministic state only when no more than one source drives the net at any particular point in time. If at any point in time more than one supply source driving the net is anything other than **OFF**, the state of the supply net shall be **UNDETERMINED**, the voltage value of the supply net shall be unspecified, and implementations may issue a warning or an error.

1) If all supply sources are **OFF**, the state of the supply net shall be **OFF**, and the voltage value of the supply net shall be unspecified.

2) If only one supply source is **FULL_ON** and all other sources are **OFF**, the state of the supply net shall be **FULL_ON**, and the voltage value of the corresponding source shall be assigned to the supply net.

3) If only one supply source is **PARTIAL_ON** and all other sources are **OFF**, the state of the supply net shall be **PARTIAL_ON** and the voltage value of the corresponding source shall be assigned to the supply net.

4) If any source is **UNDETERMINED**, the state of the supply net shall be **UNDETERMINED**, and the voltage value of the supply net shall be unspecified.

c) **parallel**

Multiple supply sources, sharing a common root supply driver, may be connected to the supply net.

The **parallel** resolution allows more than one potentially conducting path to the same root supply driver, as if the switches had been connected in parallel. It shall be an error if any of these potentially conducting paths can be traced to more than one root supply driver.

1) If all of the supply sources are **FULL_ON**, then the supply net state is **FULL_ON** and the voltage value is the value of the root supply driver.

2) If all the supply sources driving the supply net are **OFF**, the state of the supply net shall be **OFF** and the voltage is unspecified.

3) If any of the sources is **UNDETERMINED**, the resolution is **UNDETERMINED**; otherwise,

   i) If there is at least one **PARTIAL_ON** source, the supply net shall be **PARTIAL_ON** and the voltage value is the value of the root supply driver.

   ii) If there is at least one source that is **OFF** and at least one that is **FULL_ON** or **PARTIAL_ON**, the supply net shall be **PARTIAL_ON** and the voltage value is the value of the root supply driver. The voltage value of the **PARTIAL_ON** supply net shall be the voltage value of the root supply driver.

d) **parallel_one_hot**

Multiple supply sources may be connected to the supply net. A source may share a common root supply driver with one or more other sources. At most, one root supply driver is **FULL_ON** at any particular time with all sources sharing that driver resolved using parallel resolution.

The **parallel_one_hot** resolution allows resolution of a supply net that has multiple root supply drivers where each driver may have more than one path through supply sources to the supply net. Each unique root supply driver is identified and **one_hot** resolution is applied to the drivers. **parallel**

resolution is then applied to each supply source connecting the **one_hot** root supply driver to the supply net.

6.22.3 Supply nets defined in HDL

The declaration of any VHDL `signal` or SystemVerilog `wire` or `reg` as a `supply_net_type` from the UPF package (see Annex B) is equivalent to calling **create_supply_net** for every instance of that declaration, where the *net_name* is the name of the VHDL `signal` or SystemVerilog `wire` or `reg`, and the scope is the instance.

6.23 create_supply_port

Purpose	Create a supply port on a design element	
Syntax	**create_supply_port** *port_name* [**-domain** *domain_name*] [**-direction** <**in** | **out** | **inout**>]	
Arguments	*port_name*	A simple name.
	-domain *domain_name*	The domain where this port defines a supply net connection point.
	-direction <**in** | **out** | **inout**>	The direction of the port. The default is **in**.
Return value	Return the fully qualified name (from the active scope) of the created port or raise a `TCL_ERROR` if the port is not created.	

The **create_supply_port** command defines a supply port at the scope of the power domain when **-domain** is specified or at the active scope if **-domain** is not specified.

-direction defines how state information is propagated through the supply network as it is connected to the port. If the port is an input port, the state information of the external supply net (see 6.22) connected to the port shall be propagated into the design element. Likewise, for an output port, the state information of the internal supply net connected to the port shall be propagated outside the design element.

Supply ports connected to a net shall be **inout** for supply nets that have both loads and sources within that module. Supply ports are loads, sources, or both.

a) The `LowConn` of an input port is a source.
b) The `HighConn` of an input port is a load.
c) The `LowConn` of an output port is a load.
d) The `HighConn` of an output port is a source.
e) The `LowConn` of an `inout` port is both a load and a source.
f) The `HighConn` of an `inout` port is both a load and a source.

Supply ports may be defined in HDL. If a VHDL or SystemVerilog `port` is declared as a `supply_net_type` from the UPF package (see Annex B); this is equivalent to calling **create_supply_port** for every instance of that declaration, where the *port_name* is the name of the VHDL or SystemVerilog `port`, and the scope is the instance.

For a uni-directional supply port, it shall be an error if there is a driver on the receiving side and a receiver on the driving side; i.e., for an input port, it shall be an error if there is a receiver on the `HighConn` interface

and a driver on the `LowConn` interface; for an output port, it shall be an error if there is a driver on the `HighConn` interface and a receiver on the `LowConn` interface.

Syntax example:

```
create_supply_port VN1
   -direction inout
```

6.24 create_supply_set

Purpose	Create a supply set		
Syntax	**create_supply_set** *set_name* [**-function** {*func_name* [*net_name*]}]* [**-reference_gnd** *supply_net_name*] [**-update**]		
Arguments	*set_name*	The name of the supply set; this shall be a simple (non-hierarchical) name. This simple name exists in the active scope.	
	-function {*func_name* [*net_name*]}	The **-function** option defines the function (*func_name*) a supply net provides for this supply set. *net_name* is a rooted name of a supply net or supply port or a *supply_net_handle*. It shall be an error if the net_name is not defined in the active scope.	**R**
	-reference_gnd *supply_net_name*	The **-reference_gnd** option defines the rooted name of a *supply_net* that serves as the reference ground for the supply set. A *supply_net_handle* may be used. **Default**: if not specified, the voltages in this supply set shall be evaluated with no offset from the assumed default reference supply, which is by definition `0` volts.	**R**
	-update	Use **-update** if the *set_name* has already been defined.	
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.		

create_supply_set creates the supply set name within the active scope in the UPF name space. The reference ground can be specified in any invocation of this command. This command defines a *supply set* as a collection of supply nets each of which serve a specific function for the set.

-update is used to signify that this **create_supply_set** call refers to a supply set that was previously defined. Referencing a previously created supply set without the **-update** argument shall be an error. Using the **-update** argument for a supply set that has not been previously defined shall be an error.

When **-function** is specified, *func_name* may be a reserved function name (see 6.24.1): **power**, **ground**, **nwell**, **pwell**, **deepnwell**, and **deeppwell**; or a user-defined string. If *func_name* has not been previously specified for this supply set, then *func_name* shall be defined for this supply set. If a *func_name* has previously been associated with a *supply_net_name*, it shall be an error if *supply_net_name* does not denote the already associated net or a net connected to the already associated net via one or more ports and/or supply nets. *net_name* may reference a supply net in the descendant hierarchy of the active scope using a *supply_net_handle* (see 6.24.2).

The command can be called multiple times. If *func_name* has been previously defined, but no supply net has been associated with the function, this maps the function name to a supply net and is not a re-declaration of

the function. Otherwise, each instance of **-function** shall define a unique *func_name* for the set or a unique mapping of the function to a supply net.

If a supply set is created and no functions are defined for the set, a tool may implictly define the **power** and **ground** functions when it is used as a power domain primary, a retention strategy supply, isolation strategy supply, level-shifter strategy supply, or a power switch supply.

When **-reference_gnd** is specified, *supply_net_name* is the name of a supply net that serves as the reference ground for the supply set. The voltage value for each supply net in the supply set is interpreted in reference to this supply net. If this parameter is not specified, the voltages shall be evaluated with no offset or scaling. If **-reference_gnd** has previously had a *supply_net_name* specified, then it shall be an error if a specified *supply_net_name* is not an equivalent net.

To avoid early over-specification and to enable incremental refinement of the supply set specification, any supply set name may be referenced with or without any explicitly defined functions. The supply set shall be referenced with one of its handles.

Syntax example:

```
create_supply_set relative_always_on_ss
  -function {power vdd}
  -function {ground vss}
  -reference_gnd {earth_ground}
```

6.24.1 Predefined supply set functions

Predefined functions are available for use in any supply set definition (see 4.3.4).

6.24.2 Referencing supply set functions

The supply set function may also be referenced using a *supply_net_handle* as a member of the supply set (whether or not a supply net has been associated with the function name), as follows:

supply_set_name.function

If no supply net is associated with a supply set's function and that function is used in the design, an implicit supply net with an anonymous name shall be created for use in verification and analysis. When the UPF specification is used for implementation, a supply net shall not be implicitly created for a supply set function that has no associated supply net. A tool may issue a warning or an error if a supply set's function does not have an explicit supply net association.

6.25 create_upf2hdl_vct

Purpose	Define value conversion table that can be used in converting UPF `supply_net_type.state(1:0)` values into HDL logic values	
Syntax	**create_upf2hdl_vct** *vct_name* **-hdl_type** {<**vhdl** | **sv**> [*typename*]} **-table** {{*from_value to_value*}*}	
Arguments	*vct_name*	The value conversion table name.
	-hdl_type {<**vhdl** | **sv**> [*typename*]}	The HDL type for which the value conversions are defined.
	-table {{*from_value to_value*}*}	A list of UPF `state` type values to map to the values of the HDL type.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

The **create_upf2hdl_vct** command defines a value conversion table for the two LSBs of the `supply_state_type.state` value (see Annex B) when that value is propagated from a UPF supply net into a logic port defined in an HDL. It provides a 1:1 conversion for each possible combination of the partially on and on/off states. **create_upf2hdl_vct** does not check that the values are compatible with any HDL port type.

vct_name provides a name for the value conversion table for later use with the **connect_supply_net** command (see 6.13). The predefined VCTs are shown in Annex C.

-hdl_type specifies the HDL type for which the value conversions are defined. This information allows a tool to provide completeness and compatibility checks. If the *typename* is not one of the language's predefined types or one of the types specified in the next paragraph, then it shall be of the form *library*.*pkg*.*type*.

The following HDL types shall be the minimum set of types supported. An implementation tool may support additional HDL types.

a) VHDL
   1) `Bit`, `std_[u]logic`, `Boolean`
   2) Subtypes of `std_[u]logic`
b) SystemVerilog

   `reg`/`wire`, `Bit`, `Logic`

-table defines the 1:1 conversions from UPF supply net states to an HDL logic value. The values shall be consistent with the HDL type values. For example:

- When converting to SystemVerilog *logic type*, the set of legal values is `0`, `1`, `X`, and `Z`.
- When converting to SystemVerilog or VHDL `bit`, the legal values are `0` or `1`.
- When converting to VHDL `std_[u]logic`, the legal values are `U`, `X`, `0`, `1`, `Z`, `W`, `L`, `H`, and `-`.

The conversion values have no semantic meaning in UPF. The meaning of the conversion value is relevant to the HDL model to which the supply net is connected.

Syntax examples:

```
create_upf2hdl_vct upf2vlog_vdd
  -hdl_type {sv}
  -table {{OFF X} {FULL_ON 1} {PARTIAL_ON 0}}

create_upf2hdl_vct upf2vhdl_vss
  -hdl_type {vhdl std_logic}
  -table {{OFF 'X'} {FULL_ON '1'} {PARTIAL_ON 'H'}}
```

6.26 describe_state_transition

Purpose	Describe a state transition's legality	
Syntax	**describe_state_transition** *transition_name* **-object** *object_name* {**-from** {*from_list*} **-to** {*to_list*} | **-paired** {{*from_state to_state*}*} | **-from** {*from_list*} **-to** {*to_list*} **-paired** {{*from_state to_state*}*}} [**-legal** | **-illegal**]	
Arguments	*transition_name*	Simple name.
	-object *object_name*	Simple name of a power domain or supply set.
	-from {*from_list*} **-to** {*to_list*}	*from_list* is an unordered list of power state names active prior to a state transition and to*_list* is an unordered list of power state names active afterwards.
	-paired {{*from_state to_state*}*}	A list of from-state name and to-state name pairs.
	-legal | **-illegal**	Define the state transition as legal or illegal, the default is **-legal**.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

describe_state_transition specifies the legality of a transition from one object's named power state to another. The occurrence of an unnamed state during the transition from one state to another is ignored.

-from and **-to** specify many-to-many transitions. **-paired** specifies one or more one-to-one transitions.

If an empty list is specified in either the **-from** or **-to** *list*, it shall be expanded to all named power states for the specified *object_name*.

Verification tools shall emit an error when an illegal state transition occurs.

It shall be an error if the state name in a *list* refers to a supply net state.

Syntax example:

```
describe_state_transition turn_on -object PdA -from {SLEEP_MODE}
   -to {HIGH_SPEED_MODE} -illegal
```

6.27 load_simstate_behavior

Purpose	Load the simstate behavior defaults for a library	
Syntax	**load_simstate_behavior** *lib_name* **-file** {*file*}*	
Arguments	*lib_name*	The tool specific library name for which the simstate behavior file is to be loaded.
	-file {*file*}*	The file name containing the **set_simstate_behavior** commands.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

Loads a upf file that only contains **set_simstate_behavior** commands and applies these to the models in the library *lib_name*.

It shall be an error if

- *lib_name* cannot be resolved
- *file* does not exist
- a model specified in *file* cannot be found
- the **set_simstate_behavior** commands in *file* use the **-lib** argument
- *file* contains UPF commands other than **set_simstate_behavior**

Syntax example:

```
load_simstate_behavior library1 -file simstate_file.upf
```

6.28 load_upf

Purpose	Set the scope to the specified instance and execute the specified UPF commands	
Syntax	**load_upf** *upf_file_name* [**-scope** *instance_name*] [**-version** *upf_version*]	
Arguments	*upf_file_name*	The UPF file to execute.
	-scope *instance_name*	The scope where the UPF commands contained in *upf_file_name* are executed.
	-version *upf_version*	The version of *upf_file_name*. See also 6.52.
Return value	Return a `1` if all commands in the loaded UPF file completed successfully, or raise a `TCL_ERROR` if the command fails or any command in the loaded UPF file fails.	

The **load_upf** command sets the scope to the specified instance and executes the set of UPF commands in the file *upf_file_name*. Upon return, the active scope is restored to what it was prior to invocation.

load_upf does not create a new name space for the loaded UPF file. The loaded UPF file is responsible for ensuring the integrity of both its own and the caller's name space as needed using existing Tcl name space management capabilities. **load_upf** is a short-hand for the following sequence of commands:

```
system_scope_save_var = set_scope
upf-command-to-read/source-file upf_file
set_scope system_scope_save_var
```

-version can be used to specify the UPF version number according to which the UPF file is interpreted. When **-version** is not specified, the loaded UPF file shall be interpreted based on the active UPF version in the tool.

If the tool implementation does not support the specified version, the command fails.

See also 6.29.

Syntax example:

```
load_upf my.upf -version 1.0
```

6.29 load_upf_protected

Purpose	Load a UPF file in a protected environment that prevents corruption of existing variables	
Syntax	**load_upf_protected** *upf_file_name* [**-hide_globals**] [**-scope** *scope_name*] [**-version** *upf_version*] [**-params** *param_list*]	
Arguments	*upf_file_name*	The UPF file to be sourced.
	-hide_globals	Save all globals before sourcing *upf_file_name* and restore them afterwards. Globals named in the *param_list* retain any modified values resulting from sourcing the file. Any globals not in the *param_list* shall be unset before *upf_file_name* is loaded. Any globals created in the sourced file, other than the ones named in *param_list*, are unset at the end of loading.
	-scope *scope_name*	Set the scope for sourcing the file.
	-version *upf_version*	The version of *upf_file_name*. See also 6.52.
	-params *param_list*	A list of variables to be made available while sourcing the file. In *param_list*, each element has one of the following formats. a) *param_name* — declared as `"global $paramName"`. Any changes made to this variable are visible at the calling level once this command completes. b) {*param_name param_value*} — a local variable *param_name* is created and its initial value is set to *param_nalue*. The tcl variable `errorInfo` shall behave as if it has been specified in this list.
Return value	Return a `1` if all commands in the loaded UPF file completed successfully, or raise a `TCL_ERROR` if the command fails or any command in the loaded UPF file fails.	

load_upf_protected relies on any local variables prefixed with `load_protected_` not being corrupted in the sourced file. It calls the following UPF commands:

set_scope

set_upf_version

and calls the following local commands (see Annex D):

load_protected_save_globals

load_protected_restore_globals

-version can be used to specify the UPF version number according to which the UPF file is interpreted. When **-version** is not specified, the loaded UPF file shall be interpreted based on the active UPF version in the tool.

If the tool implementation does not support the specified version, the command fails.

Syntax example:

```
load_upf_protected my.upf -hide_globals -version 2.0
```

6.30 map_isolation_cell

Purpose	Map a particular isolation strategy to a library cell or range of library cells	
Syntax	**map_isolation_cell** *isolation_name* **-domain** *domain_name* [**-elements** *element_list*] [**-lib_cells** *lib_cells_list*] [**-lib_cell_type** *lib_cell_type*] [**-lib_model_name** *model_name* {**-port** {*port_name net_name*}}*]	
Arguments	*isolation_name*	Identify the isolation strategy specified in a **set_isolation** command (see 6.40) for the specified domain.
	-domain *domain_name*	The domain to which this strategy applies.
	-elements *element_list*	A list of ports to use for this strategy.
	-lib_cells *lib_cells_list*	The list of library cells to use.
	-lib_cell_type *lib_cell_type*	Raise an error.
	-lib_model_name *model_name* {**-port** {*port_name net_name*}}*	The name of the behavioral model and port connectivity for this strategy.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

The **map_isolation_cell** command provides a mechanism for further constraining implementation choices. See also **use_interface_cell** (6.53).

-elements specifies the ports (directly or indirectly) within the domain to which the mapping command is applied. If **-elements** is not specified, all ports inferred from the *isolation_name* strategy shall have the mapping applied.

For **-lib_cells** and **-lib_model_name**, the following apply:

a) If **-lib_cells** is specified
   1) A cell from *lib_cell_list* shall be used for implementation; it shall be an error if an acceptable cell is not available in *lib_cell_list*.
   2) The verification semantic is that of the inferred RTL behavior prescribed by the isolation strategy *isolation_name*; verification semantics are unchanged by the presence or absence of the **-lib_cells** option.
   3) It shall be an error if **-lib_model_name** is also specified.

b) If **-lib_model_name** is specified
   1) *model_name* shall be used as the verification and implementation model.
   2) Logic and supply ports shall be connected as specified by **-port** options.
   3) Neither implicit nor automatic supply net connections apply.
   4) All supply net connections shall be specified with **-port**.
   5) Automatic corruption verification semantics do not apply to a *model_name*.
   6) It shall be an error if **-lib_cells** is also specified.

It shall be an error if

— *domain_name* does not indicate a previously created power domain
— an element in the **-elements** *element_list* is not covered by a **set_isolation** command
— *isolation_name* is not specified
— neither **-lib_cells** nor **-lib_model_name** is specified

NOTE—All **map_*** commands specify the elements to be used rather than inferred through a strategy. The behavior of this manual mapping may lead to an implementation that is different from the RTL specification. Therefore, it may not be possible for logical equivalence checking tools to verify the equivalence of the mapped element to its RTL specification.

Syntax example:

```
map_isolation_cell test_PD1 -domain PD1 -lib_cell_type jason_iso_fast
```

6.31 map_level_shifter_cell

Purpose	Map a particular level-shifter strategy to a simulation or implementation model	
Syntax	**map_level_shifter_cell** *level_shifter_strategy* **-domain** *domain_name* **-lib_cells** *list* [**-elements** *element_list*]	
Arguments	*level_shifter_strategy*	Identify the level-shifter strategy specified in a **set_level_shifter** command (see 6.42) for the specified domain.
	-domain *domain_name*	The domain to which this strategy applies.
	-lib_cells *list*	The list of library cells to use.
	-elements *element_list*	A list of ports to use from the *level_shifter_strategy* strategy.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

The **map_level_shifter_cell** command can be used to map a particular level-shifter strategy to a simulation or implementation model. The level-shifter supply sets are automatically connected to the model ports if the ports are attributed as required to support automatic connections. See also 6.53.

If **-elements** is specified, it identifies the subset of elements defined with the *level_shifter_strategy* strategy to which this command applies. When **-elements** is not specified, this strategy applies to all elements requiring level-shifters from the strategy within the power domain.

It shall be an error if

- *domain_name* is not defined
- an element in the elements list is not covered by a **set_level_shifter** command that defines the *level_shifter_strategy* strategy
- *level_shifter_strategy* is not defined (for *domain_name*)

NOTE—All **map_*** commands specify the elements to be used rather than inferred through a strategy. The behavior of this manual mapping may lead to an implementation that is different from the RTL specification. Therefore, it may not be possible for logical equivalence checking tools to verify the equivalence of the mapped element to its RTL specification.

Syntax example:

```
map_level_shifter_cell shift_up -domain PwrDomZ
   -lib_cells {/library2/LS_LH /library2/LS_HL}
```


6.32 map_power_switch

Purpose	Specify which power switch model is to be used for the implementation of the corresponding switch instance	
Syntax	**map_power_switch** {*switch_name*} **-domain** *domain_name* **-lib_cells** {*list*} [**-port_map** {{*mapped_model_port switch_port_or_supply_net_ref*}*}]	
Arguments	{*switch_name*}	A list of switches [as defined by **create_power_switch** (see 6.20)] to map.
	-domain *domain_name*	This argument is ignored and provided for syntactic backward compatibility only.
	-lib_cells {*list*}	A list of library cells.
	-port_map {{*mapped_model_port switch_port_or_supply_net_ref*}*}	*mapped_model_port* is a port on the model being mapped. *switch_port_or_supply_net_ref* indicates a supply or logic port on a switch: an input supply port, output supply port, control port, or acknowledge port; or it references a supply net from a supply set associated with the switch. See also **create_power_switch** (6.20) or **set_power_switch** (6.46).
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

The **map_power_switch** command can be used to explicitly specify which power switch model is to be used for the corresponding switch instance.

-lib_cells specifies the set of library cells to which an implementation can be mapped.

If **-port_map** is not specified, the ports of the switch instance are associated to library cell ports by matching the respective port names, this is *named association*. It shall be an error if any ports on either the switch instance or the library cell are not mapped when named association is used.

It shall be an error if *switch_name* is not specified.

NOTE 1—All **map_*** commands specify the elements to be used rather than inferred through a strategy. The behavior of this manual mapping may lead to an implementation that is different from the RTL specification. Therefore, logical equivalence checking tools may not be able to verify the equivalence of the mapped element to its RTL specification.

NOTE 2—**create_power_switch** can be used to define an abstract power switch that implementation tools may expand into multiple switches. **create_power_switch** can also be used to specify the need for a specific switch that can then be mapped to a specific switch implementation using **map_power_switch**. It is not meant to be used as a single definition representing multiple physical switches to be mapped with **map_power_switch**.

Syntax example:

```
map_power_switch switch_sw1
-domain test_suite
-lib_cells {}
-port_map {{inp1 vin1} {inp2 vin2} {outp vout}
   {c1 ctrl_small} {c2 ctrl_large}}
```

6.33 map_retention_cell

<table>
<tr><td>Purpose</td><td colspan="2">Constrain implementation alternatives, or specify a functional model, for retention strategies</td></tr>
<tr><td>Syntax</td><td colspan="2">map_retention_cell retention_name_list
-domain domain_name
[-elements element_list]
[-exclude_elements exclude_list]
[-lib_cells lib_cell_list]
[-lib_cell_type lib_cell_type]
[-lib_model_name name {-port port_name net_ref}*]</td></tr>
<tr><td rowspan="7">Arguments</td><td>retention_name_list</td><td>A list of target retention strategy names defined in domain_name using set_retention commands (see 6.47).</td></tr>
<tr><td>-domain domain_name</td><td>The domain in which the strategies are defined.</td></tr>
<tr><td>-elements element_list</td><td>A list of design elements, named processes, or sequential <code>reg</code> or signal names whose respective sequential elements shall be mapped as specified.</td></tr>
<tr><td>-exclude_elements exclude_list</td><td>A list of design elements, named processes, or sequential <code>reg</code> or signal names whose respective sequential elements shall be excluded from mapping.</td></tr>
<tr><td>-lib_cells lib_cell_list</td><td>A list of library cell names. Each cell in the list has retention behavior and is otherwise identical to the inferred RTL behavior of the underlying sequential element.</td></tr>
<tr><td>-lib_cell_type lib_cell_type</td><td>The attribute of the library cells used to identify cells that have retention behavior and are otherwise identical to the inferred RTL behavior of the underlying sequential element.</td></tr>
<tr><td>-lib_model_name model_name {-port port_name net_ref}*</td><td>The name of the library cell or behavioral model and associated port connectivity.</td></tr>
<tr><td>Return value</td><td colspan="2">Return a <code>1</code> if successful or raise a <code>TCL_ERROR</code> if not.</td></tr>
</table>

The **map_retention_cell** command constrains retention strategy implementation choices and may also specify functional retention behavior for verification.

-elements identifies elements from the *effective_element_list* (see 6.3) from a retention strategy in *retention_name_list*. If **-elements** is not specified, the *aggregate_element_list* for this command contains all elements from the *effective_element_list* of the *retention_name_list*.

It shall be an error if at least one of **-lib_cells**, **-lib_cell_type**, or **-lib_model_name** is not specified.

— If **-lib_cells** is specified, a retention cell from *lib_cell_list* shall be used; if **-lib_cell_type** is specified, a retention cell that has the attribute *lib_cell_type* on the implementation model shall be used to implement the functionality specified by the corresponding retention strategy;
if **-lib_cells** and **-lib_cell_type** are both specified, a retention cell from *lib_cell_list* that also has the attribute *lib_cell_type* on the implementation model shall be used.
Verification semantics are unchanged by the presence or absence of **-lib_cells** or **-lib_cell_type**.

— If **-lib_model_name** is specified, *model_name* shall be used as the verification model and supply and logic ports shall be connected as specified by **-port** options; automatic corruption and retention verification semantics do not apply to a **-lib_model_name** model.

— If **-lib_model_name** is not specified, the verification semantic is that of the inferred RTL behavior of the underlying sequential element modified by the retention behavior prescribed by the applicable **set_retention** strategy.

Table 6 summarizes the semantics for combinations of **-lib_cells**, **-lib_cell_type**, and **-lib_model_name**.

Table 6—map_retention_cell option combinations

-lib_cells	-lib_cell_type	-lib_model_name	Verification semantic	Implementation cell constrained to
N	N	N	ERROR	ERROR
N	N	Y	*model_name*	*model_name*
N	Y	N	RTL with retention	*lib_cell_type*
N	Y	Y	*model_name*	*lib_cell_type*
Y	N	N	RTL with retention	*lib_cell_list*
Y	N	Y	*model_name*	*lib_cell_list*
Y	Y	N	RTL with retention	a cell from *lib_cell_list* that also has *lib_cell_type*
Y	Y	Y	*model_name*	a cell from *lib_cell_list* that also has *lib_cell_type*

For verification, an inferred register is assumed to have the following generic canonical interface:

— **CLOCK** - The signal whose rising edge triggers the register to load data.
— **DATA** - The signal whose value represents the next state of the register.
— **ASYNC_LOAD** - The signal that causes the register to load data when its value is one.
— **OUTPUT** - The signal that propagates the register output to the receivers of the register.

-port connects the specified *net_ref* to a *port* of the model. A *net_ref* may be one of the following:

a) A logic net name
b) A supply net name
c) One of the following symbolic references
   1) **retention_ref**.*function_name*

      This names a retention supply set function, where *function_name* refers to the supply net corresponding to the function it provides to the retention *ret_supply_set* (see 6.47).
   2) **primary_ref**.*function_name*

      This names a primary supply set function, where *function_name* refers to the supply net corresponding to the function it provides to the primary supply set of the domain.
   3) **save_signal**
      i) Refers to the save signal specified in the corresponding retention strategy.
      ii) To invert the sense of the save signal, the Verilog bit-wise negation operator ~ can be specified before the *net_ref*. The logic inferred by the negation shall be implicitly connected to the *ret_supply_set* from the corresponding **set_retention** command (see 6.47).

4) restore_signal

i) Refers to the restore signal specified in the corresponding retention strategy.

ii) To invert the sense of the restore signal, the Verilog bit-wise negation operator ~ can be specified before the *net_ref*. The logic inferred by the negation shall be implicitly connected to the *ret_supply_set* from the corresponding **set_retention** command (see 6.47).

5) UPF_GENERIC_CLOCK

i) Refers to the canonical **CLOCK**.

ii) To invert the sense of the clock signal, the Verilog bit-wise negation operator ~ can be specified before the *net_ref*. The logic inferred by the negation shall be implicitly connected to the *primary_supply_set*.

6) UPF_GENERIC_DATA

i) Refers to the canonical **DATA**.

ii) To invert the sense of the data signal, the Verilog bit-wise negation operator ~ can be specified before the *net_ref*. The logic inferred by the negation shall be implicitly connected to the *primary_supply_set*.

7) UPF_GENERIC_ASYNC_LOAD

i) Refers to the canonical **ASYNC_LOAD**.

ii) To invert the sense of the asynchronous load signal, the Verilog bit-wise negation operator ~ can be specified before the *net_ref*. The logic inferred by the negation shall be implicitly connected to the *primary_supply_set*.

8) UPF_GENERIC_OUTPUT

i) Refers to the canonical **OUTPUT**.

ii) To invert the sense of the output signal, the Verilog bit-wise negation operator ~ can be specified before the *net_ref*. The logic inferred by the negation shall be implicitly connected to the *primary_supply_set*.

If **UPF_GENERIC_OUTPUT** is not explicitly mapped and the model has exactly one output port, that output port shall automatically be connected to the net that propagates the register output to the receivers of the register.

NOTE—All **map_*** commands specify the elements to be used rather than inferred through a strategy. The behavior of this manual mapping may lead to an implementation that is different from the RTL specification. Therefore, it may not be possible for logical equivalence checking tools to verify the equivalence of the mapped element to its RTL specification.

It shall be an error if

— *retention_name_list* is not specified

— *domain_name* does not indicate a previously created power domain

— A retention strategy in *retention_name_list* does not indicate a previously defined retention strategy

— An element in *element_list* is not included in the element list of a targeted retention strategy

— Any retention strategy in *retention_name_list* does not specify signals needed to provide connection of the mapped functions

— After completing the *port* and *net_ref* connections, any input port is unconnected, or no output port is connected to the net that propagates the register output to the receivers of the register

— In implementation, none of the specified models in *lib_cell_list* implements the functionality specified by a targeted retention strategy

— In implementation, none of the specified models having a *lib_cell_type* attribute implements the functionality specified by a targeted retention strategy

— In implementation, none of the specified models in *lib_cell_list* that have a *lib_cell_type* attribute, when both are specified, implements the functionality specified by a targeted retention strategy.

Syntax example:

```
map_retention_cell {my_PDA_ret_strat_1 my_PDA_ret_strat_2 my_PDA_ret_strat_3}
    -domain PowerDomainA
    -elements {foo/U1 foo/U2}
   -lib_cells {RETFFIMP1 RETFFIMP2}
   -lib_cell_type FF_CKLO
    -lib_model_name RETFFVER {
       -port CP       UPF_GENERIC_CLOCK
       -port D        UPF_GENERIC_DATA
       -port SET      UPF_GENERIC_ASYNC_LOAD
       -port SAVE     save_signal
       -port RESTORE  restore_signal
       -port VDDC     primary_supply_set.power
       -port VDDRET   ret_supply_set.power
       -port VSS      primary_supply_set.ground }
```

6.34 merge_power_domains

Purpose	Merge two or more existing power domains into a single, new power domain	
Syntax	**merge_power_domains** *new_domain_name* **-power_domains** *list* [**-scope** *instance_name*] [**-all_equivalent**]	
Arguments	*new_domain_name*	The new, merged power domain.
	-power_domain *list*	The list of existing power domains to merge.
	-scope *instance_name*	The scope, i.e., the instance, where the new domain is created.
	-all_equivalent	All other power domains equivalent to the specified power domains shall be merged.
Return value	Return the number of power domains merged or raise a `TCL_ERROR` if the merge is unsuccessful.	

The **merge_power_domains** command merges two or more existing power domains into a single, new power domain. The merged domains can no longer be referenced separately. See also 6.15.

The merged power domain's scope in the logic hierarchy is specified by the **-scope** option (default is the active scope). The merged power domain extent is the union of the design elements contained in the *list* of power domains that are to be merged. Design elements may be added only to the merged power domain after the *list* of power domains have been merged. (The domains are merged upon return of this command.) The merged power domain contains all supply nets, ports, and switches created in each of the constituent power domains and those objects all exist within the same scope as they had prior to the merge.

All strategies and mappings defined for the *list* of power domains shall be applied prior to the merge to avoid ambiguity in applying the strategies. Any strategies and mappings defined after the merge command shall only refer to the merged power domain. Any power state tables defined for the *list* of power domains are ignored.

A *shared primary supply* is a supply net or supply port that is shared and used as the same supply type (including power, ground, bias, and other specified supply functions of the supply set) between two or more power domains. Identifying a shared primary supply requires tracing the supply network connectivity for each power domain, as follows:

a) The active supply is set to a power domain's primary supply net (either power or ground depending on which supply type network is being traced).

b) The active supply is pushed onto a queue for the power domain.

c) If the active supply is a net that is not connected to a source supply port, then the trace back terminates. Otherwise, the active supply is set to the source supply port connected to the net; loop back to b).

d) If the active supply is a port that is an output port of a switch or a supply port on the design top, the trace back is terminated. Otherwise, the active supply is set to the supply net connected as a source to the supply port; loop back to b).

e) Any supply port or supply net that exists in the queue of each power domain is a *shared primary supply* for that power domain. The **merge_power_domains** command shall use the first shared primary supply that is popped from the queues.

For two or more power domains to be merged, the following conditions need to be met:

— The primary power nets for both domains need to be sourced from a shared primary supply.

— The primary ground nets for both domains need to be sourced from a shared primary supply.

— New primary power and ground nets are created for the power domain with implicit supply ports created on the instance in the logic hierarchy corresponding to the merged power domain's scope. These ports are connected to the shared primary power and shared primary ground supplies external to the merged power domain (supply nets are implicitly created and connected if necessary) and to the primary power and ground nets defined within the power domain. The primary power and ground nets are connected to the design elements of the power domain normally.

If **-scope** is not specified, the power domain is created within the active scope.

Power domains that have shared primary power and shared primary ground supplies are *equivalent*.

Syntax example:

```
merge_power_domains PD9 -power_domains {PD9A PD9B PD9C}
```


6.35 name_format

Purpose	Define the format for constructing names of implicitly created objects	
Syntax	**name_format** [**-isolation_prefix** *string*] [**-isolation_suffix** *string*] [**-level_shift_prefix** *string*] [**-level_shift_suffix** *string*] [**- implicit_supply_suffix** *string*] [**- implicit_logic_prefix** *string*] [**- implicit_logic_suffix** *string*]	
Arguments	**-isolation_prefix** *string*	The string prepended in front of an existing signal or port name to create a new name used during the introduction of a new isolation cell. The default value is the empty string `""` or `NULL`.
	-isolation_suffix *string*	The string appended to the end of an existing signal or port name to create a new name used during the introduction of a new isolation cell. The default value is the string `_UPF_ISO`.
	-level_shift_prefix *string*	The string prepended in front of an existing signal or port name to create a new name used during the introduction of a new level-shifter cell. The default value is the empty string `""` or `NULL`.
	-level_shift_suffix *string*	The string appended to the end of an existing signal or port name to create a new name used during the introduction of a new isolation cell. The default value is the string `_UPF_LS`.
	-implicit_supply_suffix *string*	The string appended to an existing supply net or port name to create a unique name for an implicitly created supply net or port. The default value is the string `_UPF_IS`.
	-implicit_logic_prefix *string*	The string prepended in front of an existing logic net or port name to create a unique name for an implicitly created logic net or port. The default value is `NULL`.
	-implicit_logic_suffix *string*	The string appended to an existing logic net or port name to create a unique name for an implicitly created logic net or port.The default value is the string `_UPF_IL`.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

Inferred objects have names in the logic design. The name for these objects is constructed as follows:

a) The base name of implicitly created objects is the name of the port or net being isolated or level-shifted, or the supply net, logic net, or port implicitly created to facilitate the connection of a net across hierarchy boundaries.

b) Any specified prefix is then prepended to the base name.

c) Any specified suffix is also appended to the base name.

d) If multiple prefixes or suffixes apply to the same object, they shall be added in the alphabetical order of the option name, e.g., **isolation_prefix** followed by **level_shift_prefix**.

If the generated name conflicts with another previously defined name in the same name space, the generated name is further extended by an underscore (_) followed by a positive integer. The value of the integer is the smallest number that makes the name unique in its name space. An empty string (`""`) is a valid value for any prefix or suffix option. When the prefix and suffix are both `NULL`, only the underscore (_) and number string combination are used as a suffix to disambiguate the name.

Different prefixes and suffixes may be specified in multiple calls to **name_format** (using different arguments). When **name_format** is specified with no options, the name format is reset to the default values shown in the *Arguments* list.

It shall be an error to specify an affix more than once.

Syntax example:

```
name_format -isolation_prefix "MY_ISO_" -isolation_suffix ""
```

A signal, `MY_ISO_FOO`, is created and connected to a new cell's output (to isolate the existing net `FOO`).

6.36 save_upf

Purpose	Create a UPF file of the structures in the relative to the active scope	
Syntax	**save_upf** *upf_file_name* [**-scope** *instance_name*] [**-version** *string*]	
Arguments	*upf_file_name*	The UPF file to write.
	-scope *instance_name*	The scope relative to which the UPF commands are written.
	-version *string*	The UPF version of *upf_file_name*. See also 6.52.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

The **save_upf** command creates a UPF file relative to the active or specified scope. It writes the commands required to describe the power design intent of the scope to the file *upf_file_name*. Upon return, the active scope is restored to what it was prior to invocation.

If the implementation does not support the specified version, the command fails.

Syntax example:

```
save_upf test_suite1_Jan14
```

6.37 set_design_attributes

Purpose	Apply attributes to models or design elements	
Syntax	**set_design_attributes** < **-elements** *element_list* | **-models** *model_list* | **-elements** *element_list* **-models** *model_list* **-exclude_elements** *exclude_list* | **-exclude_elements** *exclude_list* **-models** *model_list* > [**-attribute** *name value*]*	
Arguments	**-elements** *element_list*	A list of rooted names: design elements, named processes, sequential `regs`, or signal names.
	-exclude_elements *element_list*	A list of rooted names: design elements, named processes, sequential `regs`, or signal names to exclude from the *effective_element_list* (see 6.3).
	-models *model_list*	A list of models to be attributed.
	-attribute *name value*	For the enumerated design element or model, associate the attribute *name* with the value of *value*. See Table 1.
Return value	Return 1 if successful or raise a `TCL_ERROR` if not.	

This command sets the specified attributes for models or design elements.

A **UPF_is_leaf_cell** attribute value of **"TRUE"** on a model or design element prevents the **-transitive** processing for the descendants of the attributed model or design element for the following commands.

- **connect_supply_set** (see 6.14).
- **set_isolation** (see 6.40)
- **set_level_shifter** (see 6.42)
- **set_port_attributes** (see 6.45)
- **set_retention** (see 6.47)
- **set_retention_elements** (see 6.49)
- **find_objects** (see 7.1)

The UPF leaf cell treatment of a model or design element can be annotated in HDL using the following attributes.

Attribute name: **UPF_is_leaf_cell**

Attribute value: **<"TRUE" | "FALSE">**

SystemVerilog or Verilog-2005 example:

```
(* UPF_is_leaf_cell="TRUE" *) module steve (<port list>);
```

VHDL example:

```
attribute UPF_is_leaf_cell of steve : entity is "TRUE";
```

When any register (specified or implied) with the **UPF_retention** attribute value set to **"required"** is included in a power domain that has at least one retention strategy, the register shall be included in a retention strategy defined for the domain.

Elements requiring retention can be attributed in HDL as follows:

Attribute name: **UPF_retention**

Attribute value: <**"required"** | **"optional"**>

SystemVerilog or Verilog-2005 example:

```
(* UPF_retention = "required" *) module my_mod;
```

VHDL example:

```
attribute UPF_retention of my_flip : variable is "required";
```

Syntax example:

```
set_design_attributes -elements lock_cache[0] -attribute UPF_is_leaf TRUE
```

6.38 set_design_top

Purpose	Specify the design's root	
Syntax	**set_design_top** *root*	
Arguments	*root*	The root of the design.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

The **set_design_top** command specifies the root of the design. See also 6.50.

Syntax example:

```
set_design_top ALU07
```

6.39 set_domain_supply_net

Purpose	Set the default power and ground supply nets for a power domain	
Syntax	**set_domain_supply_net** *domain_name* **-primary_power_net** *supply_net_name* **-primary_ground_net** *supply_net_name*	
Arguments	*domain_name*	The domain where the default supply nets are to applied.
	-primary_power_net *supply_net_name*	The primary power supply net.
	-primary_ground_net *supply_net_name*	The primary ground net.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

The **set_domain_supply_net** command associates the power and ground supply nets with the primary supply set for the domain.

The primary supply set's power and ground functions for the specified domain are associated with the corresponding power and ground supply net.

It shall be an error if

— *domain_name* does not indicate a previously created power domain.

— The primary supply set for *domain_name* already has a primary power or ground function association.

This command is semantically equivalent to

```
proc set_domain_supply_net {dn pp sn1 pg sn2} {
   if { string equal $pp "-primary_power_net" \
      && string equal $pg "-primary_ground_net"}{
      create_supply_set set_name -function {power $sn1}
         -function {ground $sn2}
      associate_supply_set set_name -handle $dn.primary
      return 1
   } else {
       return -code TCL_ERROR \
           -errorcode $ecode \
           -errorinfo $einfo \
           $resulttext
   }
}
```

where any *italicized* arguments are implementation-defined.

Syntax example:

```
set_domain_supply_net PD1
-primary_power_net PG1
-primary_ground_net PG0
```

6.40 set_isolation

Purpose	Define an isolation strategy		
Syntax	**set_isolation** *isolation_name* **-domain** *ref_domain_name* [**-elements** *element_list*] [**-source** *source_supply_ref* | **-sink** *sink_supply_ref* | **-source** *source_supply_ref* **-sink** *sink_supply_ref* | **-applies_to** <**inputs** | **outputs** | **both**>] [**-applies_to_clamp** <**0** | **1** | **any** | **Z** | **latch** | *value*>] [**-applies_to_sink_off_clamp** <**0** | **1** | **any** | **Z** | **latch** | *value*>] [**-applies_to_source_off_clamp** <**0** | **1** | **any** | **Z** | **latch** | *value*>] [**-isolation_power_net** *net_name*] [**-isolation_ground_net** *net_name*] [**-no_isolation**] [**-isolation_supply_set** *supply_set_list*] [**-isolation_signal** *signal_list* [**-isolation_sense** {<**high** | **low**>*}]] [**-name_prefix** *string*] [**-name_suffix** *string*] [**-clamp_value** {< **0** | **1** | **any** | **Z** | **latch** | *value*>*}] [**-sink_off_clamp** <**0** | **1** | **any** | **Z** | **latch** | *value*> [*simstate_list*]] [**-source_off_clamp** <**0** | **1** | **any** | **Z** | **latch** | *value*> [*simstate_list*]] [**-location** <**automatic** | **self** | **other** | **fanout** | **fanin** | **faninout** | **parent** | **sibling**>] [**-force_isolation**] [**-instance** {{*instance_name port_name*}*}] [**-diff_supply_only** <**TRUE** | **FALSE**>] [**-transitive** <**TRUE** | **FALSE**>] [**-update**]		
Arguments	*isolation_name*	The isolation strategy name exists in the attribute name space of the domain.	
	-domain *ref_domain_name*	The domain for which this strategy is applied.	
	-elements *element_list*	The **-elements** option defines a list of design elements or ports.	**R**
	-source *source_supply_ref*	The **-source** option defines a rooted name of a supply set reference.	**R**
	-sink *sink_supply_ref*	The **-sink** option defines a rooted name of a supply set reference.	**R**
	-applies_to <**inputs** | **outputs** | **both**>	The **-applies_to** option defines whether the domain's input ports, output ports, or both are isolated. The default is **outputs**.	**R**
	-applies_to_clamp <**0** | **1** | **any** | **Z** | **latch** | *value*>	The **-applies_to_clamp** option specifies only ports that have the specified clamp value are mapped.	**R**
	-applies_to_sink_off_clamp <**0** | **1** | **any** | **Z** | **latch** | *value*>	The **-applies_to_sink_off_clamp** option specifies only ports that have the specified **sink_off_clamp** value are mapped.	**R**
	-applies_to_source_off_clamp <**0** | **1** | **any** | **Z** | **latch** | *value*>	The **-applies_to_source_off_clamp** option specifies only ports that have the specified **source_off_clamp** value are mapped.	**R**

Arguments	**-isolation_power_net** *net_name*	This option defines the supply net used as the power for the isolation logic inferred by this strategy.	**R**
	-isolation_ground_net *net_name*	This option defines the supply net used as the power for the isolation logic inferred by this strategy.	**R**
	-no_isolation	Prohibits isolation according to *isolation_name*.	**R**
	-isolation_supply_set *supply_set_list*	This option defines the *supply_set_list* used to power the logic inferred by the *isolation_name* strategy.	**R**
	-isolation_signal *signal_list*	The **-isolation_signal** option defines the *signal_list* that causes the specified element to drive its clamp value.	**R**
	-isolation_sense {<**high** | **low**>*}	The **-isolation_sense** option defines the active level of each corresponding isolation signal in the *signal_list*. The default is **high**.	**R**
	-name_prefix *string* **-name_suffix** *string*	The name format (prefix & suffix) for generated isolation instances or nets related to implementation of the isolation strategy.	
	-clamp_value {<**0** | **1** | **any** | **Z** | **latch** | *value*>*}	The **-clamp_value** option defines the value of an isolated port for the corresponding isolation signal in *signal_list*. The default is **any**.	**R**
	-sink_off_clamp <**0** | **1** | **any** | **Z** | **latch** | *value*> [*simstate_list*]	The **-sink_off_clamp** option specifies the clamp requirement when the sink domain is off.	**R**
	-source_off_clamp <**0** | **1** | **any** | **Z** | **latch** | *value*> [*simstate_list*]	The **-source_off_clamp** option specifies the clamp requirement when the source domain is off.	**R**
	-location <**automatic** | **self** | **other** | **fanout** | **fanin** | **faninout** | **parent** | **sibling**>	The **-location** option defines where the isolation cells are placed in the logic hierarchy. The default is **automatic**.	**R**
	-force_isolation	Implements the isolation strategy regardless of any analysis of its necessity based on the design specification.	**R**
	-instance {{*instance_name port_name*}*}	*instance_name* is a hierarchical name. *port_name* is a hierarchical name.	**R**
	-diff_supply_only <**TRUE** | **FALSE**>	Determines the isolation behavior between driver and receiver supply sets. The default is **FALSE**.	**R**
	-transitive <**TRUE** | **FALSE**>	When **-transitive** is **TRUE** (the default), the command applies to the descendants of the elements.	
	-update	Use **-update** if the *isolation_name* has already been defined.	
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.		

The **set_isolation** command specifies the ports on the *ref_domain_name* to isolate using the specified strategy. The interface of a domain is defined as:

The `LowConn` side of ports defined on the top-level design elements in the extent of the domain.

The `HighConn` side of ports defined on design elements in other power domains, but instanced within design elements in the extent of the domain.

For **-diff_supply_only**, no isolation shall be introduced into the path from the driver to the receiver for an isolation strategy defined on a port on the interface of *ref_domain_name*, where the driver is powered by the same supply as a receiver of the port.

If a **-diff_supply_only**, **-source**, or **-sink** argument is used and design elements are included in designs with different power distribution or connectivity, the evaluation of the need for isolation may vary and cause a change in the logical function of a block.

It shall be an error if a **diff_supply_only**, **-source**, or **-sink** argument is used and it is not possible to determine the supply of all of the drivers and receivers for any candidate port.

NOTE 1— **-diff_supply_only**, **-source**, and **-sink** may impact the use of some design implementation flows.

When the *aggregate_element_list* (see 6.3) contains no elements, every port on the interface of the domain is added into *aggregate_element_list*.

If an element in *effective_element_list* is not on the interface of *ref_domain_name*, it shall not be isolated.

The arguments **-source**, **-sink**, **-applies_to**, **-applies_to_clamp**, **-applies_to_sink_off_clamp**, and **-applies_to_source_off_clamp** serve to filter the set of elements for a given **set_isolation** command invocation. For any port in the *prefilter_element_list*, the following filtering functions are applied:

- **-source** filters the ports receiving a net that is driven by logic powered by the supply set.
- **-sink** filters the ports driving a net that fans out to logic powered by the supply set.
- When both **-source** and **-sink** are specified, a port is included if it has a source as specified and a sink as specified.
- **-applies_to** filters the ports within the domain for which this strategy is defined that have the specified mode. For ports whose `LowConn` is on the interface of the *ref_domain_name*, the port is selected when the direction of the port matches. For ports whose `HighConn` is on the interface of the *ref_domain_name*, the port is selected when the inverse of the direction of the port matches.

  For example, if a port is on a design element in the extent of another domain and that design element is a child of a design element within the extent of *ref_domain_name*, that port shall match the **-applies_to** `IN` filter when its direction is `OUT`. **-applies_to** is always relative to *ref_domain_name*.
- **-applies_to_clamp**, **-applies_to_sink_off_clamp**, and **-applies_to_source_off_clamp** filters the ports within the *ref_domain_name* for which this strategy is defined that have the specified value for the respective port attribute.

For a selected output port on the interface of the *ref_domain_name*, isolation is only performed on the subset of the fanout that drives an element powered by the *sink_supply_ref*. For a selected input port on the interface of the *ref_domain_name*, isolation is only performed on the subset of the fanin driven by an element powered by the *source_supply_ref*.

The arguments **-isolation_power_net**, **-isolation_ground_net**, and **-isolation_supply_set** serve to specify the supply net or supply set for a given **set_isolation** command invocation.

If isolation power and isolation ground nets are specified, an implicit isolation supply set is created and used with the strategy. The isolation power net serves the **power** function in the isolation supply set and the isolation ground net serves the **ground** function in the isolation supply set. If the isolation **power** net is specified but the isolation ground net is not specified then *ref_domain_name***.primary.ground** shall be used as the isolation **ground**. If the isolation ground net is specified but the isolation **power** net is not specified then *ref_domain_name***.primary.power** shall be used as the isolation **power**. It shall be an error if a isolation supply set is specified and an isolation supply net is individually defined.

The isolation supply set(s) specified by **-isolation_supply_set** are implicitly connected to the isolation logic inferred by this command.

The isolation supply, isolation signal and sense, and isolation clamp are specified as lists. The tuples formed by associating the positional entries from each list shall be used to define separate isolation requirements for the strategy. These tuples are applied to the isolation cell for the specified port from the isolation cells' data input port to its data output port in the order in which they appear in each list.

The lists for these arguments need to match in length, except an isolation supply list, isolation sense list, or isolation clamp list may contain a single item; in which case, this item shall be applied to all isolation requirement tuples. It shall be an error if any list with more than one item does not have the same number of items as any other list that has more than one item. It shall be an error if the number of items in the isolation signal list is not the same as any other list that has more than one item.

The clamp value (**-clamp_value**) is enabled when **-isolation_signal** has the specified **-isolation_sense**. The logic value of the isolation is specified by **-clamp_value** as

logic **0**

logic **1**

logic **Z**

latch (the value of the non-isolated port when the isolation signal becomes active)

any (the port shall be isolated with any clamp value legal for the port type)
In simulation, a clamp value of **any** shall be modeled as a `0` for logic and integer types; for all other types, this shall be modeled as the default initial value of the type of the port.

value specifies a value that is legal for the type of the port, e.g., `255` might be specified for an integer-typed port (perhaps constrained to an unsigned 8-bit range).

When isolation is being implemented for a port, the clamp value is determined

a) If **-clamp_value** is specified then it is used.
b) If only **-sink_off_clamp** is defined then that is used.
c) If only **-source_off_clamp** is defined then that is used.
d) If both **-sink_off_clamp** and **-source_off_clamp** are specified with the same value, then that value is used.
e) If both **-sink_off_clamp** and **-source_off_clamp** are specified with different values and the **-clamp_value** is not specified, then **-clamp_value** is inferred such that it satisfies the **-sink_off_clamp** and **-source_off_clamp** requirements, where the **-source_off_clamp** requirement is the first item in the **-clamp_value** list and the **-sink_off_clamp** value is the second and last item in the clamp value list.
f) Otherwise, **-clamp_value** defaults to zero.

-sink_off_clamp specifies the clamp requirement when the supply set connected to the sink is in a power state with a corresponding simstate of **CORRUPT** or any of the simstates specified in *simstate_list*. **-source_off_clamp** specifies the clamp requirement when the supply set connected to the source is in a power state with a corresponding simstate of **CORRUPT** or any of the simstates specified in *simstate_list*. The simstate conditions of **-source_off_clamp** and **-sink_off_clamp** are constraints to be verified and do not represent implementation directives.

When an isolation strategy targets an input port on the interface to a power domain and the sinks of the net connected to a design element within the power domain have different **-sink_off_clamp** requirements specified, more than one isolation element shall be created to satisfy the respective destination groups where a group is a subset of the fanout that has the same **-sink_off_clamp** clamp value.

When an isolation strategy targets an output port on the interface to a power domain and the sinks of the net connected to the port have different **-source_off_clamp** requirements specified, more than one isolation element shall be created to satisfy the respective destination groups where a group is a subset of the fanout that has the same **-source_off_clamp** clamp value.

NOTE 2—The **set_port_attributes** command (see 6.45) is an alternate method to specify **-clamp_value**, **-source_off_clamp**, and **-sink_off_clamp** requirements on ports, which are not necessarily based on a domain boundary, but are propagated to a domain boundary once this information is resolved. Both sets of information can define clamping requirements.

It shall be an error if an isolation strategy targets a port on the interface of a power domain and the source ports on design elements of the net connected to the port have different **-sink_off_clamp** or different **-source_off_clamp** requirements among all sources and the port of the isolation strategy.

Verification shall issue an error when a **-sink_off_clamp** or **-source_off_clamp** requirement is violated.

-location defines where the isolation cells are placed in the logic hierarchy.

automatic—the implementation tool is free to choose the appropriate locations (the default).

self—the isolation cell is placed inside the domain whose interface port is being isolated.

other—the isolation cell is placed in the parent for ports on the interface of the domain that have connections from the parent, and in the child for ports on the interface of the domain that connect to a child.

fanout— isolation cell is placed at all fanout locations (sinks) of the port being isolated.

fanin—isolation cell is placed at all fanin locations (sources) of the port being isolated.

faninout—the isolation cell is placed at all fanout locations (sinks) for each output port being isolated, or at all fanin locations (sources) for each input port being isolated.

parent—the isolation cell is placed in the parent of the domain whose interface port is being isolated.

sibling—a new sibling is created into which the isolation cells are placed.

-instance specifies that the isolation functionality exists in the HDL design and *instance_name* denotes the design element providing the isolation functionality. In this case, the following also apply:

Isolation enable signal(s) are automatically connected to ports on an instance where the port has the attribute `isolation_cell_enable_pin` set to `TRUE`.

If the strategy specifies multiple isolation supply sets, the `isolation_cell_enable_pin` attributed ports shall have related power, ground, and bias port attributes (see 6.44 and 6.45). The supply nets of the isolation supply set corresponding to the isolation enable signal shall be automatically connected to the supply ports matching the related power, ground, and bias ports of the port connected to that isolation enable signal based on the supply set function and `pg_type`. If there is more than one isolation enable signal for the strategy, then the port attribute `isolation_cell_enable_pin_index` shall be specified and its value indicates which enable signal from the list is connected to that port (index `0` corresponds to the enable signal closest to the data input of the isolation cell).

If the strategy specifies a single isolation supply set, the supply nets of the set shall be automatically connected to the isolation supply ports on the instance based on the value of the port's `pg_type` attribute.

If there are no supply ports on the instance, then the isolation supply set(s) specified in the strategy shall be implicitly connected to the instance.

It is an error if there is a single isolation enable signal and there is more than one port on the instance with the `isolation_cell_enable_pin` attribute.

The following also apply:

- This command never applies to *inout* ports.
- The connection from one domain to another domain may have multiple isolation strategies that apply. Implementations may optimize away redundant isolation and issue a warning.
- Nets whose source and sink are in the same power domain shall not have isolation logic inserted.
- A net with sinks in multiple domains shall only have isolation logic inferred for those sinks that are in power domains different from the source.
- If more than one isolation strategy is applied to the same port, the order of the isolation cells insertion from the port's source to its sink is first the **-source_off_clamp** strategy, then the **-clamp_value** isolation strategy, and finally the **-sink_off_clamp** strategy.
- It shall be an error if the application of an isolation strategy results in an undefined topological ordering of the inferred isolation logic.
- It shall be an error if both **-force_isolation** and **-no_isolation** are specified.
- It shall be an error if the application of this command conflicts with attributes specified with **set_port_attributes**.
- It shall be an error if **-applies_to** is specified along with **-source** and/or **-sink**.
- It shall be an error if **-isolation_supply_set** is specified along with **-isolation_power_net** and/or **-isolation_ground_net**.
- After the strategy has been completely applied, it shall be an error if the isolation supply set is not defined for a strategy and the domain does not have a default *isolation_supply_set*.

Isolation clamp value port properties can be annotated in HDL using the attributes shown in 6.45.

Simulation

The simulation semantics for isolation are defined through an equivalent SystemVerilog `always` block, unless **-instance** applies to a specific isolation element or **use_interface_cell** (see 6.53) is applied.

An isolation strategy with a constant clamp value (`0`, `1`, `Z`, or a user-specified value) is functionally equivalent to the following SystemVerilog code.

```
// For -isolation_sense HIGH
genvar x;
generate for (x=0; x < <num_iso_specs>; x++)
always @( isolation_signal[x], <data_input>,
    <isolation_supply_set[x].simstate>)
    if (<isolation_supply_set[x].simstate> == NORMAL)
        if (isolation_signal[x] === 1'bX)
            <data_output> = <corrupt_value_for_logic_type>;
        else if (isolation_signal[x] == 1)
            <data_output> = <clamp_value[x]>;
        else
            <data_output> = <data_input>;
    else
        <data_output> = <corrupt_value_for_logic_type>;
endgenerate
```

The isolation cell with a clamp value of `latch` is functionally equivalent to the following SystemVerilog code.

```
    reg iso_latch;
    assign <isolation_output> = iso_latch;
```

```
// For -isolation_sense LOW
always @( <isolation_signal>, <non_isolated>,
          <isolation_supply_set.simstate>)
begin
    if (<isolation_supply_set.simstate> == NORMAL)
        if ( <isolation_signal === 1'bX )
           <iso_latch> = <corrupt_value_for_logic_type>;

        else if ( <isolation_signal> != 0)
          <iso_latch> = <non_isolated>;
        else
           ;
    else
        <iso_latch> = <corrupt_value_for_logic_type>;
end
```

Syntax example:

```
set_isolation parent_strategy
  -domain pda
  -elements {a b c d}
  -isolation_supply_set {pda_isolation_supply}
   -clamp_value {1}
```

6.41 set_isolation_control

Purpose	Specify the control signals for a previously defined isolation strategy	
Syntax	**set_isolation_control** *isolation_name* **-domain** *domain_name* **-isolation_signal** *signal_name* [**-isolation_sense** <**high** | **low**>] [**-location** <**self** | **parent** | **sibling** | **fanout** | **automatic**>]	
Arguments	*isolation_name*	Isolation strategy name.
	-domain *domain_name*	The domain where the strategy applies.
	-isolation_signal *signal_name*	The signal that causes the specified element to drive its clamp value.
	-isolation_sense <**high** | **low**>	The sense for **-isolation_signal**. The default is **high**.
	-location <**self** | **parent** | **sibling** | **fanout** | **automatic**>	Where the isolation cells are placed in the logic hierarchy. The default is **automatic**.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

The **set_isolation_control** command allows the specification of the isolation control signal and sense separate from the **set_isolation** command (see 6.40) for those situations where the isolation strategy is known, but the control signals are not known until later.

Compatibility note: Excepting **set_isolation_control** is executed within the active scope and the **-location** option, the semantics here are equivalent to having specified the isolation control signal and sense with the **set_isolation** command.

-location defines where the isolation cells are placed in the logic hierarchy.

self— the isolation cell is placed inside the model/cell being isolated.

parent—the isolation cell is placed in the parent of the cell /model being isolated.

sibling—a new sibling is created into which the isolation cells are placed.

fanout—isolation occurs at all fanout locations (sinks) of the port being isolated.

automatic—the implementation tool is free to choose the appropriate locations (the default).

Syntax example:

```
set_isolation outputs_only
-domain PD1
-isolation_power_net VDDbackup
-clamp_value 1
-applies_to outputs

set_isolation_control outputs_only
-domain PD1
-isolation_signal cpu_iso
-isolation_sense low
-location parent
```

6.42 set_level_shifter

Purpose	Specify a level-shifter strategy
Syntax	**set_level_shifter** *level_shifter_name* **-domain** *domain_name* [**-elements** *element_list*] [**-no_shift**] [**-threshold** *value* | *list*] [**-force_shift**] [**-source** *domain_name*] [**-sink** *domain_name*] [**-applies_to** <**inputs** | **outputs** | **both**>] [**-rule** <**low_to_high** | **high_to_low** | **both**>] [**-location** <**self** | **parent** | **sibling** | **fanout** | **automatic**>] [**-name_prefix** *string*] [**-name_suffix** *string*] [**-input_supply_set** *supply_set_name*] [**-output_supply_set** *supply_set_name*] [**-internal_supply_set** *supply_set_name*] [**-instance** {{*instance_name port_name*}*}] [**-transitive** <**TRUE** | **FALSE**>] [**-update**]

Arguments	*level_shifter_name*	Level-shifter strategy name.	
	-domain *domain_name*	The domain for which this strategy is applied.	
	-elements *element_list*	Defines a list of design elements, input ports, and output ports to which this strategy is applied.	**R**
	-no_shift	**-no_shift** prevents the insertion of level-shifters on the specified ports regardless on any other specifications.	
	-threshold *value* | *list*	The voltage threshold (in volts) for determining when level-shifters are required. The default is `0`. *list* is a matrix of values.	
	-force_shift	Unconditional insertion of a level-shifter.	
	-source *domain_name*	Hierarchical name to the domain from the active scope and its descendants.	
	-sink *domain_name*	Hierarchical name to the domain from the active scope and its descendants.	
	-applies_to <**inputs** | **outputs** | **both**>	Whether the domain's input ports, output ports, or both are shifted. The default is **outputs**.	
	-rule <**low_to_high** | **high_to_low** | **both**>	Which type of level-shifters are required. The default is **both**.	
	-location <**automatic** | **self** | **other** | **fanout** | **fanin** | **faninout** | **parent** | **sibling**>	The **-location** option defines where the level-shifter cells are placed in the logic hierarchy. The default is **automatic**.	**R**
	-name_prefix *string* **-name_suffix** *string*	The name format (prefix & suffix) for generated level-shifter instances or nets related to implementation of the shifting strategy.	
Arguments	**-input_supply_set** *supply_set_name*	The **-input_supply_set** option defines the supply set used to power the input portion of the level-shifter.	**R**
	-output_supply_set *supply_set_name*	The **-output_supply_set** option defines the supply set used to power the output portion of the level-shifter.	**R**
	-internal_supply_set *supply_set_name*	The **-internal_supply_set** option defines the supply set used to power internal circuits within the level-shifter.	**R**
	-instance **{{***instance_name port_name***}*}**	The name of a technology library leaf cell instance and the name of the logic port that it level-shifts. If this instance has any unconnected supply ports, then these ports need to have identifying attributes in the cell model and the ports shall be connected in accordance with this **set_level_shifter** command.	**R**
	-transitive <**TRUE** | **FALSE**>	When **-transitive** is **TRUE** (the default), the command applies to the descendants of the elements.	
	-update	Use **-update** if the *level_shifter_name* has already been defined.	
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.		

The **set_level_shifter** command can be used to explicitly specify a strategy for level-shifting during implementation. *Level-shifters* are placed on the connections between domains that operate at different supply levels.

If no level-shifter strategy applies to a connection between domains, level-shifters shall be inferred based on power states (see 5.4). All of the relevant supply sets need to be in the **NORMAL** simstate for the signal to be transmitted without corruption.

The interface of a domain is defined as

— The `LowConn` side of ports defined on the top-level design elements in the extent of the domain.

— The `HighConn` side of ports defined on design elements in other power domains, but instanced by design elements in this domain.

If **-elements** and **-update** are not specified, this is equivalent to the elements list containing every port on the interface of the domain. When **-elements** is specified, the element names shall reference design elements or ports in the *domain_name*.

When the *aggregate_element_list* (see 6.3) contains no elements, every port on the interface of the domain is added into *aggregate_element_list*.

The filters (**-source**, **-sink**, and **-applies_to**) are applied to design element names in the elements list to define a list of ports that is combined with any ports in the **-elements** list. This is the list of candidate ports for the **set_level_shifter** command. The level-shifter strategy is applied only to the candidate ports that are part of the domain's interface.

The complete set of elements for a strategy in a domain is the union of all elements specified by all invocations of the command targeting the strategy.

NOTE 1—This allows a reference to an element that does not exist, but not to an object that exists but is outside the domain. Implementations may issue warnings if an element does not exist within the extent of *domain_name*. The command has no effect on any element that does not exist within the extent of the domain.

The arguments **-source**, **-sink**, and **-applies_to** serve to filter the set of elements for a given **set_level_shifter** command invocation. For any name in **-elements** that refers to a design element (not a port):

— **-source** selects the ports receiving a net that is driven from a port on the interface of the domain specified with this option.

— **-sink** selects the ports driving a net that fans out to a port on the interface of the domain specified with this option.

— When **-source** and **-sink** are specified, a port is included if it has a source as specified or a sink as specified.

— **-applies_to** selects the ports within the domain for which this strategy is defined that have the specified mode. For ports whose `LowConn` is on the interface of the domain, the port is selected when the direction of the port matches. For ports whose `HighConn` is on the interface of the domain, the port is selected when the inverse of the direction of the port matches.

  For example, if a port is on a design element in the extent of another domain and that design element is a child of a design element within the extent of *domain*, that port shall match the **-applies_to** `IN` filter when its direction is `OUT`. **-applies_to** is always relative to *domain*.

For a selected output port on the interface of a domain for which this strategy is specified, level-shifting is only performed on the subset of the fanout that drives an element in the domain specified by the **-sink** option. For a selected input port on the interface of a domain for which this strategy is specified, level -shifting is only performed on the subset of the fanin driven by an element in the domain specified by the **-source** option.

NOTE 2—When only ports are specified in the **-elements** list, any specified filters are ignored.

-threshold can be used to define how large the absolute voltage difference between the source and sink needs to be before level-shifters are inserted.

-threshold *value* is evaluated as shown in this pseudo code.

```
foreach A in the legal power states of the input supply set
   foreach B in the legal power states of the output supply set
      if exists legal power state (A, B)
         if (T_value < max (|A_Nominal_power - B_nominal_power)|,
               |(A_nominal_ground - B_nominal_ground)|))
            return(REQUIRED)
         endif
      endif
   next B
next A
return (NOT REQUIRED)
```

-threshold *list* causes 12 values to be used; these are evaluated in the pseudo code loop {LB_IAP LB_IAG LB_AIP LB_AIG LB_OOP LB_OOG UB_IAP UB_IAG UB_AIP UB_AIG UB_OOP UB_OOG} to provide for threshold evaluations. A shifter may be omitted if all of the following tests are satisfied. Specifying a larger value than any possible input / output voltage difference can be used to disable any individual test.

```
LB_IAP     <    input_mIn_power - output_mAx_Power      <    UB_IAP
LB_IAG     <    input_mIn_ground - output_mAx_Ground     <    UB_IAG
LB_AIP     <    input_mAx_power - output_mIn_Power      <    UB_AIP
LB_AIG     <    input_mAx_ground - output_mIn_Ground     <    UB_AIG
LB_OOP     <    input_nOm_power - output_nOm_Power      <    UB_OOP
LB_OOG     <    input_nOm_ground - output_nOm_Ground     <    UB_OOG
```

When **-no_shift** is specified, the ports to which this command applies shall not be level-shifted.

If **-force_shift** is specified, a level-shifter shall be unconditionally inserted on that port. It shall be an error if a legal location cannot be determined.

-rule can be **low_to_high**, **high_to_low**, or **both**. If **low_to_high** is specified, ports going from a lower voltage to a higher voltage (where *voltage* means power - ground voltage) get a level-shifter if the voltage difference exceeds that specified by **-threshold**. If **high_to_low** is specified, ports going from a higher voltage to a lower voltage get a level-shifter when the voltage difference exceeds that specified by **-threshold**. If **both** is specified, it is equivalent to having specified both rules in the strategy.

-location defines where the level-shifter cells are placed in the logic hierarchy.

automatic—the implementation tool is free to choose the appropriate locations (the default).

self—the level-shifter cell is placed inside the domain whose interface port is being shifted.

other—the level-shifter cell is placed in the parent for ports on the interface of the domain that have connections from the parent, and in the child for ports on the interface of the domain that connect to a child.

fanout—level-shifter cell is placed at all fanout locations (sinks) of the port being shifted.

fanin—level-shifter cell is placed at all fanin locations (sources) of the port being shifted.

faninout—the level-shifter cell is placed at all fanout locations (sinks) for each output port being isolated, or at all fanin locations (sources) for each input port being shifted.

parent—the level-shifter cell is placed in the parent of the domain whose interface port is being shifted.

sibling—a new sibling is created into which the level-shifter cells are placed.

-name_prefix specifies the substring to place at the beginning of any generated name implementing this strategy.

-name_suffix specifies the substring to place at the end of any generated name implementing this strategy.

-input_supply_set is connected to the supply ports related to the input port of the level-shifter. The default is the primary supply set of the domain containing the source of the level-shifter input when the source is within the logic design starting at the design root. This default is used if and only if this primary supply set's component supply nets exist in the extent of the power domain where the level-shifter is going to be located. If the default primary supply set is not in the extent of the power domain, the level-shifter may not be inserted unless the **-force_shift** option is specified and these supply connections are left open. If the input data pin of the level-shifter cell used for this *level_shifter_name* is related to the same supply ports as the output data pin, then the default for this argument is not needed and none is used.

-output_supply_set is connected to the supply ports related to the output port of the level-shifter. The default is the primary supply set of the domain containing the sink of the level-shifter output when the sink is within the logic design starting at the design root. This default is used if and only if this primary supply set's component supply nets exist in the extent of the power domain where the level-shifter is going to be located. If the default primary supply set is not in the extent of the power domain, the level-shifter may not be inserted unless the **-force_shift** option is specified and these supply connections are left open.

-internal_supply_set is connected to the supply ports that are not related to the inputs or outputs of the level-shifter. There are no default auto-connections defined for the **internal_supply_set.**

The following also apply:

- This command never applies to *inout* ports.
- The simstate semantics of all implicitly connected supply sets apply to the output of a level-shifter.
- It shall be an error if the specified location is not within the logic design starting at the design root.
- It shall be an error if **-no_shift** is specified along with any of the following: **-threshold**, **-force_shift**, **-source**, **-sink**, **-applies_to**, **-rule**, **-location**, **-name_prefix**, **-name_suffix**, **-input_supply_set**, or **-output_supply_set**.

It shall be an error if there is a connection between a driver and receiver and all of the following apply:

- The supplies powering the driver and receiver are at different voltage levels.
- A level-shifter is not specified for the connection using a level-shifter strategy.
- A level-shifter cannot be inferred for the connection by analysis of the power states of the supplies to the driver and receiver.

Simulation semantics

A level-shifter has the logical functionality of a buffer.

Syntax example:

```
set_level_shifter shift_up
  -domain PowerDomainZ
  -applies_to outputs
  -threshold 0.02
  -rule both
```

6.43 set_partial_on_translation

Purpose	Define the translation of **PARTIAL_ON** for named tools	
Syntax	**set_partial_on_translation** [**OFF** | **FULL_ON**] [**-full_on_tools** {*string_list*}] [**-off_tools** {*string_list*}]	
Arguments	**OFF** | **FULL_ON**	The default translation for unlisted tools.
	-full_on_tools {*string_list*}	A list of strings.
	-off_tools {*string_list*}	A list of strings.
Return value	Return the setting of the translation if successful or raise a `TCL_ERROR` if not.	

This command defines the translation of **PARTIAL_ON** to **FULL_ON** or **OFF** for purposes of evaluating the power state of supply sets and power domains. The state of a supply *set* is evaluated after this tool-specific translation of **PARTIAL_ON** to **FULL_ON** or **OFF** for each supply *net* in the set.

It shall be an error if

— This command is invoked more than once per tool.

— The same string occurs in both the **-full_on_tools** and **-off_tools** *string_list*s.

Tools shall define the string(s) they recognize in the *string_list* arguments.

If this command is not specified, then tools shall translate **PARTIAL_ON** to **OFF**.

Syntax example:

```
set_partial_on_translation OFF -full_on_tools {power_analysis_tool_name}
   -off_tools {steves_simulator}
```

6.44 set_pin_related_supply

Purpose	Define the related power/ground pair for a library cell	
Syntax	**set_pin_related_supply** *library_cell* **-pins** *list* **-related_power_pin** *supply_pin* **-related_ground_pin** *supply_pin*	
Arguments	*library_cell*	The library cell where the supply nets are to defined.
	-pins *list*	A list of pins that is to have a related power/ground supply defined.
	-related_power_pin *supply_pin*	The instance supply pin to which the pin is related.
	-related_ground_pin *supply_pin*	The instance supply pin to which the pin is related.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

The **set_pin_related_supply** command provides the ability to define the related power and ground pins for signal pins on a library cell. This command conveys information similar to `related_power_pin` and `related_power_pin` in Liberty, but may override them. This command is restricted to only leaf-library cells and not synthesizable hierarchical modules.

UPF file sets that include this command may result in designs that can not be formally verified.

This command has no simulation semantics. To specify simulation semantics, use the receiver and driver supplies in **set_port_attributes** (see 6.45).

A port-supply relationship can be annotated in HDL using the following attributes:

Attribute name: **UPF_related_power_pin** or **UPF_related_ground_pin**.

Attribute value: "*supply_port_name*", where *supply_port_name* is a string whose value is the simple name of a port on the same interface as the attributed port.

SystemVerilog or Verilog-2005 example:

```
(* UPF_related_power_pin = "my_Vdd" *) output my_Logic_Port;
```

VHDL example:

```
attribute UPF_related_power_pin of my_Logic_Port : signal is
"my_Vdd";
```

Syntax example:

```
set_pin_related_supply library1/cell1 -pins {A B C} -related_power_pin VDDX
   -related_ground_pin VSSX
```

6.45 set_port_attributes

Purpose	Define information on ports	
Syntax	**set_port_attributes** [**-ports** {*port_list*}] [**-exclude_ports** {*port_list*}] [{**-domains** {*domain_list*} [**-applies_to** <**inputs** | **outputs** | **both**>]}] [{**-exclude_domains** {*domain_list*} [**-applies_to** <**inputs** | **outputs** | **both**>]}] [{**-elements** {*element_list*} [**-applies_to** <**inputs** | **outputs** | **both**>]}] [{**-exclude_elements** {*exclude_list*} [**-applies_to** <**inputs** | **outputs** | **both**>]}] [**-model** *name*] [**-attribute** *name value*]* [**-clamp_value** <**0** | **1** | **any** | **Z** | **latch** | *value*>] [**-sink_off_clamp** <**0** | **1** | **any** | **Z** | **latch** | *value*>] [**-source_off_clamp** <**0** | **1** | **any** | **Z** | **latch** | *value*>] [**-receiver_supply** *supply_set_ref*] [**-driver_supply** *supply_set_ref*] [**-related_power_port** *supply_port*] [**-related_ground_port** *supply_port*] [**-related_bias_ports** *supply_port_list*] [**-repeater_supply** *supply_set_ref*] [**-pg_type** *pg_type_value*] [**-transitive** <**TRUE** | **FALSE**>]	
Arguments	**-ports** {*port_list*}	A list of ports to be attributed.
	-exclude_ports {*port_list*}	A list of ports to be excluded from the command.
	{**-domains** {*domain_list*} [**-applies_to** <**inputs** | **outputs** | **both**>]}	A list of domains whose ports are to be attributed.
	{**-exclude_domains** {*domain_list*} [**-applies_to** <**inputs** | **outputs** | **both**>]}	A list of domains whose ports are excluded from being attributed.
	{**-elements** {*element_list*} [**-applies_to** <**inputs** | **outputs** | **both**>]}	A list of elements to be attributed.
	{**-exclude_elements** {*exclude_list*} [**-applies_to** <**inputs** | **outputs** | **both**>]}	A list of elements to be excluded from the command.
	-model *name*	A module or library cell.
	-attribute *name value*	Associate the attribute *name* with the selected ports with the value of *value*.

Arguments	**-clamp_value** <**0** | **1** | **any** | **Z** | **latch** | *value*>	The clamp requirement.
	-sink_off_clamp <**0** | **1** | **any** | **Z** | **latch** | *value*>	The clamp requirement when the sink domain is off.
	-source_off_clamp <**0** | **1** | **any** | **Z** | **latch** | *value*>	The clamp requirement when the source domain is off.
	-receiver_supply *supply_set_ref*	The supply set used by receivers of the port.
	-driver_supply *supply_set_ref*	The supply set used by drivers of the port.
	-related_power_port *supply_port*	The supply port for the attributed port.
	-related_ground_port *supply_port*	The supply port for the attributed port.
	-related_bias_ports *supply_port_list*	The supply port(s) for the attributed port.
	-repeater_supply *supply_set_ref*	The supply set used by a repeater driving the port.
	-pg_type *pg_type_value*	The value of the `pg_type` for the port.
	-transitive <**TRUE** | **FALSE**>	When **-transitive** is **TRUE** (the default), the command applies to the descendants of the elements.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

The **set_port_attributes** command specifies information relevant to ports on the interface of power domains. This information is used to determine isolation and guard requirements for the port.

The set of ports attributed is determined as follows:

a) The ports attributed by **set_port_attributes** are determined as follows:
   1) **-elements** identifies a set of ports on the interface to the specified elements, excluding any supply ports.
   2) **-domains** identifies a set of ports on the interface to the specified domains, excluding any supply ports.
   3) **-ports** identifies a set of ports.

b) The union of the ports identified by **-elements**, **-domains**, and **-ports** is the set of candidate ports. Any port in the candidate set that meets any of the following criteria are removed from this union.
   1) The port is part of the **-exclude_ports** list.
   2) The port is a port on a design element specified in the **-exclude_elements** list.
   3) The port is a port on the interface of a power domain specified in the **-exclude_domains** list.

c) If a port on the interface of an element within the extent of a domain has port attributes defined for it and that port is connected to a port on the interface of the domain and the value of the element port is always the same as the value of the domain port to which it is connected (excluding any temporary delays in value transition due to implicit or explicit buffers on the connectivity path), then the attributes of the element port shall be applied to the domain port. Otherwise, the port attributes shall have no effect.

If **-model** is specified, the port attributes are applied to every design element that is an instance of the model. In this case, only names that are visible to the model may be referenced in arguments to this command.

-clamp_value specifies the clamp requirement of the sinks when the supply set connected to the source is in a power state with a corresponding simstate of **CORRUPT**.

-sink_off_clamp specifies the clamp requirement when the supply set connected to the sink is in a power state with a corresponding simstate of **CORRUPT**. **-source_off_clamp** specifies the clamp requirement when the supply set connected to the source is in a power state with a corresponding simstate of **CORRUPT**.

When a user-defined clamp *value* is specified for **-sink_off_clamp** or **-source_off_clamp**, it shall be a legal value for the type of the port. A clamp value of **any** specifies any clamp value legal for the port type is allowed. This value can be constrained to a specific value in a subsequent **set_isolation** command.

If any of **-related_power_port**, **-related_ground_port**, or **-related_bias_ports** is specified, an implicit supply set is created containing the supply nets connected to the ports. If the port being attributed is `in` mode, the implicitly created supply set is treated as the **-receiver_supply** set. If the port being attributed is `out` mode, the implicitly created supply set is treated as the **-driver_supply** set. If the port being attributed is `inout` mode, the implicitly created supply set is treated as both the **-receiver_supply** and **-driver_supply** set.

The **-pg_type** attribute can be specified on a supply port for use with automatic connection semantics. *pg_type_value* is a string denoting the supply port type.

NOTE— **-pg_type** only applies to supply ports and is the only attribute that applies to supply ports. All other attributes apply to ports that are not supply ports.

When **-receiver_supply** is attributed on a port, it specifies the supply of the logic reading the port. If the receiving logic is not within the logic design starting at the design root, it is presumed the receiver supply is the supply for the receiving logic.

When **-driver_supply** is attributed on a port, it specifies the supply of the logic driving the port. If the driving logic is not within the logic design starting at the design root, it is presumed the driver supply is the supply for the driver logic and the port is corrupted when the driver supply is in a simstate other than **NORMAL**.

When **-repeater_supply** is attributed on a port, it specifies a repeater shall be inserted to drive an output port and the repeater shall be connected to the specified supply set.

The following also apply:

- For **-receiver_supply**, when the receiving logic is within the logic design starting at the design root, it shall be an error if its supply is not the receiver supply.
- For **-driver_supply**, when the driver logic is within the logic design starting at the design root, it shall be an error if its supply is not the driver supply.
- It shall be an error if **-model** is specified and **-domains** and/or **-elements** is also specified.
- It shall be an error if a supply port is included in **-ports** and **-pg_type** is not specified.
- It shall be an error if **-repeater_supply** is attributed on a bidirectional port.
- It shall be an error if **-pg_type** is associated with a port that is not a supply port.
- It shall be an error if **-pg_type** is specified with any other attribute.
- It shall be an error if no argument is used.

A port-supply relationship can be annotated in HDL using the following attributes:

Attribute name: **UPF_related_power_pin** or **UPF_related_ground_pin**.

Attribute value: **"***supply_port_name***"**, where *supply_port_name* is a string whose value is the simple name of a port on the same interface as the attributed port.

Attribute name: **UPF_related_bias_pin**.

Attribute value: **"***list_of_supply_port_names***"**, where *list_of_supply_port_names* is a string whose value is a space-separated list of one or more simple names of port(s) on the same interface as the attributed port.

SystemVerilog or Verilog-2005 example (`power_pin`):

```
(* UPF_related_power_pin = "my_Vdd" *) output my_Logic_Port;
```

VHDL example (`power_pin`):

```
attribute UPF_related_power_pin of my_Logic_Port : signal is
"my_Vdd";
(* UPF_related_power_pin = "my_Vdd" *) output my_Logic_Port;
```

SystemVerilog or Verilog-2005 example (`bias_pin`):

```
(* UPF_related_bias_pin = "my_VNWELL my_VPWELL" *) output
my_Logic_Port;
```

VHDL example (`bias_pin`):

```
attribute UPF_related_bias_pin of my_Logic_Port : signal
is "my_VNWELL my_VPWELL";
```

Isolation clamp value port properties can be annotated in HDL using the following attributes:

Attribute name: **UPF_clamp_value**

Attribute value: <**"0"** | **"1"** | **"Z"** | **"latch"** | **"any"** | **"***value***"**>

SystemVerilog or Verilog-2005 example:

```
(* UPF_clamp_value = "1" *) output my_Logic_Port;
```

VHDL example:

```
attribute UPF_clamp_value of my_Logic_Port : signal is "1";
```

pg_type port properties can be annotated in HDL using the following attributes:

Attribute name: **UPF_pg_type** or **pg_type**

Attribute value: <**"primary_power"** | **"primary_ground"** | **"backup_power"** | **"backup_ground"** >

SystemVerilog or Verilog-2005 example:

```
(* UPF_pg_type = "primary_power" *) output myVddPort;
```

VHDL example:

```
attribute UPF_primary_power of myVddPort : signal
   is "primary_power";
```

Syntax example:

```
set_port_attributes
   -ports {my_Logic_Port} -clamp_value 1
```

6.46 set_power_switch

<table>
<tr><td>Purpose</td><td colspan="2">Extend an HDL model containing no more than acknowledge logic to complete switch definition</td></tr>
<tr><td>Syntax</td><td colspan="2">set_power_switch switch_name
-output_supply_port {port_name [supply_net_name]}
{-input_supply_port {port_name [supply_net_name]}}*
{-control_port {port_name}}*
{-on_state {state_name input_supply_port {boolean_function}}}*
[-supply_set supply_set_name]
[-on_partial_state {state_name input_supply_port {boolean_function}}]*
[-off_state {state_name {boolean_function}}]*
[-error_state {state_name {boolean_function}}]*</td></tr>
<tr><td rowspan="8">Arguments</td><td>switch_name</td><td>The name of the switch instance in the logic design relative to the active scope; this shall be a simple name.</td></tr>
<tr><td>-output_supply_port {port_name [supply_net_name]}</td><td>The output supply port of the switch and, optionally, the net where this port connects. The output port is added to the switch.</td></tr>
<tr><td>-input_supply_port {port_name [supply_net_name]}</td><td>The input supply port of the switch and, optionally, the net where this port is connected. The input ports are added to the switch.</td></tr>
<tr><td>-control_port {port_name}</td><td>A list of pre-existing ports on the switch that are the control ports.</td></tr>
<tr><td>-on_state {state_name input_supply_port {boolean_function}}</td><td>A named state, the input_supply_port for which this is defined, and its corresponding Boolean function.</td></tr>
<tr><td>-supply_set supply_set_name</td><td>Associate a supply set with a switch.</td></tr>
<tr><td>-on_partial_state {state_name input_supply_port {boolean_function}}</td><td>A named state, the input_supply_port for which this is defined, and its corresponding Boolean function where the switch is in a current-limited state.</td></tr>
<tr><td>-off_state {state_name {boolean_function}}</td><td>A named state and its corresponding Boolean function.</td></tr>
<tr><td>-error_state {state_name {boolean_function}}</td><td>Any error states, which if defined on the switch can be flagged during simulation or analysis.</td></tr>
<tr><td>Return value</td><td colspan="2">Return a <code>1</code> if successful or raise a <code>TCL_ERROR</code> if not.</td></tr>
</table>

The **set_power_switch** command extends a switch by adding the input supply port(s), output supply port(s), and states to the switch. The supply ports are connected to the specified nets.

switch_name shall denote a design element. This design element models any acknowledgement functionality.

Any control ports shall be pre-existing ports on the switch design element.

An input supply port without a connected supply net has the value **UNDETERMINED**.

When the Boolean expression (see 4.10) for

a) Any **-error_state** evaluates to *True*, the state of the switch's output port is set to **UNDETERMINED**.

b) An **-on_state** evaluates to *True*, the switch is on and in that named state, and the value on the input supply port for that **-on_state** is propagated to the output supply port. If more than one **-on_state** concurrently evaluates to *True*, the state of the switch's output port is set to **UNDETERMINED** unless the root supply drivers of the respective **-input_supply_port**s are the same.

c) All **-off_states** evaluate to *True*, the switch is *off* and the supply port's state is set to **OFF**.

d) One or more **-on_partial_state** evaluate to *True*, the switch is *partially on* and in that named state, and the value on the input supply port for that **-on_partial_state** is propagated to the output supply port and degraded to **PARTIAL_ON** if the input is **FULL_ON**.

e) All **-on_states** and all **-on_partial_state**s evaluate to *False* and no **-off_state** is defined, the switch is *off* and the supply port's state is set to **OFF**.

Otherwise, the state of the switch's output port is set to **UNDETERMINED**.

NOTE—When **-off_state** has a null Boolean expression, it can not evaluate *True*; therefore, when no **-on_state** or **-on_partial_state** evaluates *True*, the state of the switch's output port is set to **UNDETERMINED**.

Any **-on_state**, **-on_partial_state**, **-off_state**, or **-error_state** *boolean_function* shall be a SystemVerilog Boolean expression (see 4.10).

If a supply set is associated with a switch, it powers logic or timing control circuitry within the switch. The supply set is implicitly connected to the design element. When the supply set simstate is anything other than **NORMAL**, the state of the output supply port of a switch is **UNDETERMINED**. If a supply set is not associated with a switch, the output of the supply port implicitly operates in a **NORMAL** simstate.

The following also apply:

— Any name in a *boolean_function* needs to refer to a control port of the switch.

— All states not covered by the states on, off, and error are anonymous error states.

— If the Boolean expression (see 4.10) for more than one state evaluates to *True*, the switch shall be put into an anonymous error state.

— If the implementation of a switch can not be inferred, **map_power_switch** (see 6.32) can be used to specify it.

— If *net_name* is not specified for any of the switch's port definitions, **connect_logic_net** (see 6.12) or **connect_supply_net** (see 6.13) can be used to create the port connections.

— Each state name shall be unique.

— Any *port_name*s specified in this command are user-defined (e.g., `my_dogs_name`).

It shall be an error if the named switch instance does not exist in the logic design.

Syntax example:

```
set_power_switch hmacro/sw1
  -input_supply_port {i1 always_on_power}
  -output_supply_port {o1 switched_power}
  -control_port {on}
  -on_state {sw1_on i1 {on}}
  -off_state {sw1_off {~on}}
```

6.47 set_retention

<table>
<tr><td>Purpose</td><td colspan="3">Specify which objects in the domain need to be retention registers and set the save and restore signals for the retention functionality</td></tr>
<tr><td>Syntax</td><td colspan="3">set_retention <i>retention_name</i>
-domain <i>domain_name</i>
[-elements <i>element_list</i>] [-exclude_elements <i>exclude_list</i>]
[-retention_power_net <i>net_name</i>] [-retention_ground_net <i>net_name</i>]
[-retention_supply_set <i>ret_supply_set</i>] [-no_retention]
[-save_signal {{<i>logic_net</i> <high | low | posedge | negedge>}}
-restore_signal {{<i>logic_net</i> <high | low | posedge | negedge>}}]
[-save_condition {{<i>boolean_function</i>}}]
[-restore_condition {{<i>boolean_function</i>}}]
[-retention_condition {{<i>boolean_function</i>}}]
[-use_retention_as_primary]
[-parameters {< <RET_SUP_COR | NO_RET_SUP_COR> |
<SAV_RES_COR | NO_SAV_RES_COR> >*}]
[-instance {{<i>instance_name</i> [<i>signal_name</i>]}*}]
[-transitive <TRUE | FALSE>]
[-update]</td></tr>
<tr><td rowspan="10">Arguments</td><td>retention_name</td><td colspan="2">Retention strategy name.</td></tr>
<tr><td>-domain domain_name</td><td colspan="2">The domain for which this strategy is applied.</td></tr>
<tr><td>-elements element_list</td><td>The -elements option defines a list of objects: design elements, retention_list_name of elements lists (see 6.49), named processes, or sequential <code>reg</code> or signal names to which this strategy is applied.</td><td>R</td></tr>
<tr><td>-exclude_elements exclude_list</td><td>The -exclude_elements option defines a list of objects: design elements, retention_list_name of elements lists (see 6.49), named processes, or sequential <code>reg</code> or signal names that are not included in this strategy.</td><td>R</td></tr>
<tr><td>-retention_power_net net_name</td><td>This option defines the supply net used as the power for the retention logic inferred by this strategy.</td><td>R</td></tr>
<tr><td>-retention_ground_net net_name</td><td>This option defines the supply net used as the power for the retention logic inferred by this strategy.</td><td>R</td></tr>
<tr><td>-no_retention</td><td>When this option is used, the storage elements specified by this strategy shall not have retention capability added.</td><td>R</td></tr>
<tr><td>-retention_supply_set ret_supply_set</td><td>This option defines the supply set used to power the logic inferred by the retention_name strategy.</td><td>R</td></tr>
<tr><td>-save_signal {{<i>logic_net</i> <high | low | posedge | negedge>}} -restore_signal {{<i>logic_net</i> <high | low | posedge | negedge>}}</td><td>The -save_signal and -restore_signal options define a rooted name of a logic net or port and its active level or edge.The default sensitivity is high (for both -save_signal and -restore_signal).</td><td>R</td></tr>
<tr><td>-save_condition{{boolean_function}}</td><td>The -save_condition option defines a Boolean expression (see 4.10). The default is True.</td><td>R</td></tr>
<tr><td>-restore_condition {{boolean_function}}</td><td>The -restore_condition option defines a Boolean expression.The default is True.</td><td>R</td></tr>
</table>

Arguments	**-retention_condition** {{*boolean_function*}}	The **-retention_condition** option defines a Boolean expression. The default is *True*.	**R**
	-use_retention_as_primary	The **-use_retention_as_primary** option specifies that the storage element and its output are powered by the retention supply.	**R**
	-parameters {< <**RET_SUP_COR** | **NO_RET_SUP_COR**> | <**SAV_RES_COR** | **NO_SAV_RES_COR**> >*}	The **-parameters** option provides control over retention register corruption semantics. **RET_SUP_COR** activates and **NO_RET_SUP_COR** deactivates corruption of the normal mode register when retention supplies are **CORRUPT**; **RET_SUP_COR** is the default. **SAV_RES_COR** activates and **NO_SAV_RES_COR** deactivates corruption of the normal mode register during concurrent assertion of level-sensitive **save**, **save_condition**, **restore**, and **restore_condition**; **SAV_RES_COR** is the default.	**R**
	-instance {{*instance_name* [*signal_name*]}*}	The name of a technology library leaf cell instance and the optional name of the signal that it retains. If this instance has any unconnected supply ports or save and restore control ports, then these ports need to have identifying attributes in the cell model, and the ports shall be connected in accordance with this **set_retention** command.	**R**
	-transitive <**TRUE** | **FALSE**>	When **-transitive** is **TRUE** (the default), the command applies to the descendants of the elements.	
	-update	Use **-update** if the *retention_name* has already been defined.	
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.		

The **set_retention** command specifies a set of objects in the domain that need to be retention registers and identifies the save and restore behavior. If a design element is specified, all registers within the design element acquire the specified retention strategy. If a process is specified, all registers inferred by the process acquire the specified retention strategy. If a `reg`, signal, or variable is specified and that object is a sequential element, the implied register acquires the specified retention strategy. Any specified `reg`, signal, or variable that does not infer a sequential element shall not be changed by this command.

If **-elements** is specified, element names outside the extent of *domain_name* are excluded. When **-elements** is not specified, this is equivalent to using the elements list that defines the power domain. When used with **-update**, **-elements** is additive such that the set of elements or signals is the union of all calls of this command for a given strategy specifying any of these parameters.

-exclude_elements can also be used to define a list of storage elements that are not included in this strategy. When used with **-update**, **-exclude_elements** is additive such that the set of elements or signals excluded is the union of all calls of this command for a given strategy.

If retention power and retention ground nets are specified, an implicit retention supply set is created and used with the specified strategy. The retention power net serves the power function in the retention supply set and the retention ground net serves the ground function in the retention supply set. If the retention power net is specified but the retention ground net is not specified then the domain's primary supply set's ground function shall be used as the retention ground. If the retention ground net is specified but the retention power net is not specified then the domain's primary supply set's power function shall be used as the retention power. It shall be an error if a retention supply set is specified and a retention supply net is individually defined.

-retention_supply_set powers the register holding the retained value.

After the strategy has been completely applied, it shall be an error if the retention supply set is not defined for a strategy and the domain does not have a default *ret_supply_set*.

The retained value [see `retained_value` in b)] shall be the register's value at the time of the save event when **-save_condition** evaluates to *True*. The *save event* is the rising- or falling-edge of an edge-triggered save event or the trailing-edge of a level-sensitive save event.

The retained value is transferred to a register on the restore event when **-restore_condition** evaluates to *True*. The *restore event* is the rising- or falling-edge of an edge-triggered restore event or the trailing-edge of a level-sensitive restore event. The retained value and register state shall be **CORRUPTED** (see 6.51) when level-sensitive: save, save condition, restore, and restore condition are simultaneously active. A level-sensitive restore event has priority over any other register operation.

-restore_condition gates the restore event, defining the restore behavior of the register. For example, if the restore signal is `my_restore` and the condition is `!clock`, the register restores if and only if `my_restore` is active and the clock signal is low. If the value of the restore signal is active and the restore condition is unknown, or the restore signal is unknown and the restore condition is *True*, the register value is corrupted.

-save_condition gates the save event, defining the save behavior of the register. For example, if the save signal is `your_save` and the condition is `(!clock && !reset)`, the register saves if and only if `your_save` is active and `clock` and `reset` are low. If the value of the save signal is active and the save condition is unknown, or the save signal is unknown and the save condition is *True*, the register value is corrupted.

The *retained value* is corrupted if the **-retention_condition** does not hold *True* while the primary supply is not **NORMAL** (see 6.51).

-save_condition, **-restore_condition**, and **-retention_condition** shall only reference logic nets or ports rooted in the active scope.

If **-save** and **-restore** are not specified, the register is supplied by the retention supply set. In this case, the register value is corrupted according to the state of the retention supply or when the retention condition does not hold *True*.

-use_retention_as_primary powers the storage element and the output drivers of the register using the retention supply. The result of this is the simstate for the retention supply set is applied to the register's output. Inferred state elements shall be consistent with the **-use_retention_as_primary** constraint. No level-shifting or isolation shall be added to the design as a result of using **-use_retention_as_primary**.

During simulation, the behavior of each register to be retained is modified as follows:

a) The process sensitivity list for the register is expanded to include the following signals:
   1) Primary supply signals
   2) Retention supply signals, if **RET_SUP_COR** is active
   3) Restore signal; if the restore signal's sensitivity is posedge or negedge, then the process is sensitive to that edge; otherwise, the restore signal's sensitivity is high or low, so the process is sensitive to any change on the restore signal
   4) Restore condition signals
   5) Save signal and save condition signals, only if neither the restore nor save signal is edge-sensitive, and **SAV_RES_COR** is active

b) The body of the process is modified as illustrated by the following Verilog:


```
   if ( <primary_supplies_are_normal>
'ifdef RET_SUP_COR
         && <retention_supplies_are_normal>
'endif
         && ! ( <restore_signal_is_active> && <restore_condition_is_true> ) )
      begin
         <original_process_body>
      end
   else if ( <primary_supplies_are_normal>
'ifdef RET_SUP_COR
         && <retention_supplies_are_normal>
'endif
         && <restore_signal_is_active> && <restore_condition_is_true>
'ifdef SAV_RES_COR
         && ! ( <save_signal_is_active> && <save_condition_is_true> )
'endif
         )
      <register_value> <= <retained_value>
   else
      <register_value> <= <corrupt_value>
```

c) An additional process is created for the saved state, with a process sensitivity list comprised of the following signals:

   1) Retention supply signals
   2) Retention condition signals
   3) Save signal; if the save signal's sensitivity is posedge or negedge, then the process is sensitive to that edge; otherwise, the save signal's sensitivity is high or low, so the process is sensitive to any change on the save signal
   4) Save condition signals
   5) Restore signal and restore condition signals, only if neither the restore nor save signal is edge-sensitive, and **SAV_RES_COR** is active

d) The body of the additional process is constructed as illustrated by the following Verilog:

```
   if ( <retention_supplies_are_normal> && <retention_condition_is_true>
         && ! ( <save_signal_is_active> && <save_condition_is_true> ) )
      begin
      end
   else if ( <retention_supplies_are_normal>
         && <save_signal_is_active> && <save_condition_is_true>
'ifdef SAV_RES_COR
         && ! ( <restore_signal_is_active> && <restore_condition_is_true> )
'endif
         )
      <retained_value> <= <register_value>
   else
      <retained_value> <= <corrupt_value>
```

The elements requiring retention can be attributed in HDL as shown in 6.49.

Syntax example:

```
set_retention my_retention_strategy
   -retention_supply_set PDA_ret_supply
   -save {my_save posedge}
   -restore {my_restore posedge}
```

6.48 set_retention_control

Purpose	Specify the control signals and assertions for a previously defined retention strategy	
Syntax	**set_retention_control** *retention_name* **-domain** *domain_name* **-save_signal** {{*net_name* <**high** | **low** | **posedge** | **negedge**>}} **-restore_signal** {{*net_name* <**high** | **low** | **posedge** | **negedge**>}} [**-assert_r_mutex** {{*net_name* <**high** | **low** | **posedge** | **negedge**>}}]* [**-assert_s_mutex** {{*net_name* <**high** | **low** | **posedge** | **negedge**>}}]* [**-assert_rs_mutex** {{*net_name* <**high** | **low** | **posedge** | **negedge**>}}]*	
Arguments	*retention_name*	Retention strategy name (used only for reporting).
	-domain *domain_name*	The domain for which this strategy is applied.
	-save_signal *save_net*	The signal that causes the register values to be saved into the shadow registers.
	-restore_signal *restore_net*	The signal that causes the register values to be restored from the shadow registers.
	-assert_r_mutex {{*net_name* <**high** | **low** | **posedge** | **negedge**>}}	The restore signal for assertion.
	-assert_s_mutex {{*net_name* <**high** | **low** | **posedge** | **negedge**>}}	The save signal for assertion.
	-assert_rs_mutex {{*net_name* <**high** | **low** | **posedge** | **negedge**>}}	Both signals (save and restore) for assertion.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

The **set_retention_control** command allows the specification of the retention control signal and sense separate from the **set_retention** command (see 6.47) for those situations where the retention strategy is known, but the control signals are not known until later. As the assertions are related to the save and restore signals, they can also be specified with this command.

Compatibility note: Except that the **set_retention_control** command is executed within the active scope, the semantics here are equivalent to having specified the retention control signals, senses, and assertions with the **set_retention** command.

The **set_retention_control** command can also be used to specify any assertion options. Each assert retention option creates one or more assertions, which verification tools can trigger when the indicated RTL signals are active simultaneously with: the restore signal, **-assert_r_mutex**; the save signal, **-assert_s_mutex**; or both signals, **-assert_rs_mutex**. If **-assert_rs_mutex** does not specify a list of signals, this indicates the save and restore signals themselves are mutually exclusive.

The following also apply:

— The save signal shall be an existing net or port in the design.

— The restore signal shall be an existing net or port in the design.

Syntax example:

```
set_retention test_retention
 -domain PDA
 -retention_power_net ret_pwr

set_retention_control test_retention
 -domain PDA
 -save_signal {save_a high}
 -restore_signal {restore_a low}
 -assert_rs_mutex
 -assert_rs_mutex {reset_a low}
 -assert_s_mutex {clock_a posedge}
```

The above command creates three assertions. Then to ensure

a) Save and restore are mutually exclusive, use:

```
(save_a && !restore_a) == 0
```

b) Save and restore are mutually exclusive with a low level sensitive `reset_a`, use:

```
((save_a | !restore_a) && !reset_a) == 0
```

c) Save and the positive edge of signal `clk` are mutually exclusive, use:

```
save_a && (posedge clock_a) == 0
```

6.49 set_retention_elements

Purpose	Create a named list of elements to be used in a **set_retention** or **map_retention_cell** command	
Syntax	**set_retention_elements** *retention_list_name* [**-elements** *element_list*] [{**-applies_to** <**required** | **not_optional** | **not_required** | **optional**>}] [**-exclude_elements** *exclude_list*] [**-retention_purpose** <**required** | **optional**>] [**-transitive** <**TRUE** | **FALSE**>]	
Arguments	*retention_list_name*	A simple name; this shall be unique within the active scope.
	-elements *element_list*	A list of rooted names: design elements, named processes, sequential `regs`, or signal names.
	-applies_to <**required** | **not_optional** | **not_required** | **optional**>	Filter elements based on the **UPF_retention** attribute value.
	-exclude_elements *exclude_list*	A list of rooted names: design elements, named processes, sequential `regs`, or signal names.
	-retention_purpose <**required** | **optional**>	The intended retention use of *retention_list_name*.
	-transitive <**TRUE** | **FALSE**>	When **-transitive** is **TRUE** (the default), the command applies to the descendants of the elements.
	-expand <**TRUE** | **FALSE**>	When **-expand** is **TRUE**, elements are expanded as though every register that otherwise would be included had been specified directly in *element_list*. The default is **FALSE**.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

The **set_retention_elements** command defines a list of objects that can then be used in **set_retention** and **map_retention_cell** commands (see 6.47 and 6.33).

-applies_to filters the *effective_element_list,* removing any elements that do not have a **UPF_retention** attribute value consistent with the selected filter choice: **required**, **not_optional**, **not_required**, or **optional**.

required matches all elements that have the **UPF_retention** attribute value **"required"**.

optional matches all elements that have the **UPF_retention** attribute value **"optional"**.

not_required matches all elements that do not have the **UPF_retention** attribute value **"required"**.

not_optional matches all elements that do not have the **UPF_retention** attribute value **"optional"**.

It shall be an error if a *retention_list_name* created with **-retention_purpose required** is not used in a retention strategy when a domain containing an element from *retention_list_name* has a defined retention strategy.

Syntax example:

```
set_retention_elements ret_chk_list
   -elements {proc_1 sig_a}
```


6.50 set_scope

Purpose	Specify the active UPF scope	
Syntax	**set_scope** *instance*	
Arguments	*instance*	The instance that becomes the active scope upon completion of the command.
Return value	Return the active scope prior to execution of the command as a full path string relative to the active design top if successful or raise a `TCL_ERROR` if it fails (e.g., if the instance does not exist).	

If the **set_scope** command is called with no arguments or the UPF scope is not set, the scope is set to the root of the design.

If *instance* is `.`, the scope remains unchanged. If *instance* is `..`, the context is moved up one level in the instance hierarchy. If *instance* is `/`, the scope is set to the root of the design. See also 6.38.

Syntax examples:

```
set_scope foo/bar

set_scope ..
```

6.51 set_simstate_behavior

Purpose	Specify the simulation simstate behavior for a model or library	
Syntax	**set_simstate_behavior <ENABLE | DISABLE>** [**-lib** *name*] [**-model** *list*]	
Arguments	<**ENABLE | DISABLE**>	Define if the UPF simstate behavior shall be enabled for the specified model(s).
	-lib *name*	The library name.
	-model *list*	One or more model names.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

This command specifies the simstate behavior for every design element that is an instance of a model.

If **ENABLE** is specified, the simstate simulation semantics are applied for every supply set automatically connected to an instance of the model. See also 5.6.

a) If there is a single supply set connected, the simstates for that supply set are applied.

b) When no supply set is connected, but there are connected supply nets, an anonymous supply set is created containing the supply nets and the default simstates for that supply set are applied.

c) When there are multiple supply sets connected, the simstates of all supply sets are applied.

d) For a combination of connected supply nets and supply sets, an anonymous supply set is created containing the supply nets, and the default simstates of the anonymous supply set and all other connected supply sets are applied.

If **-model** is not defined and **-lib** is specified, the simstate behavior is defined for all models in *name*.

It shall be an error if

— **-model** is specified and any of the model(s) cannot be found.
— **DISABLE** is specified and the model has no supply ports.
— A model has conflicting simstate behaviors specified.
— Any design element that is an instance of a model that has simstate behavior **ENABLED** is connected to more than one supply set.

Simstate behavior of a module can be enabled or disabled in HDL using the following attributes:

Attribute name: **UPF_simstate_behavior**

Attribute value: <**"ENABLE"** | **"DISABLE"**>

SystemVerilog or Verilog-2005 example:

```
(* UPF_simstate_behavior = "ENABLE" *) module my_adder;
```

VHDL example:

```
attribute UPF_simstate_behavior of my_adder : entity is
"ENABLE";
```

Syntax example:

```
set_simstate_behavior ENABLE -lib library1 -model ANDX7_non_power_aware
```

6.52 upf_version

Purpose	Specify the UPF version for interpreting subsequent commands
Syntax	**upf_version** [*string*]
Arguments	*string* The UPF version number.
Return value	If called with an argument, return the argument. If called with no arguments, return the current version number. If the call fails, raise a `TCL_ERROR`.

As the UPF standard matures, new updated versions of the UPF standard can occur. The **upf_version** command can be used to specify the UPF version the input for processing subsequent UPF commands. **upf_version** may be called multiple times; any previously processed UPF commands are not affected.

NOTE—**load_upf_protected** (see 6.29) can be used to preserve the version number.

The version numbers are

UPF1.0 returns `1.0`.

IEEE1801_2009 returns `2.0`.

Syntax example:

```
upf_version 2.0
```


6.53 use_interface_cell

Purpose	Specify the functional model and a list of implementation targets for isolation and level-shifting
Syntax	**use_interface_cell** *interface_implementation_name* **-strategy** *list_of_isolation_level_shifter_strategies* **-domain** *domain_name* **-lib_cells** *lib_cell_list* [**-map** {{*port net_ref*}*}] [**-elements** *element_list*] [**-exclude_elements** *exclude_list*] [**-applies_to_clamp** <**0** | **1** | **any** | **Z** | **latch** | *value*>] [**-update_any** <**0** | **1** | **known** | **Z** | **latch** | *value*>] [**-force_function**] [**-inverter_supply_set** *list*]
Arguments	*interface_implementation_name* — The interface cell implementation strategy. **-strategy** *list_of_isolation_level_shifter_strategies* — The isolation or level-shifter strategy as defined by **set_isolation** and **set_level_shifter**. **-domain** *domain_name* — The domain in which the strategies are defined. **-lib_cells** *lib_cell_list* — A list of library cell names. **-map** {{*port net_ref*}*} — The *port* and the net (*net_ref)* connections. **-elements** *element_list* — A list of ports from the *list_of_isolation_level_shifter_strategies* to which the command applies. **-exclude_elements** *exclude_list* — A list of ports from the *list_of_isolation_level_shifter_strategies*. **-applies_to_clamp** <**0** | **1** | **any** | **Z** | **latch** | *value*> — Only ports that have the specified clamp value are mapped. **-update_any** <**0** | **1** | **known** | **Z** | **latch** | *value*> — What is now the known clamp value when **-applies_to_clamp** is **any**. **-force_function** — The first model in *model_list* is used as the functional specification of isolation behavior. **-inverter_supply_set** *list* — The supply set implicitly connected to any inversion logic required by an isolation signal connection.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.

The **use_interface_cell** command provides user control for the integration of isolation and level-shifting. The command specifies the implementation choices through **-lib_cells** and the functional isolation behavior to be used if **-force_function** is specified.

NOTE—Unlike **map_interface_cell** and **map_retention_cell**, **use_interface_cell** can be used to manually map isolation, level-shifting, and combined isolation level-shifting cells.

When **-force_function** is specified the first model in *lib_cell_list* shall be used as the functional model. The isolation sense specification for the isolation strategy is ignored when **-force_function** is specified. It is erroneous if the functional model clamps to a value that is different to the previously specified port clamp value.

-elements selects the ports from the specified list of strategies to which the mapping command is applied. If **-elements** is not specified, all ports inferred from the list of strategies shall have the mapping applied. When **-applies_to_clamp** is specified, this command is applied only to the ports with that clamp value.

When **-applies_to_clamp** is **any**, **-update_any** shall be used to specify the clamp value after mapping. An **-update_any** value of **known** specifies that the isolation function is more complex than can be specified by a single value.

-map connects the specified *net_ref* to a *port* of the model. A *net_ref* may be one of the following:

a) A logic net name

b) A supply net name

c) One of the following symbolic references

1) **isolation_supply_set.***function_name*

function_name refers to the supply net corresponding to the function it provides to the **isolation_supply_set**.

2) **isolation_supply_set**[*index*]**.***function_name*

i) *index* is a non-negative integer corresponding to the position in the **isolation_supply_set** list specified for the isolation strategy.

ii) The **isolation_supply_set** *index* shall be specified if the isolation strategy specified more than one **isolation_supply_set**.

3) isolation_signal

i) Refers to the isolation signal specified in the corresponding isolation strategy.

ii) To invert the sense of the isolation signal the Verilog bit-wise negation operator ~ can be specified before the isolation_signal. The logic inferred by the negation shall be implicitly connected to the **inverter_supply_set** if specified, otherwise the **isolation_supply_set** shall be used.

4) **isolation_signal**[*index*]

i) *index* is a non-negative integer corresponding to the position in the **isolation_signal** list specified for the isolation strategy.

ii) The **isolation_signal** *index* shall be specified if the isolation strategy specified more than one **isolation_signal**.

iii) To invert the sense of the isolation signal the Verilog bit-wise negation operator ~ can be specified before the isolation_signal. If the **isolation_signal** is being inverted then the **inverter_supply_set[***index***]** if specified shall be implicitly connected to the inferred inverter, otherwise the **isolation_supply_set**[*index*] shall be used.

5) **input_supply_set.***function_name*

function_name refers to the supply net corresponding to the function it provides to the level-shifter **input_supply_set**.

6) **output_supply_set.***function_name*

function_name refers to the supply net corresponding to the function it provides to the level-shifter **output_supply_set**.

7) **internal_supply_set.***function_name*

function_name refers to the supply net corresponding to the function it provides to the level-shifter **internal_supply_set**.

The **-map** command shall not reference the data input port or the data output port. The input port shall be connected to the data input for the interface cell and the output port connected to the data output for the interface cell.

It shall be an error if

- *domain_name* does not indicate a previously created power domain.
- A port in the *port_list* is not covered by a **set_isolation** command.
- *list_of_isolation_level_shifter_strategies* is a null list.
- **-force_function** is not specified and none of the specified models in *lib_cell_list* implements the functionality specified by the corresponding *isolation_strategy* and port attributes.
- **-update_any** is specified and **-applies_to_clamp** is not **any**.
- After completing the *port* and *net_ref* connections and the data input and output connections, any port is unconnected.
- Ports specified by **-elements** are not included in all specified strategies.
- More than one isolation strategy is specified.
- More than one level-shifter strategy is specified.

Syntax example:

```
use_interface_cell my_interface -strategy {ISO1 LS1} -domain PD1\
-elements {top/moduleA/port1 top/moduleA/port2 top/moduleA/port3}
```

7. Queries

This clause documents the syntax for each of the **query_*** commands (and **find_objects**). Each return value is a Tcl string-object that is a list of defined objects, all options of the object, or individual settings for the object. The names returned (*Return value*s) are relative to the active scope. If there are any names to be returned that are not rooted in the active scope, the query shall raise an "out of scope" error. This could occur, for example, if the power domain of an object was queried, but the scope of the domain that was to be returned via this query was not visible in the scope as specified by the active **set_scope** command (see 6.50).

— Each query in this clause consists of a keyword followed by one or more parameters. All parameters begin with a hyphen (-). The meta-syntax for the description of the syntax rules uses the conventions shown in Table 5.

— For general information on how errors are handled, see 6.5.

— Since the queries only return information about the active design, they have no implementation or simulation semantics.

— Queries are not guaranteed to, and in virtually all situations do not, return information in the order that their corresponding command (see Clause 6) supplied it.

— Additional information can be returned by the queries, for example if a design element is added to a domain using **add_domain_elements**, then **query_power_domain** also returns this added element. Command refinement reconciliation is incorporated in query return values (see 6.4).

— All **query_*** commands search from the active scope down, unless otherwise stated.

find_objects, **query_upf**, and all **query_*** commands that accept the **-non_leaf** and **-leaf_only** options can be interpreted differently between tools depending upon the library source. For example, a simulation tool may have a hierarchical model representation of a IP block that is not returned if **-leaf_only** is specified (the search would traverse through this boundary to find leaf cells). However, an implementation tool could have this IP block represented as a timing abstract and thus could be treated as a leaf cell.

Query commands that have the **-detailed** option provide the ability to return information as a list of {*key value*} pairs. The *key* is derived from the argument name of the corresponding command (see Clause 6) that is being queried.

Commands that have a Boolean option, such as **-include_scope**, shall have a Boolean return flag of `1` if the option was specified and `0` if it was not. For commands that have arguments that accept Tcl lists, the query returns the entire list, e.g., `-ports` *`list`* produces the **-detailed** output of the form `{ports {{port_list_index_0} {port_list_index_1}{...}}}`. For commands that have arguments that have lists containing optional arguments, e.g., `-supply` `{`*`supply_set_handle`* [*`supply_set_ref`*]} the query returns the optional argument (*`supply_set_ref`*) or a null string if the optional argument has not been specified, e.g., `{supply {{supply_set_handle_index_0 {}} {supply_set_handle_index_1 {supply_set_ref}}...}`.

When the **-detailed** argument to a query returns an argument for which no value has been specified, then the default value is returned. If there is no default, then a null list ({}) is returned.

Compliance requirement: A tool compliant to this standard shall support the **find_objects** command. A tool compliant to this standard may support the **query_*** commands in this clause.

NOTE—These **query_*** commands do not make up the *power intent* of a design; they are only used for *querying* the design database and are included in this standard to enable portable, user-specified query procedures across tools that are compliant to this standard.

7.1 find_objects

Purpose	Find logical hierarchy objects within a scope	
Syntax	**find_objects** *scope* **-pattern** *search_pattern* [**-object_type** <**inst** | **port** | **net** | **process**>] | [**-direction** <**in** | **out** | **inout**>] [**-transitive** <**TRUE** | **FALSE**>] [**-regexp** | **-exact**] [**-ignore_case**] [**-non_leaf** | **-leaf_only**]	
Arguments	*scope*	The search is restricted to the specified scope.
	-pattern *search_pattern*	The string used for searching. By default, *search_pattern* is treated as an Tcl `glob` expression.
	-object_type <**inst** | **port** | **net** | **process**>	Limits the objects returned. By default, design elements, named processes, ports, and nets are returned; this can be restricted by specifying a specific **-object_type**. **inst** does not return named processes.
	-direction <**in** | **out** | **inout**>	If **-object_type** is **port**, then **-direction** can be used to restrict the directions of the returned ports.
	-transitive <**TRUE** | **FALSE**>	When **-transitive** is **TRUE**, the command applies to the descendants of the elements; the default is **FALSE**.
	-regexp | **-exact**	**-regexp** enables support for regular expression in the specified *search_pattern*. **-exact** disallows wildcard expansion on the specified *search_pattern*. If neither **-regexp** or **-exact** are specified, then *search_pattern* is interpreted as a Tcl `glob` expression.
	-ignore_case	Performs case-insensitive searches. By default, all matches are case sensitive.
	-non_leaf | **-leaf_only**	If **-non_leaf** is specified, only non-leaf design elements (elements that have children) are returned; if **-leaf_only** is specified, only leaf-level design elements (elements without children) are returned. By default, both leaf and non-leaf design elements are returned.
Return value	Return a list of the found hierarchical names (relative to the active scope); when nothing is found, a *null string* is returned.	

The **find_objects** command searches for design elements, nets, or ports that are defined in the HDL. This command works on the logical hierarchy and only searches in the *scope* (or in and below the *scope* when **-transitive** is specified).

NOTE—To find UPF objects, such as isolation logic or retention elements, use the corresponding **query_*** commands.

The **-non_leaf** and **-leaf_only** options can be interpreted differently between tools, depending upon the library source. For example, a simulation tool may have a hierarchical model of a IP block, which to the implementation tool is represented as a timing abstract and, thus, treated as a leaf cell. A module may be tagged as a leaf cell by using **set_module_attribute** (see 6.37).

The following conditions also apply:

— The specified *scope* cannot start with `..` or `/`, i.e., **find_objects** needs to be referenced from the active scope, and reside in the active scope or below it.

— If *scope* is specified as **.** (a dot), the active scope is used as the root of the search.
— All elements returned are referenced to the active scope.
— It shall be an error if *scope* is not defined in the active scope.

Syntax examples:

```
find_objects A/B/D -pattern *BW1*
   -object_type inst
   -transitive
```

7.1.1 Pattern matching and wildcarding

To improve usability and allow multiple objects (design elements, ports, etc.) to be easily specified without onerous verbosity, pattern matching (wildcarding) is allowed (only) in **find_objects** and **query_upf** (see 7.2). Pattern matching is supported using the Tcl `glob` style, matching against the symbols in the scope rather than filenames. For `glob`-style wildcarding, the following special operators are supported.

? matches any single character.

* matches any sequence of zero or more characters.

[*chars***]** matches any single character in *chars*. If *chars* contains a sequence of the form `a-b`, any character between `a` and `b` (inclusive) shall match.

x matches the character *x*.

{*a*, *b*, ...} Matches any of the strings *a*, *b*, etc.

Tcl regular expression matching is described in the Tcl documentation for `re_syntax` [B4].

7.1.2 Wildcarding examples

Table 7 shows the pattern match for each of the following examples of **find_objects**.

```
find_objects top -pattern {a}
find_objects top -pattern {bc[0-3]}
find_objects top -pattern {e*}
find_objects top -pattern {d?f}
find_objects top -pattern {g\[0\]}
```

Table 7—Pattern matches

`a`	Only matches a design element called `a` in the active scope.
`bc[0-3]`	Matches any design element called `bc` followed by a numerical value from `0` to `3`, i.e., `bc0`, `bc1`, `bc2`, and `bc3`.
`e*`	Matches any design element starting with `e`, i.e., `e12`, `eab`, `ef`, etc.
`d?f`	Matches any design element starting with `d` followed by another character and ending in `f`, i.e., `daf`, `d4f`, etc.
`g\[0\]`	Matches a design element called `g[0]`.

NOTE 1—The use of the Tcl quote semantics of "{*string*}" in the example illustrates an effective means to pass characters that would otherwise be "special" to a Tcl interpreter.

NOTE 2—To select the four bits (`0` to `3`) of the bus `my_bus`, use the Tcl expression `{my_bus\[[0-3]\]}`.

7.2 query_upf

<table>
<tr><td>Purpose</td><td colspan="2">Find objects (including UPF created or inferred objects) in the logical hierarchy</td></tr>
<tr><td>Syntax</td><td colspan="2">query_upf <<i>domain_name</i> | <i>scope</i>>
    -pattern <i>search_pattern</i>
    [-object_type <inst | port | supply_port | net | supply_net | supply_set>]
    [-inst_type <level_shifter | isolation_cell | switch_cell | retention_cell | all>] |
    [-direction <in | out | inout>]
    [-transitive <TRUE | FALSE>]
    [-regexp | -exact]
    [-ignore_case]
    [-non_leaf | -leaf_only]</td></tr>
<tr><td rowspan="9">Arguments</td><td>domain_name | scope</td><td>Either a power domain or a scope can be specified. If a power domain is specified, the search is restricted to that power domain; otherwise. the search is restricted to the specified scope.</td></tr>
<tr><td>-pattern search_pattern</td><td>The string used for searching. By default, search_pattern is treated as an Tcl <code>glob</code> expression.</td></tr>
<tr><td>-object_type <inst | port | supply_port | net | supply_net | supply_set></td><td>Limits the objects returned. By default, all objects are returned.</td></tr>
<tr><td>-inst_type <level_shifter | isolation_cell | switch_cell | retention_cell | all></td><td>If -object is inst, this option limits the type of design elements returned to be level-shifter, isolation, switch, or retention cells. The default is all, which returns all design elements.</td></tr>
<tr><td>-direction <in | out | inout></td><td>If -object is port, then -direction can be used to restrict the directions of the returned ports.</td></tr>
<tr><td>-transitive <TRUE | FALSE></td><td>When -transitive is TRUE, the command applies to the descendants of the elements; the default is FALSE.</td></tr>
<tr><td>-regexp | -exact</td><td>-regexp enables support for regular expression in the specified search_pattern. -exact disallows wildcard expansion on the specified search_pattern. If neither -regexp or -exact are specified, then search_pattern is interpreted as a Tcl <code>glob</code> expression.</td></tr>
<tr><td>-ignore_case</td><td>Performs case-insensitive searches. By default, all matches are case sensitive.</td></tr>
<tr><td>-non_leaf | -leaf_only</td><td>If -non_leaf is specified, only non-leaf design elements are returned; if -leaf_only is specified, only leaf-level design elements are returned. By default, both leaf and non-leaf design elements are returned.</td></tr>
<tr><td>Return value</td><td colspan="2">Return a list of the found objects; when no object is found, a null string is returned.</td></tr>
</table>

The **query_upf** command searches for design elements, nets, supply nets, ports, and supply ports in and below the *scope* or within the extent of a *domain_name*. This command works on the logical hierarchy and can be executed post-UPF annotation. Different tools process the power intent at different times, e.g., a tool could build a representation of the power intent directly after loading the UPF constraints, whereas another tool could evaluate the different constraints as and when required. This means tools can issue an error if a search is performed at a point in the flow where the UPF intent has not been implemented.

The **query_upf** command works on the logical hierarchy from a domain-centric or hierarchy-centric approach. A *domain-centric approach* restricts the search to design elements, net, or ports that are logically within the extent of the specified *domain_name*. A *hierarchy-centric approach* searches in the *scope* only, or in and below the *scope* when **-transitive** is specified.

A domain-centric search examines all logical levels that are members of the specified domain. Based on Figure 7 and Figure 8, the command `query_upf {PD1} -pattern *` looks for any object (port, net, or design element) matching the specified string in the logical hierarchies `A`, `A/B`, `A/C`, or `A/B/D/F`.

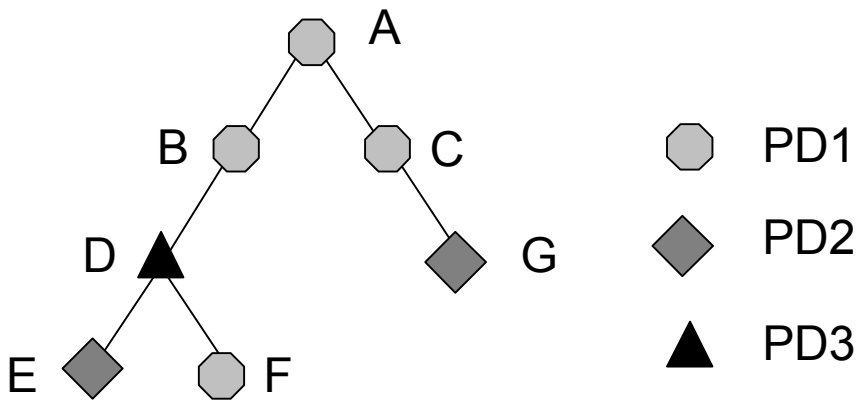

Figure 7—Logical hierarchy

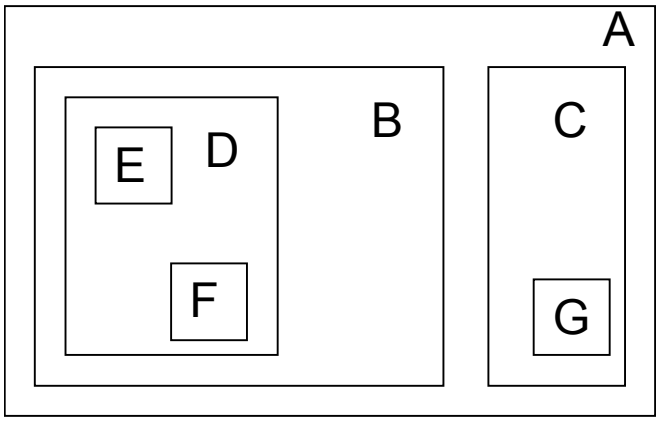

Figure 8—Physical layout

If searching for inputs into `PD3`, the command

```
query_upf {PD3} -pattern * -object_type port -direction in
```

returns any inputs from {`B->D`, `F->D`, and `E->D`}.

-inst_type only returns design elements of a particular type. For example, to find all level-shifters in the domain `PD3`, the following **query_upf** command could be used.

```
query_upf {PD3} -pattern * -inst_type level_shifter -object inst
```

A domain-centric search examines all logical levels that are members of the *domain_name*. Based on Figure 7 and Figure 8, the command `query_upf {PD1} -pattern *BW1*` looks for any object (port, supply port, net, supply net, or design element) that matched the specified string in the logical hierarchies `A`, `A/B`, `A/C`, or `A/B/D/F`.

If searching for inputs into `PD3`, the command

```
query_upf {PD3} -pattern * -object port -direction in
```

returns any inputs from {`B->D`, `F->D`, and `E->D`}.

The following conditions also apply:

— **-transitive** is ignored in a domain-centric search.

— The specified *domain_name* or *scope* cannot start with `..` or `/`, i.e., **find_objects** needs to be referenced from the active scope, and reside in the active scope or below it.

— All elements returned are referenced to the active scope.

— If *domain_name* or *scope* is specified as **.** (a dot), the active scope is used as the root of the search.

Syntax examples:

```
query_upf A/B/D \
-pattern *BW1* \
-object inst \
-transitive
```

7.3 query_associate_supply_set

Purpose	Query a previously defined supply set association	
Syntax	**query_associate_supply_set** *supply_set_ref* [**-detailed**]	
Arguments	*supply_set_ref*	Specifies the name of the supply set *supply_set_ref* to query.
	-detailed	Returns the supply set association information as a list of {*key value*} pairs, where *key* is the name of an argument of the **associate_supply_set** command (any **-** prefixes are removed) and *value* is the value of that argument. Valid keys are **supply_set_ref** and **handle**.
Return value	There are two distinct return structures. a) If **-detailed** is not specified then the supply set association shall be returned in the format of the corresponding **associate_supply_set** command. b) If **-detailed** is specified then the supply set association shall be returned as {*key value*} pairs.	

The **query_associate_supply_set** commands queries the association between a supply set and a domain or strategy.

If a supply set is associated with a domain using the following **associate_supply_set** command:

```
associate_supply_set some_supply_set
    -handle U1/PD1.rolf_mem_ss
```


then `query_associate_supply_set some_supply_set` returns the corresponding **associate_supply_set** command as defined above. If the **-detailed** option is specified the association shall be returned as {*key value*} pairs, i.e.,

```
{supply_set_ref some_supply_set} {handle U1/PD1.rolf_mem_ss}
```

Syntax example:

```
query_associate_supply_set some_supply_set
```

7.4 query_bind_checker

Purpose	Query a previously defined checker module	
Syntax	**query_bind_checker** *instance_name* [**-detailed**]	
Arguments	*instance_name*	Specifies the *instance_name* of the checker module to query. If * is specified, then all checker modules defined shall be returned.
	-detailed	Returns the checker information as a list of {*key value*} pairs, where *key* is the name of an argument of the **bind_checker** command (any **-** prefixes are removed) and *value* is the value of that argument. Valid keys are **instance_name**, **elements**, **module**, **bind_to**, **arch**, and **ports**.
Return value	There are three distinct return structures. a) If * is specified for *instance_name* then the instance names of all previously defined checker modules shall be returned as a Tcl list (a *null string* shall be returned if no power states are defined). b) If *instance_name is* specified then the checker module information for the specified instance shall be returned. c) If **-detailed** is specified then the checker module information for the specified instance *instance_name* shall be returned as {*key value*} pairs.	

The **query_bind_checker** command queries any previously defined bind checkers in and below the active scope.

If a bind checker was previously defined as

```
bind_checker chk_p_clks
-module assert_partial_clk
-bind_to aars
-ports {{prt1 clknet2} {port3 net4}}
```

then `query_bind_checker chk_p_clks` returns the corresponding **bind_checker** command as defined above. `query_bind_checker *` returns the instance names of all the previously defined bind checkers, i.e., `{chk_p_clks}` and if the `-detailed` option is used, i.e., `query_bind_checker chk_p_clks -detailed` then the state information is returned as `{{instance_name chk_p_clks} {elements {}} {module assert_partial_clk} {bind_to aars} {arch {}} {ports {}}`.

It shall be an error if **-detailed** is specified and * is specified for *instance_name*.

Syntax example:

```
query_bind_checker *
```

7.5 query_cell_instances

Purpose	Query the instances of a mapped cell within the active scope	
Syntax	**query_cell_instances** *cell_name* [**-domain** *domain_name*]	
Arguments	*cell_name*	The name of the cell or module to find.
	-domain *domain_name*	Limits the search to the instances in the domain specified.
Return value	Return a list of the instances that use the named cell or module. The list may be empty.	

The **query_cell_instances** command can locate all uses of a particular cell.

Syntax example:

```
//To find all instances of a cell named rolf in the active scope
query_cell_instances rolf
```

7.6 query_cell_mapped

Purpose	Query which cell is mapped to this instance	
Syntax	**query_cell_mapped** *instance_name*	
Arguments	*instance_name*	The name of the instance.
Return value	Return a cell name.	

The **query_cell_mapped** command can identify the cell that is used for the named instance *instance_name*.

Syntax example:

```
query_cell_mapped top/a/my_inst
```

7.7 query_composite_domain

Purpose	Query a composite domain	
Syntax	**query_composite_domain** *composite_domain_name* [**-detailed**]	
Arguments	*composite_domain_name*	Specifies the *composite_domain_name* to query. If ***** is specified then the name of all composite domains shall be returned as a Tcl list.
	-detailed	Returns the composite domain information as a list of {*key value*} pairs, where *key* is the name of an argument of the **create_composite_domain** command (any **-** prefixes are removed) and *value* is the value of that argument. Valid keys are **composite_domain_name**, **subdomains**, and **supply**.
Return value	There are three distinct return structures. a) If ***** is specified for *composite_domain_name* then all previously defined composite domains shall be returned as a Tcl list (a *null string* shall be returned if no power states are defined). b) If *composite_domain_name* is specified (and it is not *****) then the composite domain information shall be returned. If the specified *composite_domain_name* is not a composite domain, but is a non-composite domain, then a `0` shall be returned to indicate this. c) If **-detailed** is specified then the composite domain information for the specified *composite_domain_name* shall be returned as {*key value*} pairs.	

The **query_composite_domain** command returns any previously defined composite domains, in and below the active scope. If a composite domain is defined as

```
create_composite_domain dom_combined
   -subdomains {IP1/PDtop IP2/SIM/PD2}
   -supply {primary IP1/PDtop}
```

then `query_composite_domain dom_combined` returns the composite domain information using the **create_composite_domain** command defined above. `query_composite_domain *` returns all defined composite domains, i.e., `{dom_combined}`. If the `-detailed` option is used, i.e., `query_composite_domain dom_combined -detailed` then the composite domain information information is returned as `{{composite_domain_name dom_combined} {subdomains {IP1/PDtop IP2/SIM/PD2}} {supply {{primary IP1/PDtop}}}}`.

It shall be an error if **-detailed** is specified and ***** is specified for *composite_domain_name*.

Syntax example:

```
query_composite_domain dom_combined
```

7.8 query_design_attributes

Purpose	Query attributes for a design element or model	
Syntax	**query_design_attributes** <**-element** *element_name* | **-model** *model_name*> [**-detailed**]	
Arguments	**-element** *element_name*	A rooted name of a design elements, named processes, sequential `regs`, or signal names.
	-model *model_name*	A model to query.
	-detailed	Returns the design attribute information as a list of {*key value*} pairs, where *key* is the name of the attribute *value* is the value of that attribute. Valid keys are **elements**, **model**, and **attribute**.
Return value	There are two distinct return structures. a) If **-detailed** is not specified then the attribute information shall be return in the form of the corresponding **set_design_attributes** command, or a null string shall be returned if no attribute information is defined for the specified *element* or *model*. b) If **-detailed** is specified then the attribute information for the specified *element* or *model* shall be returned as {*key value*} pairs.	

The **query_design_attributes** command queries attribute information for a specified *element_name* or *model_name*.

For an element that has the following attribute information applied:

```
set_design_attributes -elements lock_cache[0] -attribute UPF_is_leaf TRUE
set_design_attributes -elements lock_cache[0] -attribute UPF_retention
   required
```

then `query_design_attributes -elements lock_cache[0]` shall return the attribute information in the form of the corresponding **set_design_attributes** command. The **-detailed** argument shall return the attribute information as *{key value}* pairs, i.e.,

```
{UPF_is_leaf TRUE} {UPF_retention required}
```

Syntax example:

```
query_design_attributes -elements lock_cache[0] -detailed
```

7.9 query_hdl2upf_vct

Purpose	Query a value conversion table	
Syntax	**query_hdl2upf_vct** *vct_name* [**-detailed**]	
Arguments	*vct_name*	Specifies the *vct_name* to query. If ***** is specified then the name of all defined VCTs shall be returned as a Tcl list.
	-detailed	Returns the VCT information as a list of {*key value*} pairs, where *key* is the name of an argument of the **create_hdl2upf_vct** command (any **-** prefixes are removed) and *value* is the value of that argument. Valid keys are **vct_name**, **hdl_type**, and **table**.
Return value	There are three distinct return structures. a) If ***** is specified for *vct_name* then all previously defined VCTs shall be returned as a Tcl list (a *null string* shall be returned if no VCTs are defined). b) If *vct_name* is specified (and it is not *****) then the VCT information shall be returned in the form of the corresponding **create_hdl2upf_vct** command. c) If **-detailed** is specified then the VCT information for the specified *vct_name* shall be returned as {*key value*} pairs.	

The **query_hdl2upf_vct** command can list and query any previously defined value conversion table.

If a value conversion table specified as

```
create_hdl2upf_vct stdlogic2upf_vss
-hdl_type {vhdl std_logic}
-table {{'U' OFF}
{'X' OFF}
{'0' OFF}
{'1' FULL_ON}
{'Z' PARTIAL_ON}
{'W' OFF}
{'L' OFF}
{'H' FULL_ON}
{'-' OFF}}
```

then `query_hdl2upf_vct stdlogic2upf_vss` returns the VCT information in the formation of the **create_hdl2upf_vct** command defined above. `query_hdl2upf_vct *` returns the defined VCTs, i.e., `{stdlogic2upf_vss}` and `query_hdl2upf_vct stdlogic2upf_vss -detailed` returns the VCT information using {*key value*} pairs, i.e.,

```
{vct_name stdlogic2upf_vss} {hdl_type {vhdl std_logic}} {table {{'U' OFF} {'X'
   OFF} {'0' OFF} {'1' FULL_ON} {'Z' PARTIAL_ON} {'W' OFF} {'L' OFF} {'H'
   FULL_ON} {'-' OFF}}}
```

It shall be an error if **-detailed** is specified and ***** is specified for *vct_name*.

Syntax example:

```
query_hdl2upf_vct stdlogic2upf_vss
```

7.10 query_isolation

Purpose	Query information for an isolation strategy	
Syntax	**query_isolation** *isolation_name* **-domain** *ref_domain_name* [**-detailed**]	
Arguments	*isolation_name*	Specifies the isolation strategy to be queried. If ***** is specified then a list of isolation strategy names defined for *domain_name* shall be returned (or a *null string* if no strategies have been previously defined).
	-domain *ref_domain_name*	Specifies the *ref_domain_name* for which the isolation strategies are to be queried.
	-detailed	Returns the strategy information as a list of {*key value*} pairs. Where *key* is the name of the arguments from the **set_isolation** command (any **-** prefixes are removed) and *value* is the value of that argument. Valid keys are **isolation_name**, **domain**, **elements**, **exclude_elements**, **isolation_power_net**, **isolation_ground_net**, **no_isolation**, **isolation_supply_set**, **isolation_signal**, **name_prefix**, **name_suffix**, **isolation_sense**, **clamp_value**, **sink_off_clamp**, **source_off_clamp**, **location**, **force_isolation**, **diff_supply_only**, and **instance**.
Return value	There are three distinct return structures. a) If a ***** is specified for *isolation_name*, then a list of the defined isolation strategies for the specified *domain_name* shall be returned. b) If a previously defined isolation strategy is specified for *isolation_name* and **-detailed** is not specified then all arguments of the isolation strategy shall be returned in the format of the corresponding **set_isolation command** (see 6.20). c) If **-detailed** is specified then the isolation strategy information shall be returned as a list of {*key value*} pairs.	

The **query_isolation** command can list the previously defined isolation strategies for the specified power domain *domain_name*. All elements returned are referenced to the active scope.

If ***** is specified for *isolation_name*, then a list of the previously defined isolation strategies for the specified *domain_name* shall be returned. If no strategies are defined then a *null string* shall be returned.

If **-detailed** is specified then all the parameters of the specified isolation strategy *isolation_name* shall be returned as a Tcl list consisting of {*key value*} pairs. If *value* is a Boolean, then `0` is returned for *False* and `1` is returned for *True*. For example, if the following isolation strategies have been previously defined

```
set_isolation clamp0_strategy
-domain pda
-isolation_supply_set {ISO1 ISO2} -source_off_clamp 0

set_isolation clamp1_strategy
-domain pda
-isolation_supply_set {ISO1 ISO2} -clamp 1 -applies_to outputs
```

then `query_isolation * -domain pda` returns `{clamp0_strategy clamp1_strategy}`. `query_isolation clamp0_strategy -domain pda` returns the isolation strategy information in the form of the corresponding **set_isolation** command, as defined above. `query_isolation clamp0_strategy -domain pda -detailed` returns


```
{isolation_name clamp0_strategy} {domain pda} {elements {}} {
   isolation_power_net {}} {isolation_ground_net {}} {no_isolation 0}
   {isolation_supply_set {ISO1 ISO2}} { isolation_signal {}} |{clamp_value
   any} {sink_off_clamp {}} {source_off_clamp 0} {location automatic}
   {force_isolation 0}
```

The following arguments of the **set_isolation** command (see 6.40) are not supported by **query_isolation**, as they are expanded on the invocation of the **set_isolation** command.

-applies_to*

-source

-sink

NOTE—If it is not be possible to return all the strategy information in a single return string, i.e., because of layering, the return information shall be returned as a list of lists. The return value of a detailed query of this form shall be composed as {{*detailed_unique_1*} {*detailed_unique_2*} ...}, where each *detailed_unique_** shall be an entire detailed query as shown above.

It shall be an error if

— **-detailed** is specified and *isolation_name* is *.

— the specified *domain_name* starts with .. or /, i.e., the domain needs to be referenced from the active scope, and reside in the active scope or below it.

Syntax example:

```
query_isolation * -domain pda
```

7.11 query_isolation_control

Purpose	Query the control information for an isolation strategy	
Syntax	**query_isolation_control** *isolation_name* **-domain** *domain_name* [**-detailed**]	
Arguments	*isolation_name*	Specifies the isolation strategy to be queried. If * is specified then a list of isolation strategy names defined for *domain_name* shall be returned (or a *null string* if no strategies have been previously defined).
	-domain *domain_name*	Specifies the *domain_name* for which the isolation strategies are to be queried.
	-detailed	Returns the parameters of the isolation strategy as a list of {*key value*} pairs, where *key* is the name of an argument of the **set_isolation_control** command (any **-** prefixes are removed) and *value* is the value of that argument. Valid keys are **isolation_name**, **domain**, **isolation_signal**, **isolation_sense**, and **location**.
Return value	There are three distinct return structures. a) If a * is specified for *isolation_name*, then a list of the defined isolation strategies for the specified *domain_name* shall be returned. b) If a previously defined isolation strategy is specified for *isolation_name* and **-detailed** is not specified then all arguments of the isolation strategy shall be returned in the format of the corresponding **set_isolation_control** command (see 6.41). c) If **-detailed** is specified then the isolation control information for the isolation strategy shall be returned as a list of {*key value*} pairs.	

The **query_isolation_control** command can query the isolation control information for an isolation strategy.

If a control specification is defined as

```
set_isolation_control outputs_only
-domain PD1
-isolation_signal cpu_iso
-isolation_sense low
-location parent
```

then `query_isolation_control * -domain PD1` returns all the defined isolation strategies for domain `PD1`, i.e., `{outputs_only}`. `query_isolation_control outputs_only -domain PD1` returns the isolation control information in the format of the corresponding **set_isolation_control** command, as defined above. `query_isolation_control outputs_only -domain PD1 -detailed` returns the information as a list of **{***key value***}** pairs, i.e.,

```
{isolation_name output_only} {domain PD1} {isolation_signal cpu_iso}
   {isolation_sense low} {location parent}
```

It shall be an error if

— **-detailed** is specified and *isolation_name* is *****.

— The specified *domain_name* starts with `..` or `/`, i.e., the domain needs to be referenced from the active scope, and reside in the active scope or below it.

Syntax example:

```
query_isolation_control output_only -domain pda
```


7.12 query_level_shifter

Purpose	Query information for a level-shifter strategy	
Syntax	**query_level_shifter** *level_shifter_name* **-domain** *domain_name* [**-detailed**]	
Arguments	*level_shifter_name*	Specifies the level-shifter strategy to be queried. If * is specified then a list of level-shifter strategy names defined for *domain_name* shall be returned (or a *null string* if no strategies have been previously defined).
	-domain *domain_name*	Specifies the *domain_name* for which the level-shifter strategies are to be queried.
	-detailed	Returns the parameters of the level-shifter strategy as a list of {*key value*} pairs, where *key* is the name of an argument of the **set_level_shifter** command (any **-** prefixes are removed) and *value* is the value of that argument. Valid keys are **level_shifter_name**, **domain**, **elements**, **no_shift**, **threshold**, **force_shift**, **rule**, **location**, **instance**, **name_prefix**, **name_suffix**, **input_supply_set**, **output_supply_set,** and **internal_supply_set**.
Return value	There are three distinct return structures. a) If a * is specified for *level_shifter_name*, then a list of the defined level-shifter strategies for the specified *domain_name* shall be returned. b) If a previously defined level-shifter strategy is specified for *level_shifter_name* and **-detailed** is not specified then all arguments of the level-shifter strategy shall be returned in the format of the corresponding **set_level_shifter** command (see 6.42). c) If **-detailed** is specified then the level-shifter strategy information for the specified *level_shifter_name* strategy shall be returned as a list of {*key value*} pairs.	

The **query_level_shifter** command can list the previously defined level-shifter strategies and parameters of these strategies. All elements returned are referenced to the active scope.

If a level-shifter strategy is defined as

```
set_level_shifter shift_up
-domain PowerDomainZ
-applies_to outputs
-threshold 0.02
-rule both
```

then `query_level_shifter * -domain PowerDomainZ` returns all the level-shifter strategies defined for the power domain `PowerDomainZ`, i.e., `{shift_up}`. `query_level_shifter shift_up -domain PowerDomainZ` returns the level-shifter strategy information in the format of the corresponding **set_level_shifter** command, as defined above. `query_level_shifter shift_up -domain PowerDomainZ -detailed` returns the level-shifter information as {*key value*} pairs, i.e.,

```
{level_shifter_name shift_up} {domain PowerDomainZ} {elements {}} {no_shift 0}
   {threshold 0.02} {force_shift 0} {rule both} {location automatic}
   {name_prefix {}} {name_suffix {}} {input_supply_set {}} {output_supply_set
   {}} {internal_supply_set {}}
```

The following arguments of the **set_level_shifter** command (see 6.42) are not support by **query_level_shifter**, as they are expanded on the invocation of the **set_level_shifter** command.

-applies_to <**input** | **output** | **both**>

-source

-sink

NOTE—If it is not be possible to return all the strategy information in a single return string, i.e., because of layering, the return information shall be returned as a list of lists. The return value of a detailed query of this form shall be composed as {{*detailed_unique_1*} {*detailed_unique_2*} ...}, where each *detailed_unique_** shall be an entire detailed query as shown above.

It shall be an error if

— **-detailed** is specified and *level_shifter_name* is *.

— The specified *domain_name* starts with .. or /, i.e., the domain needs to be referenced from the active scope, and reside in the active scope or below it.

Syntax example:

```
query_level_shifter * -domain pda
```

7.13 query_map_isolation_cell

Purpose	Query the mapping for an isolation strategy	
Syntax	**query_map_isolation_cell** *isolation_name* **-domain** *domain_name* [**-detailed**]	
Arguments	*isolation_name*	Specifies the isolation strategy name *isolation_name* to query. If * is specified then the name of all isolation strategies shall be returned as a Tcl list.
	-domain *domain_name*	The *domain_name* for which the strategies are to be queried.
	-detailed	Returns the strategy information as a list of {*key value*} pairs, where *key* is the name of an argument of the **map_isolation_cell** command (any - prefixes are removed) and *value* is the value of that argument. Valid keys are **isolation_name**, **domain**, **elements**, **lib_cell**, **lib_cell_type**, **lib_model_name**, and **port**.
Return value	There are three distinct return structures. a) If * is specified for *isolation_name* then all previously defined isolation strategies shall be returned as a Tcl list (a *null string* shall be returned if no isolation strategies are defined). b) If *isolation_name* is specified (and it is not *) then the strategy information shall be returned in the form of the corresponding **map_isolatation_cell** command. c) If **-detailed** is specified then the isolation information for the specified *isolation_name* shall be returned as {*key value*} pairs.	

The **query_map_isolation_cell** command queries the mapping specification for an isolation strategy *isolation_name*.

If an isolation strategy has a mapping specification defined as

```
map_isolation_cell test_PD1 -domain PD1 -lib_cell_type jims_iso_fast
```


then `query_map_isolation_cell * -domain PD1` returns all defined isolation strategies for power domain `PD1`, i.e., `{test_PD1}`. `query_map_isolation_cell test_PD1 -domain PD1` returns the isolation mapping information in the format of the corresponding **map_isolation_cell** command as defined above. `query_map_isolation_cell test_PD1 -domain PD1 -detailed` returns the mapping specification as a list of {*key value*} pairs, i.e,

```
{isolation_name test_PD1} {domain PD1} {elements {}} {lib_cells {}}
   {lib_cell_type jims_iso_fast} {lib_model_name {}} {port {}}
```

NOTE—If multiple mapping specification are defined for different design elements of an isolation strategy, then multiple **map_isolation_cell** commands shall be returned if **-detailed** is not specified and if **-detailed** is specified, then a list of list of {*key value*} pairs shall be returned, e.g., `{{mapping_specification_1} {mapping_specification_2}}`.

It shall be an error if

— *isolation_name* is ***** and **-detailed** is specified.

— *isolation_name* is not a valid isolation strategy.

Syntax example:

```
query_map_isolation_cell * -domain PD1
```

7.14 query_map_level_shifter_cell

Purpose	Query the mapping for a level_shifter strategy	
Syntax	**query_map_level_shifter_cell** *level_shifter_strategy* **-domain** *domain_name* [**-detailed**]	
Arguments	*level_shifter_strategy*	Specifies the level-shifter strategy name *level_shifter_strategy* to query. If * is specified then the name of all level-shifter strategies shall be returned as a Tcl list.
	-domain *domain_name*	The *domain_name* for which the strategies are to be queried.
	-detailed	Returns the strategy information as a list of {*key value*} pairs, where *key* is the name of an argument of the **map_level_shifter_cell** command (any **-** prefixes are removed) and *value* is the value of that argument. Valid keys are **level_shifter_strategy**, **domain**, **elements**, **model**, **map**, and **applies_to**.
Return value	There are three distinct return structures. a) If * is specified for *level_shifter_strategy* then all previously defined level-shifter strategy names shall be returned as a Tcl list (a *null string* shall be returned if no level-shifter strategies are defined). b) If *level_shifter_strategy* is specified (and it is not *) then the strategy information shall be returned in the form of the corresponding **map_level_shifter_cell** command. c) If **-detailed** is specified then the level-shifter strategy information for the specified *level_shifter_strategy* shall be returned as {*key value*} pairs.	

The **query_map_level_shifter_cell** command can query the mapping specification for a level-shifter strategy.

If a level-shifter strategy has a mapping specification defined as

```
map_level_shifter_cell shift_up -domain PD1 -applies_to verification
```

then `query_map_level_shifter_cell * -domain PD1` returns all defined level-shifter strategies for power domain `PD1`, i.e., `{shift_up}`. `query_map_level_shifter_cell shift_up -domain PD1` returns the level-shifter mapping information in the format of the corresponding **map_level_shifter_cell** command, as defined above. `query_map_level_shifter_cell shift_up -domain PD1 -detailed` returns the mapping specification as a list of {*key value*} pairs, i.e.,

```
{level_shifter_strategy shift_up} {domain PD1} {elements {}} {model {}} {map
    {}} {applies_to verification}
```

NOTE—If multiple mapping specification are defined for different design elements of a level-shifter strategy, then multiple **map_level_shifter_cell** commands shall be returned if **-detailed** is not specified and if **-detailed** is specified, then a list of list of {*key value*} pairs shall be returned, e.g., `{{mapping_specification_1} {mapping_specification_2}}`.

It shall be an error if

— *level_shifter_strategy* is * and **-detailed** is specified.

— *level_shifter_strategy* is not a valid level-shifter strategy.

Syntax example:

```
query_map_level_shifter_cell * -domain PD1
```

7.15 query_map_power_switch

Purpose	Query the mapping for a switch cell	
Syntax	**query_map_power_switch** *switch_name* [**-detailed**]	
Arguments	*switch_name*	Specifies the switch *switch_name* to query. If * is specified then the name of all switches shall be returned as a Tcl list.
	-detailed	Returns the switch information as a list of {*key value*} pairs, where *key* is the name of an argument of the **map_power_switch** command (any - prefixes are removed) and *value* is the value of that argument. Valid keys are **switch_name**, **lib_cells**, and **port_map**.
Return value	There are three distinct return structures. a) If * is specified for *switch_name* then all previously defined switches shall be returned as a Tcl list (a *null string* shall be returned if no switches are defined). b) If *switch_name* is specified (and it is not *) then the switch information shall be returned in the form of the corresponding **map_power_switch** command. c) If **-detailed** is specified then the switch information for the specified *switch_name* shall be returned as {*key value*} pairs.	

The **query_map_power_switch** command can query the mapping specification for a switch cell.

If a switch cell has a mapping specification defined as

```
map_power_switch switch_sw1 -lib_cells test_model -port_map {{test_port
    control_port_test}}
```


then `query_map_power_switch *` returns all defined switches, i.e., `{switch_sw1}`. `query_map_power_switch switch_sw1` returns the switch mapping information in the format of the corresponding **map_power_switch** command, as defined above. `query_map_power_switch switch_sw1 -detailed` returns the mapping specification as a list of {*key value*} pairs, i.e.,

```
{switch_name switch_sw1} {lib_cells {test_model}} {port_map {}}
```

It shall be an error if

— *switch_name* is * and **-detailed** is specified.

— *switch_name* is not a valid switch.

Syntax example:

```
query_map_power_switch switch_sw1 -detailed
```

7.16 query_map_retention_cell

Purpose	Query the mapping for a retention strategy	
Syntax	**query_map_retention_cell** *retention_name_list* **-domain** *domain_name* [**-detailed**]	
Arguments	*retention_name_list*	Specifies the retention strategy name *retention_name_list* to query. If * is specified then the name of all retention strategies shall be returned as a Tcl list.
	-domain *domain_name*	The *domain_name* for which the strategies are to be queried.
	-detailed	Returns the strategy information as a list of {*key value*} pairs, where *key* is the name of the arguments from the **map_retention_cell** command (any **-** prefixes are removed) and *value* is the value of that argument. Valid keys are **retention_strategy**, **domain**, **lib_cells**, **lib_cell_type**, **lib_model_name**, **port**, and **elements**.
Return value	There are three distinct return structures. a) If * is specified for *retention_name_list* then all previously defined retention strategy names shall be returned as a Tcl list (a *null string* shall be returned if no retention strategies are defined). b) If *retention_name_list* is specified (and it is not *) then the strategy information shall be returned in the form of the corresponding **map_retention_cell** command. c) If **-detailed** is specified then the retention strategy information for the specified *retention_name_list* shall be returned as {*key value*} pairs.	

The **query_map_retention_cell** can query the mapping specification for a retention strategy.

Give a retention mapping specification defined as

```
map_retention_cell test_PdA -domain {PdA} -elements {foo/U1 foo/U2}
```

then `query_map_retention_cell * -domain PdA` returns all the retention strategies defined on `PdA`, i.e.,. `{test_PdA}`. `query_map_retention_cell test_PdA -domain PdA` returns the retention mapping information in the format of the corresponding **map_retention_cell** command, as defined above. `query_map_retention_cell test_PdA -domain PdA -detailed` returns the mapping information as {*key value*} pairs, i.e.,

```
{retention_strategy test_PdA} {domain PdA} {elements {foo/U1 foo/U2} {model
    {}} {map {}}
```

NOTE—If multiple mapping specification are defined for different design elements of a retention strategy, then multiple **map_retention_cell** commands shall be returned if **-detailed** is not specified and if **-detailed** is specified, then a list of list of {*key value*} pairs shall be returned, e.g., {{mapping_specification_1} {mapping_specification_2}}.

It shall be an error if

— *retention_strategy* is * and **-detailed** is specified.

— *retention_strategy* is not a valid level-shifter strategy.

Syntax example:

```
query_map_retention_cell * -domain PD1
```

7.17 query_name_format

Purpose	Query information on the name formatting rules	
Syntax	**query_name_format** [**-isolation_prefix** | **-isolation_suffix** | **-level_shift_prefix** | **-level_shift_suffix** | **-implicit_supply_prefix** | **-implicit_supply_suffix** | **-implicit_logic_prefix** | **-implicit_logic_suffix** | **-detailed**]	
Arguments	**-isolation_prefix**	Returns the isolation instance and net prefix.
	-isolation_suffix	Returns the isolation instance and net suffix.
	-level_shift_prefix	Returns the level-shifter instance and net prefix.
	-level_shift_suffix	Returns the level-shifter instance and net suffix.
	-implicit_supply_prefix	Returns the implicitly created supply net and port prefix.
	-implicit_supply_suffix	Returns the implicitly created supply net and port suffix.
	-implicit_logic_prefix	Returns the implicitly created logic net and port prefix.
	-implicit_logic_suffix	Returns the implicitly created logic net and port suffix.
	-detailed	Returns the parameters of the name format as a list of {*key value*} pairs, where *key* is the name of an argument of the query (any **-** prefixes are removed) and *value* is the value of that argument.
Return value	Return the queried parameter if an optional argument is specified or all of the parameters of the **name_format** command (see 6.35) if none of the optional arguments are specified.	

The **query_name_format** command lists the current name format rules in effect.

-detailed returns all the name format parameters as a Tcl list consisting of {*key value*} pairs. For example, if **-isolation_prefix** is set to `ISO_` and **-level_shift_prefix** is set to `LS_`, the **-detailed** option returns the following:


```
{{isolation_prefix ISO_} {isolation_suffix _UPF_ISO} {level_shift_prefix LS_}
   {level_shift_suffix _UPF_LS} {implicit_supply_prefix ""}
   {implicit_supply_suffix _UPF_IS} {implicit_logic_prefix ""}
   {implicit_logic_suffix _UPF_IL}}
```

Syntax example:

```
query_name_format \
-isolation_suffix
```

7.18 query_net_ports

Purpose	Return ports logically connected to a net	
Syntax	**query_net_ports** *net_name* [**-transitive** <**TRUE** | **FALSE**>] [**-leaf**]	
Arguments	*net_name*	The net for which the connected ports are to be listed. Any port connected to *net_name* in the active scope is returned.
	-transitive <**TRUE** | **FALSE**>	When **-transitive** is **TRUE**, the command applies to the descendants of the elements; the default is **FALSE**.
	-leaf	Only returns leaf cell ports connected to *net_name*. By default, both non-leaf and leaf cell ports shall be returned.
Return value	Return a list of ports that are logically connected to the specified net. If no ports are connected, a *null string* is returned.	

The **query_net_ports** command lists all ports that are logically connected to a specified net.

-transitive returns all ports that are connected hierarchically to this net (and that are visible within the active scope); otherwise, only ports connected at the scope of *net_name* are returned.

The following conditions also apply:

— The specified *net_name* cannot start with `..` or `/`, i.e., the net needs to be referenced from the active scope, and reside in the active scope or below it.

— All ports returned are referenced to the active scope.

Syntax example:

```
query_net_ports top/a/b/c -transitive
```

7.19 query_partial_on_translation

Purpose	Return the translation of **PARTIAL_ON** for named tools
Syntax	**query_partial_on_translation**
Return value	Return the current setting of the translation as a list of {*key value*} pairs, where *key* is the name of an argument of the **set_partial_on_translation** command (any - prefixes are removed) and *value* is the value of that argument. Valid keys are **partial_on_translation**, **full_on_tools**, and **off_tools**.

The **query_partial_on_translation** command provides the ability to determine the translation of **PARTIAL_ON** to **FULL_ON** or **OFF**.

The query returns the translation settings as {*key value*} pairs. If the translation settings are specified as

```
set_partial_on_translation OFF -full_on_tools {power_analysis_tool_name}
   -off_tools {steves_simulator}
```

then **query_partial_on_translation** shall return the settings as {*key value*} pairs of the form

```
{partial_on_translation OFF} {full_on_tools {power_analysis_tool_name}}
   {off_tools {steves_simulator}}
```

Syntax example:

```
query_partial_on_translation
```

7.20 query_pin_related_supply

Purpose	Query the related power and ground pairs for a library cell	
Syntax	**query_pin_related_supply** *library_cell* [**-detailed**]	
Arguments	*library_cell*	Specifies the library cell to query.
	-detailed	Returns the parameters of the level-shifter strategy as a list of {*key value*} pairs, where *key* is the name of an argument of the **set_pin_related_supply** command (any **-** prefixes are removed) and *value* is the value of that argument. Valid keys are **library_cell**, **pins**, **related_power_pin**, and **related_ground_pin**.
Return value	There are two distinct return structures. a) If **-detailed** is not specified then the related supply information for *library_cell* shall be returned in the form of the corresponding **set_pin_related_supply** command or a *null string* shall be returned if no related supply information has been defined. b) If **-detailed** is specified then the related supply information for the specified *library_cell* shall be returned as a list of {*key value*} pairs.	

The **query_pin_related_supply** command queries the related power and ground pins for signal pins on a library cell.

If a library cell has the following related supply settings:

```
set_pin_related_supply library1/cell1 -pins {A} -related_power_pin VDDI
   -related_ground_pin VSS
set_pin_related_supply library1/cell1 -pins {Y} -related_power_pin VDDO
   -related_ground_pin VSS
```

then `query_pin_related_supply library1/cell1` returns the related supply information in the form of the corresponding **set_pin_related_supply** command, as defined above (two **set_pin_related_supply** commands are returned in this example). `query_pin_related_supply library1/cell1 -detailed` returns the related supply information as **{***key value***}** pairs, i.e.,

```
{{library_cell library1/cell1} {pins {A}} {related_power_pin VDDI}
   {related_ground_pin VSS}} {library_cell library1/cell1} {pins {Y}}
   {related_power_pin VDDO} {related_ground_pin VSS}}}
```

NOTE—Because a library cell can have *n* signal pins with different related power and ground pins, the **query_pin_related_supply** command can result in multiple **set_pin_related_supply** commands being returned if **-detailed** is not specified, and if **-detailed** is specified then a list of list of **{***key value***}** pairs shall be returned, i.e., `{{signal_pin_to_pg_pin_group_1} {signal_pin_to_pg_pin_group_2}{...}...}`.

Syntax example:

```
query_pin_related_supply library1/cell1
```

7.21 query_port_attributes

Purpose	Query the port attributes for a specified port	
Syntax	**query_port_attributes** *port* [**-detailed**]	
Arguments	*port*	Specifies the port to query.
	-detailed	Returns the parameters of the level-shifter strategy as a list of **{***key value***}** pairs, where *key* is the name of an argument of the **set_port_attributes** command (any **-** prefixes are removed) and *value* is the value of that argument. Valid keys **ports**, **exclude_ports**, **domains**, **exclude_domains**, **elements**, **exclude_elements**, **model**, **attribute**, **sink_off_clamp**, **source_off_clamp**, **receiver_supply**, **driver_supply**, **related_power_port**, **related_ground_port**, **related_bias_ports**, **repeater_supply**, and **pg_type**.
Return value	There are two distinct return structures. a) If **-detailed** is not specified then all attributes of the port port_name shall be returned in the format of the corresponding **set_port_attributes** command (see 6.45). b) If **-detailed** is specified then the port attribute information for the specified shall be returned as a list of **{***key value***}** pairs.	

The **query_port_attributes** command queries the port attribute information for a specified *port*.

If a port has the following attribute specification

```
set_port_attributes -ports {A B} -sink_off_clamp 0
```

then `query_port_attributes A` returns the attribute information in the form of the corresponding **set_port_attributes** command, as defined above. `query_port_attributes {A} -detailed` returns the attribute information as **{***key value***}** pairs, i.e.,

```
{port A} {model {}} {sink_off_clamp 0} {source_off_clamp {}} {receiver_supply
   {}} {driver_supply {}}
```

Syntax example:

```
query_port_attributes B
```

7.22 query_port_direction

Purpose	Return the direction of the specified port	
Syntax	**query_port_direction** *port*	
Arguments	*port*	The name of the port for which the direction is being queried.
Return value	Return **in**, **out**, or **inout**.	

The **query_port_direction** command returns the direction of the specified port. The port can be a signal port or a supply port.

The specified *port* cannot start with `..` or `/`, i.e., the port needs to be referenced from the active scope, and reside in the active scope or below it.

Syntax example:

```
query_port_direction {top/a/b}
```

7.23 query_port_net

Purpose	Return the net logically connected to a port	
Syntax	**query_port_net** *port_name* **-conn** <**low** | **high**>	
Arguments	*port_name*	The port where this net is to be returned.
	-conn <**low** | **high**>	Returns the `LowConn` or `HighConn` (the default) connection. This option can only be specified if *port_name* is not on a leaf cell.
Return value	Return the name of the net connected to the specified *port_name*. If no net is connected, a *null string* is returned.	

The **query_port_net** command returns the net connected to a specified port (if such a connection exists). A hierarchal port can have both a `LowConn` and `HighConn`, so the **-conn** option can be used to specify the net name to return. If no net is connected to the specified port, a *null string* is returned.

The following conditions also apply:

— The specified *port_name* cannot start with `..` or `/`, i.e., the port needs to be referenced from the active scope, and reside in the active scope or below it.

— The returned net is referenced to the active scope.

Syntax example:

```
query_port_net top/a/b -conn low
```

7.24 query_port_state

Purpose	Return the state information for a specified port	
Syntax	**query_port_state** *port_name* **-state** *state_name* [**-detailed**]	
Arguments	*port_name*	Simple or hierarchical name of a supply port for which the power state information is to be queried.
	-state *state_name*	The *state_name* being queried. If ***** is specified then state information for all states defined for *port_name* shall be returned.
	-detailed	Returns the port state information as a list of {*key value*} pairs. Where valid keys are **port_name**, **state_name**, and **state**.
Return value	There are three distinct return structures. a) If **-state** is not specified then a list of all defined states for the *port_name* shall be returned as a Tcl list. A *null string* shall be returned if no states are defined. b) If a *state_name* is specified then the state information for the specified state shall be returned, using the corresponding **add_port_state** command. If ***** is specified then state information for all states shall be returned. c) If **-detailed** is specified then the state information shall be returned as {*key value*} pairs.	

The **query_port_state** command lists the previously defined states for *port_name*. If *state_name* is not specified then a list of defined states for the port shall be returned. If *state_name* is defined then all parameters of the specified state shall be returned.

-detailed returns all the parameters of *state_name* as a Tcl list consisting of {*key value*} pairs. For example, if a state called `active_state` is defined on the port `VN1` with the state information `{0.88 0.90 0.92)` then **-detailed** option returns the following:

```
{port_name VN1} {state_name active_state} {state {0.88 0.90 0.92}}
```

Without the **-detailed** option, the format of the returned parameters shall be in the format of the corresponding **add_port_state** command, i.e.,

```
add_port_state VN1 -state {active_state {0.88 0.90 0.92}}
```

It shall be an error if **-detailed** is specified and ***** is specified for *state_name*.

Syntax example:

```
query_port_state VN1
```

7.25 query_power_domain

Purpose	Query a power domain	
Syntax	**query_power_domain** *domain_name* [**-non_leaf** | **-all** | **-no_elements**] [**-detailed**]	
Arguments	*domain_name*	Specifies the power domain to query. If ***** is specified then the name of all defined power domains shall be returned as a Tcl list.
	-non_leaf | **-all** | **-no_elements**	Allows filtering or exclusion for any elements from being returned by the query. The default is to return the non-leaf cells attached to the domain only.
	-detailed	Returns the domain information as a list of {*key value*} pairs, where *key* is the name of an argument of the **create_power_domain** command (any **-** prefixes are removed) and *value* is the value of that argument. Valid keys are **domain_name**, **simulation_only**, **elements**, **include_scope**, **supply**, **scope**, and **define_func_type**.
Return value	There are three distinct return structures. a) If ***** is specified for *domain_name* then all previously defined domains shall be returned as a Tcl list (a *null string* shall be returned if no domains are defined). b) If *domain_name* is specified (and it is not *****) then the domain information shall be returned in the format of the corresponding **create_power_domain** command. c) If **-detailed** is specified then the domain information for the specified *domain_name* shall be returned as {*key value*} pairs.	

The **query_power_domain** command queries the parameters of a power domain. The **-no_elements** option prevents the elements attached to the domain from being returned by the query command.

If a power domain is created as follows:

```
create_power_domain PD1 -elements {top/U1}
-supply {primary PD1_Primary}
-supply {isolation PD1_ret}
-supply {retention PD1_ret}
-supply {mem_array PD1_ma}
```

then `query_power_domain *` returns any power domains defined in and below the active scope, i.e., `{PD1}`. `query_power_domain PD1` returns the power domain information in the format of the corresponding **create_power_domain** command, as defined above. `query_power_domain PD1 -detailed` returns the power domain information as {*key value*} pairs, i.e.,

```
{domain_name PD1} {elements {top/U1}} {supply {{primary PD1_Primary}
   {isolation PD1_ret} {retention PD1_ret} {mem_array PD1_ma}}}
```

It shall be an error if **-detailed** is specified and ***** is specified for *domain_name*.

Syntax example:

```
query_power_domain PD1 -no_elements -detailed
```


7.26 query_power_domain_element

Purpose	Return the domain membership information for a design element	
Syntax	**query_power_domain_element** *design_element*	
Arguments	*design_element*	The design_element for which the domain membership information is to be returned.
Return value	Return the domain for the specified *design_element*. If no domain is found, a *null string* is returned.	

The **query_power_domain_element** returns the domain membership of the specified object.

The following conditions also apply:

— The specified *design_element* cannot start with `..` or `/`, i.e., the object needs to be referenced from the active scope, and reside in the active scope or below it.

— The returned domain is referenced to the active scope.

Syntax example:

```
query_power_domain_element {net@top/a/b}
```

NOTE—Nets are propagated as necessary through the descendant tree and may be renamed to avoid name collision; therefore, the same simple name in different scopes may refer to nets that are independent and unconnected.

7.27 query_power_state

Purpose	Return the state information for a power domain or supply set	
Syntax	**query_power_state** *object_name* **-state** *state_name* [**-detailed**]	
Arguments	*object_name*	Simple name of a power domain or supply set.
	-state *state_name*	*state_name* is the simple name of the state being queried. If ***** is specified then state information for all states are returned.
	-detailed	Returns the power state information as list of {*key value*} pairs, where *key* is the name of an argument of the **add_power_state** command (any **-** prefixes are removed) and *value* is the value of that argument. Valid keys are **object_name**, **state_name**, **supply_expr**, **logic_expr**, **simstate**, **legal**, and **illegal**.
Return value	There are three distinct return structures. a) If **-state** is not specified a list of defined power states for *object_name* shall be returned as a Tcl list (a *null string* shall be returned if no power states are defined). b) If *state_name* is specified then the state information for this state shall be returned in the format of the corresponding **add_power_state** command. c) If **-detailed** is specified then the power state information shall be returned as {*key value*} pairs.	

The **query_power_state** command lists the previously defined power states for the specified *object_name,* which can be a power domain or a supply set. If *state_name* is not specified then a list of defined states for

object_name shall be returned. If a *state_name* is defined then all parameters of the specified state shall be returned.

-detailed returns all the parameters of the specified power state *state_name* as a Tcl list consisting of **{***key value***}** pairs. For example, if a legal state called `LPS` on the supply set `PDA_SUPPLY` has the **-supply_expr** condition `{power == '{FULL_ON, 0.8}}` and the **-logic_expr** `condition {u1/PdA == GO_MODE}` then the **-detailed** option returns the following.

```
{state_name LPS} {object_name PDA_SUPPLY} {supply_expr {power == '{FULL_ON,
   0.8}}} {logic_expr {u1/PdA == GO_MODE}} {legal 1} {illegal 0} {simstate {}}
```

Without the **-detailed** option, the format of the returned parameters shall be in the format of the corresponding **add_power_state** command, i.e.,

```
add_power_state PDA_RET
   -state LPS
   -supply_expr {power == '{FULL_ON, 0.8}}
   -logic_expr {u1/PdA == GO_MODE}
   -legal
```

It shall be an error if **-detailed** is specified and ***** is specified for *state_name*.

Syntax example:

```
query_power_state PDA_RET -detailed
```

7.28 query_power_switch

Purpose	Query the information for a UPF power switch	
Syntax	**query_power_switch** *switch_name* [**-detailed**]	
Arguments	*switch_name*	Specifies the power switch to query. If ***** is specified then the name of all power switches shall be returned as a Tcl list.
	-detailed	Returns the power switch information as a list of **{***key value***}** pairs, where *key* is the name of an argument of the **create_power_switch** command (any **-** prefixes are removed) and *value* is the value of that argument. Valid keys are **switch_name**, **domain**, **output_supply_port**, **input_supply_port**, **control_port**, **on_state**, **off_state**, **supply_set**, **on_partial_state**, **ack_port**, **ack_delay**, and **error_state**.
Return value	There are three distinct return structures. a) If ***** is specified for *switch_name* then all previously defined switches shall be returned as a Tcl list (a *null string* shall be returned if no domains are defined). b) If *switch_name* is specified (and it is not *****) then the switch information shall be returned. c) If **-detailed** is specified then the switch information for the specified *switch_name* shall be returned as **{***key value***}** pairs.	

The **query_power_switch** command queries the parameters for a UPF defined power switch.

If a power switch is defined as

```
create_power_switch sw1
-output_supply_port {vout VN3}
```



```
-input_supply_port {vin1 VN1}
-input_supply_port {vin2 VN2}
-control_port {ctrl_small ON1}
-control_port {ctrl_large ON2}
-control_port {ss SUPPLY_SELECT}
-on_partial_state {partial_s1 vin1 {ctrl_small & !ctrl_large & ss}}
-on_state {full_s1 vin1 {ctrl_small & ctrl_large & ss}}
-on_partial_state {partial_s2 vin2 {ctrl_small & !ctrl_large & !ss}}
-on_state {full_s2 vin2 {ctrl_small & ctrl_large & !ss}}
-off_state {not_required {~ctrl_small | ctrl_large | ss}}
-error_state {no_small {!ctrl_small & ctrl_large}}
```

then `query_power_switch *` returns the name of any switches defined in and below the active scope. `query_power_switch sw1` returns the switch information in the format of the corresponding **create_power_switch** command, as defined above. `query_power_switch sw1 -detailed` returns the switch information as a list of {*key value*} pairs, i.e.,

```
{switch_name sw1} {domain {}} {output_supply_port {vout VN3}}
   {input_supply_port {{vin1 VN1} {vin2 VN2}}} {control_port {{ctrl_small ON1}
   {ctrl_large ON2} {ss SUPPLY_SELECT}}} {on_state {{full_s1 vin1 {ctrl_small
   & ctrl_large & ss}} {full_s2 vin2 {ctrl_small & ctrl_large & !ss}}}}
   {off_state {not_required {~ctrl_small | ctrl_large | ss}} {supply_set {}}
   {on_partial_state {{partial_s1 vin1 {ctrl_small & !ctrl_large & ss}}
   {partial_s2 vin2 {ctrl_small & !ctrl_large & !ss}}}} {ack_port {}}
   {ack_delay {}} {error_state {{no_small {!ctrl_small & ctrl_large}}}}
```

It shall be an error if **-detailed** is specified and * is specified for *switch_name*.

Syntax example:

```
query_power_switch *
```

7.29 query_pst

Purpose	Query a power state table	
Syntax	**query_pst** *table_name* [**-detailed**]	
Arguments	*table_name*	Specifies the pst table name to query. If * is specified then the name of all PSTs shall be returned as a Tcl list.
	-detailed	Returns the pst information as a list of {*key value*} pairs, where *key* is the name of an argument of the **create_pst** command (any **-** prefixes are removed) and *value* is the value of that argument. Valid keys are **table_name** and **supplies**.
Return value	There are three distinct return structures. a) If * is specified for *table_name* then all previously defined PSTs shall be returned as a Tcl list (a *null string* shall be returned if no PSTs are defined). b) If *table_name* is specified (and it is not *) then the table information shall be returned. c) If **-detailed** is specified then the table information for the specified *table_name* shall be returned as {*key value*} pairs.	

The **query_pst** command queries the information for any defined PSTs.

If a power state table is defined as

```
create_pst MyPowerStateTable -supplies {PN1 PN2 SOC/OTC/PN3}
```

then `query_pst *` returns any defined PSTs, i.e., `MyPowerStateTable`. `query_pst MyPowerStateTable` returns the PST information in the form of the **create_pst** command, as defined above. `query_pst MyPowerStateTable -detailed` returns the PST information as a list of {*key value*} pairs, i.e.,

```
{table_name MyPowerStateTable} {supplies {PN1 PN2 SOC/OTC/PN3}}
```

It shall be an error if **-detailed** is specified and ***** is specified for *table_name*.

Syntax example:

```
query_pst *
```

7.30 query_pst_state

Purpose	Return the state information for a power domain or supply set	
Syntax	**query_pst_state** *state_name* **-pst** *table_name* [**-detailed**]	
Arguments	*state_name*	Specifies the name of the state or ***** for all states.
	-pst *table_name*	The power state table for which the state information is to be queried.
	-detailed	Returns the power state information as list of {*key value*} pairs, where *key* is the name of an argument of the **add_pst_state** command (any **-** prefixes are removed) and *value* is the value of that argument. Valid keys are **state_name**, **pst**, and **state**.
Return value	There are three distinct return structures. a) If ***** is specified for *state_name* then all states defined for the specified power state table *table_name* shall be returned as a Tcl list (a *null string* shall be returned if no power states are defined). b) If *state_name* is specified then the state information for the specified state shall be returned in the format of the corresponding **add_pst_state** command. c) If **-detailed** is specified the power state information shall be returned as {*key value*} pairs.	

The **query_pst_state** command lists the previously defined power states for *table_name*. If ***** is specified for the *state_name* then a list of defined state names shall be returned as a Tcl list. If a *state_name* is defined (and is not *) then all parameters of the specified state shall be returned.

-detailed returns all the parameters of the specified power state *state_name* as a Tcl list consisting of {*key value*} pairs. If a power state table is defined as:

```
create_pst pt -supplies { PN1 PN2 SOC/OTC/PN3 }
add_pst_state s1 -pst pt -state { s08 s08 s08 }
add_pst_state s2 -pst pt -state { s08 s08 off }
add_pst_state s3 -pst pt -state { s08 s09 off }
```


then `query_pst_state * -pst pt` returns all the specified states, i.e., `{s1 s2 s3}`. If the `-detailed` option is used, i.e., `query_pst_state s1 -pst pt -detailed`, then the state information shall be returned as `{pst pt} {state_name s1} {state {s08 s08 s08}`.

NOTE—Without the **-detailed** option, the format of the returned parameters shall be in the format of the corresponding **add_pst_state** command.

It shall be an error if **-detailed** is specified and ***** is specified for *state_name*.

Syntax example:

```
query_pst_state s1 -pst pt -detailed
```

7.31 query_retention

Purpose	Query the retention strategy information for a domain	
Syntax	**query_retention** *retention_name* **-domain** *domain_name* [**-detailed**]	
Arguments	*retention_name*	Specified the retention strategy to be queried. If ***** is specified then a list of retention strategy names defined for *domain_name* shall be returned (or a *null string* if no strategies have been previously defined).
	-domain *domain_name*	Specifies the *domain_name* for which the retention strategies are to be queried.
	-detailed	Returns the parameters of the retention strategy as a list of **{***key value***}** pairs, where *key* is the name of an argument of the **set_retention** command (any **-** prefixes are removed) and *value* is the value of that argument. Valid keys are **retention_name**, **domain**, **elements**, **exclude_elements**, **retention_power_net**, **retention_ground_net**, **retention_supply_set**, **no_isolation**, **save_signal**, **restore_signal**, **save_condition**, **restore_condition**, **retention_condition**, **use_retention_as_primary**, **parameters**, and **instance**.
Return value	There are three distinct return structures. a) If a ***** is specified for *retention_name*, then a list of the defined retention strategies for the specified *domain_name* shall be returned. b) If a previously defined retention strategy is specified for *retention_name* and **-detailed** is not specified then all arguments of the retention strategy shall be returned in the format of the corresponding **set_retention** command (see 6.47). c) If **-detailed** is specified then the retention strategy information for the specified *retention_name* strategy shall be returned as a list of **{***key value***}** pairs.	

The **query_retention** command lists the previously defined retention strategies for the specified power domain *domain_name*. All elements returned are referenced to the active scope.

If ***** is specified for *retention_name*, then a list of the previously defined retention strategies for the specified *domain_name* shall be returned. If no strategies are defined, then a *null string* shall be returned.

If **-detailed** is specified then all the parameters of the specified retention strategy *retention_name* shall be returned as a Tcl list consisting of **{***key value***}** pairs.

If a retention strategy is defined as

```
set_retention my_retention_strategy -domain pda
-retention_supply_set PDA_ret_supply
-save_signal {my_save posedge} -restore_signal {my_restore negedge }
```

then `query_retention * -domain pda` returns `{my_retention_strategy}`. `query_retention my_retention_strategy -domain pda` returns the retention strategy information in the form of the corresponding **set_retention** command, as defined above. `query_retention my_retention_strategy -domain pda -detailed` returns the retention strategy information as a list of {*key value*} pairs, i.e.,

```
{retention_name my_retention_strategy} {domain pda} {elements {}}
   {exclude_elements {}} {retention_power_net {}} {retention_ground_net {}}
   {retention_supply_set PDA_ret_supply} {save_signal {my_save posedge}}
   {restore_signal {my_restore negedge}} {save_condition {}}
   {restore_condition {}} {output_related_supply_set {}}
```

NOTE—If it is not be possible to return all the strategy information in a single return string, i.e., because of layering, the return information shall be returned as a list of lists. The return value of a detailed query of this form shall be composed as {{*detailed_unique_1*} {*detailed_unique_2*} ...}, where each *detailed_unique_** shall be an entire detailed query as shown above.

It shall be an error if

- **-detailed** is specified and *retention_name* is *****.
- The specified *domain_name* starts with `..` or `/`, i.e., the domain needs to be referenced from the active scope, and reside in the active scope or below it.

Syntax example:

```
query_retention * -domain pda
```


7.32 query_retention_control

Purpose	Query the retention strategy control information for a domain	
Syntax	**query_retention_control** *retention_name* **-domain** *domain_name* [**-detailed**]	
Arguments	*retention_name*	Specified the retention strategy control information to be queried. If ***** is specified then a list of retention strategy names defined for *domain_name* shall be returned (or a *null string* if no strategies have been previously defined).
	-domain *domain_name*	Specifies the *domain_name* for which the retention strategies are to be queried.
	-detailed	Returns the control parameters of the retention strategy as a list of {*key value*} pairs, where *key* is the name of an argument of the **set_retention_control** command (any **-** prefixes are removed) and *value* is the value of that argument. Valid keys are **retention_name**, **domain**, **save_signal**, **restore_signal**, **assert_r_mutex**, **assert_s_mutex**, and **assert_rs_mutex**.
Return value	There are three distinct return structures. a) If a ***** is specified for *retention_name*, then a list of the defined retention strategies for the specified *domain_name* shall be returned. b) If a previously defined retention strategy is specified for *retention_name* and **-detailed** is not specified then all the control arguments of the retention strategy shall be returned in the format of the corresponding **set_retention_control** command (see 6.48). c) If **-detailed** is specified then the retention strategy information for the specified *retention_name* strategy shall be returned as a list of {*key value*} pairs.	

The **query_retention_control** command queries the retention control information for a retention strategy.

If a control specification is defined as

```
set_retention_control my_retention_strategy
-domain PDA
-save_signal {power_controller_inst/save_1 high}
-restore_signal {power_controller_inst/restore_1 low}
-assert_rs_mutex {clock_a posedge}
```

then `query_retention_control * -domain PD1` returns all the defined retention strategies for domain `PDA`, i.e., `{my_retention_strategy}`. `query_retention_control my_retention_strategy -domain PDA` returns the retention control information in the format of the corresponding **set_retention_control** command, as defined above. `query_retention_control my_retention_strategy -domain PDA -detailed` returns the information as a list of {*key value*} pairs, i.e.,

```
{retention_name my_retention_strategy} {domain PDA} {save_signal
   {power_controller_inst/save_1 high}} {restore_signal
   {power_controller_inst/restore_1 low}} {assert_rs_mutex {clock_a posedge}}
   {assert_r_mutex {}} {assert_s_mutex {}}
```

It shall be an error if

— **-detailed** is specified and *retention_name* is *****.

— The specified *domain_name* starts with `..` or `/`, i.e., the domain needs to be referenced from the active scope, and reside in the active scope or below it.

Syntax example:

```
query_retention_control my_retention_strategy -domain PDA
```

7.33 query_retention_elements

Purpose	Query the retention strategy elements	
Syntax	**query_retention_elements** *retention_list_name* [**-detailed**]	
Arguments	*retention_list_name*	Specifies the retention element group identifier to be queried. If ***** is specified then a list of retention element group identifiers shall be returned (or a *null string* if no groups have been previously defined).
	-detailed	Returns the retention elements as a list of **{***key value***}** pairs, where *key* is the name of an argument of the **set_retention_elements** command (any **-** prefixes are removed) and *value* is the value of that argument. Valid keys are **retention_list_name**, **elements**, **applies_to**, and **retention_purpose**.
Return value	There are three distinct return structures. a) If a ***** is specified for *retention_list_name*, then a list of the defined retention group identifiers shall be returned. b) If a previously defined retention group identifier is specified for *retention_list_name* and **-detailed** is not specified then the retention group information shall be returned in the format of the corresponding **set_retention_elements** command (see 6.49). c) If **-detailed** is specified then the retention group information for the specified *retention_list_name* shall be returned as a list of **{***key value***}** pairs.	

The **query_retention_elements** command returns the list of objects that can be used in a **set_retention_elements** command.

If a retention elements definition is

```
set_retention_elements my_retention_group -elements {state_reg}
   -exclude_elements {awake_from_sleep_reg}
```

then `query_retention_elements * ` returns all defined retention element groups, i.e., `my_retention_group`. `query_retention_elements my_retention_group` returns the retention elements in the form of the **set_retention_elements** command, as defined above. `query_retention_elements my_retention_group -detailed` returns the retention elements as a list of **{***key value***}** pairs, i.e.,

```
{retention_list_name my_retention_group} {elements {state_reg}}
   {exclude_elements {awake_from_sleep_reg}}
```

It shall be an error if **-detailed** is specified and *retention_list_name* is *****.

Syntax example:

```
query_retention_elements my_retention_group
```


7.34 query_simstate_behavior

Purpose	Query the simstate behavior information for a domain	
Syntax	**query_simstate_behavior** **-lib** *name* [**-model** *name*] [**-detailed**]	
Arguments	**-lib** *name*	Specifies the library name.
	-model *name*	Specifies the model name or use * to query all models in the given library.
	-detailed	Returns the simstate behavior as a list of {*key value*} pairs, where *key* is the name of an argument of the **set_simstate_behavior** command (any - prefixes are removed) and *value* is the value of that argument. Valid keys are **simstate_behavior**, **lib**, and **model**.
Return value	There are two distinct return structures. a) If **-detailed** is not specified then the simstate behavior information shall be returned in the format of a corresponding **set_simstate_behavior** command. b) If **-detailed** is specified then the simstate behavior shall be returned as a list of {*key value*} pairs.	

The **query_simstate_behavior** command queries the simulation simstate behavior for a model or a library.

If a simstate is defined as

```
set_simstate_behavior ENABLE -lib library1 -model ANDX7_non_power_aware
```

then `query_simstate_behavior -lib library1 -model ANDX7_non_power_aware` returns the simstate behavior for the specified model in the format of the corresponding **set_simstate_behavior** command, as defined above. `query_simstate_behavior -lib library1 -model ANDX7_non_power_aware -detailed` returns the simstate information as a list of {*key value*} pairs, i.e.,

```
{{lib library1} {simstate_behavior ENABLE} {model ANDX7_non_power_aware}}
```

If **-model** * is specified, the simstate information shall be returned for all models in the specified library. Because different models can have different simstate behaviors, a list of a list shall be returned for the two behaviors, i.e.,

```
{{simstate_behavior } {lib } {model } {model } ...} {{simstate_behavior }
   {lib } {model } {model } ...}
```

If a simstate is defined as

```
set_simstate_behavior ENABLE -lib library1 -model ANDX7_non_power_aware
set_simstate_behavior DISABLE -lib library1 -model NANDX7_power_aware
```

then a detailed simstate query returns

```
{{simstate_behavior ENABLE} {lib library1} {model ANDX7_non_power_aware}}
   {{simstate_behavior DISABLE} {lib library1} {model NANDX7_power_aware}}
```

Syntax example:

```
query_simstate_behavior -lib library1 -model ANDX7_non_power_aware
```

7.35 query_state_transition

Purpose	Query a state transition	
Syntax	**query_state_transition** *transition_name* **-object** *object_name* [**-from** {*from_state_list*}] [**-to** {*to_state_list*}] [**-paired** {*paired_state_list*}] [**-legal** | **-illegal**] [**-detailed**]	
Arguments	*transition_name*	Specifies the *transition_name* to query. If * is specified then all state transitions for the specified *object_name* shall be returned as a Tcl list.
	-object *object_name*	Name of a power domain or supply set for which the state transition information shall be queried.
	-from {*from_state_list*}	If *transition_name* is *, then *from_state_list* can be used to filter the returned transitions. A transition name shall only be returned if it starts from any one of the states in *from_state_list*.
	-to {*to_state_list*}	If *transition_name* is *, then *to_state_list* can be used to filter the returned transitions. A transition name shall only be returned if it ends at any one of the states in *to_state_list*.
	-paired {*paired_state_list*}	If *transition_name* is *, then *paired_state_list* can be used to filter the returned transitions. A transition name shall only be returned if it ends at any one of the states in *paired_state_list*.
	-legal | **-illegal**	If *transition_name* is *, then **-legal** or **-illegal** can be specified to restrict the returned transition names. If neither are specified then both illegal and legal transitions shall be returned.
	-detailed	Returns the transition information as a list of {*key value*} pairs, where *key* is the name of an argument of the **describe_state_transition** command (any **-** prefixes are removed) and *value* is the value of that argument. Valid keys are **transition_name**, **object**, **from**, **to**, **paired, legal**, and **illegal**.
Return value	There are three distinct return structures. a) If * is specified for *transition_name* then the name of all transitions for the specified *object_name* shall be returned as a Tcl list (a *null string* shall be returned if no state transitions are defined). Returned transition names shall be filtered by the **-from**, **-to**, **-paired, -illegal**, and **-legal** arguments, if specified. b) If *transition_name* is specified (and it is not *) then the state transition information shall be returned in the form of the corresponding **describe_state_transition** command. c) If **-detailed** is specified then the state information for the specified *transition_name* shall be returned as {*key value*} pairs.	

The **query_state_transition** command queries state transition information. All transition states for a specified *object_name* can be queried as a Tcl list if * is specified for *transition_name*. The **-from, -to**, **-paired**, **-legal**, and **-illegal** arguments can be used to filter the returned state transitions when *transition_name* is *.

If a state transition is specified as

```
describe_state_transition turn_on -object PdA -from {SLEEP_MODE} -to {GO_MODE}
   -paired {DROWSY SLEEP_MODE} -legal
```

then `query_state_transition * -object PdA` returns a Tcl list of all defined state transitions for `PdA`, i.e., `{turn_on}`. `query_state_transition * -object PdA -from SLEEP_MODE`, returns any state transitions starting from the state `SLEEP_MODE`. `query_state_transition turn_on -object PdA -detailed` returns the state transition information as a list of **{***key value***}** pairs, i.e.,

```
{transition_name turn_on} {object PdA} {from {SLEEP_MODE}} {to {GO_MODE}}
   {paired {{DROWSY SLEEP_MODE}} {legal 1} {illegal 0}
```

It shall be an error if

— *transition_name* is not ***** and **-from**, **-to**, **-paired**, **-legal**, **-illegal**, or **-detailed** is specified.

— *transition_name* is not a transition state.

Syntax example:

```
query_state_transition * -object PdA
```

7.36 query_supply_net

Purpose	Query a supply net	
Syntax	**query_supply_net** *net_name* [**-domain** *domain_name*] [**-detailed**]	
Arguments	*net_name*	Specifies the net_name to query. If ***** is specified then the name of all supply nets shall be returned as a Tcl list.
	-domain *domain_name*	Restricts the query to a specified *domain_name*.
	-is_supply | **-detailed**	If **-is_supply** is specified then a `1` shall be returned if the specified *net_name* is a supply port and a `0` shall be returned if it is not. If **-detailed** is specified then the supply port information shall be returned as a list of **{***key value***}** pairs, where *key* is the name of an argument of the **create_supply_net** command (any **-** prefixes are removed) and *value* is the value of that argument. Valid keys are **net_name**, **domain**, and **resolve**.
Return value	There are four distinct return structures. a) If ***** is specified for *net_name* then all previously defined supply nets shall be returned as a Tcl list (a *null string* shall be returned if no supply nets are defined). b) If *net_name* is specified (and it is not *****) then the net information shall be returned. c) If **-detailed** is specified then the net information for the specified *net_name* shall be returned as **{***key value***}** pairs. d) if **-is_supply** is specified then a `1` shall be returned if the specified *net_name* is a supply port; otherwise, a `0` shall be returned.	

The **query_supply_net** command returns the information about a previously created supply net. When called with the **-is_supply** argument, this query can be used to check if the specified *net_name* is a supply net. The **-domain** option restricts the query to the specified *domain_name*.

If a supply net is created as follows

```
create_supply_net oneh_supply -resolve one_hot
```

then `query_supply_net *` returns all supply nets in and below the active scope, i.e., `oneh_supply`. `query_supply_net oneh_supply` returns the supply net information in the format of the corresponding **create_supply_net** command, as defined above. `query_supply_net oneh_supply -detailed` returns the supply net information as a list of {*key value*} pairs, i.e.,

```
{net_name oneh_supply} {resolve {one_hot}}
```

The following also apply:

— It shall be an error if **-detailed**, **-is_supply**, or **-supply_set** is specified and * is specified for *net_name*.

— *net_name* is not a supply net unless **-is_supply** is specified.

Syntax example:

```
query_supply_net andrews_net -is_supply
```

7.37 query_supply_port

Purpose	Query a supply port	
Syntax	**query_supply_port** *port_name* [**-domain** *domain_name*] [**-is_supply** | **-detailed**]	
Arguments	*port_name*	Specifies the *port_name* to query. If * is specified then the name of all supply ports shall be returned as a Tcl list. By default, ports are listed on the active scope unless **-domain** is specified.
	-domain *domain_name*	Restricts the query to the interface of the specified *domain_name*.
	is_supply | **-detailed**	If **-is_supply** is specified then a `1` shall be returned if the specified *port_name* is a supply port and a `0` shall be returned if it is not. If **-detailed** is specified then the supply port information shall be returned as a list of {*key value*} pairs, where *key* is the name of an argument of the **create_supply_port** command (any **-** prefixes are removed) and *value* is the value of that argument. Valid keys are **port_name** and **direction**.
Return value	There are four distinct return structures. a) If * is specified for *port_name* then all previously defined supply ports shall be returned as a Tcl list (a *null string* shall be returned if no supply ports are defined). b) If *port_name* is specified (and it is not *) then the port information shall be returned. c) If **-detailed** is specified then the port information for the specified *port_name* shall be returned as {*key value*} pairs. d) if **-is_supply** is specified then a `1` shall be returned if the specified *port_name* is a supply port; otherwise, a `0` shall be returned.	

The **query_supply_port** command returns the information about a previously created supply port. When called with the **-is_supply** argument, this query can be used to check if the specified *port_name* is a supply port. The **-domain** option restricts the query to the interface of the specified *domain_name*. The interface of a domain in this context is the logical hierarchy boundary between one domain and another, or between a domain and the top-level scope.

If a supply port is created as

```
create_supply_port VN1 -direction inout
```

then `query_supply_port *` returns all supply ports on the active scope, i.e., `{VN1}`. `query_supply_port VN1` returns the supply port information in the format of the corresponding **create_supply_port** command, as defined above. `query_supply_port VN1 -detailed` returns the supply port information as a list of {*key value*} pairs, i.e.,

```
{port_name VN1} {direction inout}
```

The following also apply:

— It shall be an error if **-is_supply** or **-detailed** are specified and *port_name* is *.

— It shall be an error if *port_name* is not * and **-domain** is specified.

— *port_name* is not a supply port, unless **-is_supply** is specified.

Syntax example:

```
query_supply_port joes_port -is_supply
```

7.38 query_supply_set

Purpose	Query a supply set	
Syntax	**query_supply_set** *set_name* [**-detailed**] [**-transitive** <**TRUE** | **FALSE**>]	
Arguments	*set_name*	Specifies the supply set *set_name* to query. If * is specified then the name of all supply sets shall be returned as a Tcl list.
	-detailed	If **-detailed** is specified then the supply set information shall be returned as a list of {*key value*} pairs, where *key* is the name of an argument of the **create_supply_set** command (any **-** prefixes are removed) and *value* is the value of that argument. Valid keys are **set_name**, **function**, and **reference_gnd**.
	-transitive <**TRUE** | **FALSE**>	When **-transitive** is **TRUE**, the command applies to the descendants of the elements; the default is **FALSE**.
Return value	There are three distinct return structures. a) If * is specified for *set_name* then all previously defined supply sets in the active scope (and below if **-transitive** is specified) shall be returned as a Tcl list (a *null string* shall be returned if no supply ports are defined). b) If *set_name* is specified (and it is not *) then the supply set information shall be returned in the form of the corresponding **create_supply_set** command. c) If **-detailed** is specified then the supply set information for the specified *set_name* shall be returned as {*key value*} pairs.	

The **query_supply_set** commands queries any previously defined supply sets.

If a supply set is created as

```
create_supply_set relative_always_on_ss
-function {power vdd}
```

```
-function {ground vss}
-reference_gnd {earth_ground}
```

then `query_supply_set *` returns the names of any previously created supply sets, i.e., `{relative_always_on_ss}`. `query_supply_set relative_always_on_ss` returns the supply set information in the format of the corresponding **create_supply_set** command, as defined above. `query_supply_set relative_always_on_ss -detailed` returns the supply set information using {*key value*} pairs, i.e.,

```
{set_name relative_always_on_ss} {function {{power vdd} {ground vss}}}
   {reference_gnd {earth_ground}}
```

It shall be an error if

— **-detailed** is specified and *set_name* is *.

— **-transitive** is specified and *set_name* is not *.

Syntax example:

```
query_supply_set relative_always_on_ss
```

7.39 query_upf2hdl_vct

Purpose	Query a value conversion table	
Syntax	**query_upf2hdl_vct** *vct_name* [**-detailed**]	
Arguments	*vct_name*	Specifies the *vct_name* to query. If * is specified then the name of all defined VCTs shall be returned as a Tcl list.
	-detailed	Returns the VCT information as a list of {*key value*} pairs, where *key* is the name of an argument of the **create_upf2hdl_vct** command (any **-** prefixes are removed) and *value* is the value of that argument. Valid keys are **vct_name**, **hdl_type**, and **table**.
Return value	There are three distinct return structures. a) If * is specified for *vct_name* then all previously defined VCTs shall be returned as a Tcl list (a *null string* shall be returned if no VCTs are defined). b) If *vct_name* is specified (and it is not *) then the VCT information shall be returned in the form of the corresponding **create_upf2hdl_vct** command. c) If **-detailed** is specified then the VCT information for the specified *vct_name* shall be returned as {*key value*} pairs.	

The **query_upf2hdl_vct** command queries can list and query any previously defined value conversion table.

If a VCT is created as

```
create_upf2hdl_vct upf2vlog_vdd
-hdl_type {sv}
-table {{OFF X} {FULL_ON 1} {PARTIAL_ON 0}}
```

then `query_upf2hdl_vct *` returns `upf2vlog_vdd`. `query_upf2hdl_vcd upf2vlog_vdd` returns the VCT information in the format of the corresponding **create_upf2hdl_vct** command, as defined

above. `query_upf2hdl_vct upf2vlog_vdd -detailed` returns the VCT information as {*key value*} pairs, i.e.,

```
{vct_name upf2vlog_vdd} {hdl_type {sv}} {table {{OFF X} {FULL_ON 1} {PARTIAL_ON
    0}}}
```

It shall be an error if **-detailed** is specified and * is specified for *vct_name*.

Syntax example:

```
query_upf2hdl_vcd upf2vlog_vdd
```

7.40 query_use_interface_cell

Purpose	Query the interface cell information for a domain	
Syntax	**query_use_interface_cell** *interface_implementation_name* **-strategy** *list_of_isolation_level_shifter_strategies* **-domain** *domain_name* [**-detailed**]	
Arguments	*interface_implementation_name*	Specifies the *interface_implementation_name* to be queried. If * is specified then a list of interface implementation names shall be returned (or a *null string* if none have been previously defined).
	-strategy *list_of_isolation_level_shifter_strategies*	Specifies the levelshifter or isolation strategy for which the *interface_implementation_name* is to be queried.
	-domain *domain_name*	Specifies the *domain_name* for which the *list_of_isolation_level_shifter_strategies* is defined.
	-detailed	Returns the interface cell information as a list of {*key value*} pairs, where *key* is the name of an argument of the **use_interface_cell** command (any **-** prefixes are removed) and *value* is the value of that argument. Valid keys are **interface_implementation_name**, **strategy**, **domain**, **lib_cells**, **map**, **elements**, **with_clamp**, **update_any**, **force_function**, and **inverter_supply_set**
Return value	There are three distinct return structures. a) If a * is specified for *interface_implementation_name*, then a list of the defined interface implementation identifiers shall be returned. b) If a previously defined interface implementation identifier is specified for *interface_implementation_name* and **-detailed** is not specified, then the interface implementation information shall be returned in the format of the corresponding **use_interface_cell** command (see 6.53). c) If **-detailed** is specified then the interface implementation information for the specified *interface_implementation_name* shall be returned as a list of {*key value*} pairs.	

The **query_use_interface_cell** command provides the ability to query the interface cell information for a specific *interface_implementation_name*.

If an interface cell is specified as

```
use_interface_cell my_interface -strategy {ISO1 LS1} -domain PD1
   -elements {top/moduleA/port1 top/moduleA/port2 top/moduleA/port3}
   -lib_cells LS_ISO_COMBO
```

then `query_use_interface_cell * -domain PD1 -strategy ISO1` returns all the interface cell specifications defined for strategy `ISO1` on domain `PD1`. `query_use_interface_cell my_interface -domain PD1 -strategy ISO1` returns the interface cell information in the format of the corresponding **use_interface_cell** command for the strategy `ISO1`, as defined above. `query_use_interface_cell my_interface -domain PD1 -strategy ISO1 -detailed` returns the interface cell information as a list of **{***key value***}** pairs, i.e.,

```
{{interface_implementation_name my_interface} {strategy ISO1} {domain PD1}
   {lib_cells CLASS1} {map {}} {elements {top/moduleA/port1 top/moduleA/port2
   top/moduleA/port3}} {with_clamp {}} {update_any {}} {force_function 0}
   {inverter_supply_set {}}}
```

It shall be an error if **-detailed** is specified and *interface_implementation_name* is *****.

Syntax example:

```
query_use_interface_cell * -domain PD1 -strategy ISO1
```

8. Switching Activity Interchange Format (SAIF)

The Switching Activity Interchange Format (SAIF) is designed to assist in the extraction and storing of the switching activity information generated by EDA (electronic design automation) tools.

A SAIF file containing switching activity information can be generated using an HDL simulator and then the switching activity can be back-annotated into the power analysis/optimization tool as shown in Figure 9. This type of SAIF file is called a *backward SAIF file*.

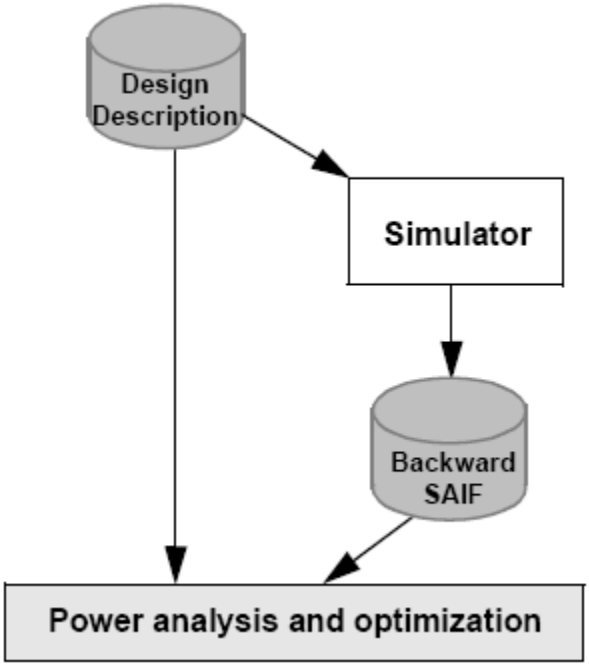

Figure 9—Backward SAIF file

The power analysis/optimization tool, or some other EDA tool, may issue directives (instructions) to the backward SAIF file generation application on the format of the required SAIF file. These directives can be stored into a SAIF file, called a *forward SAIF file*, as shown in Figure 10.

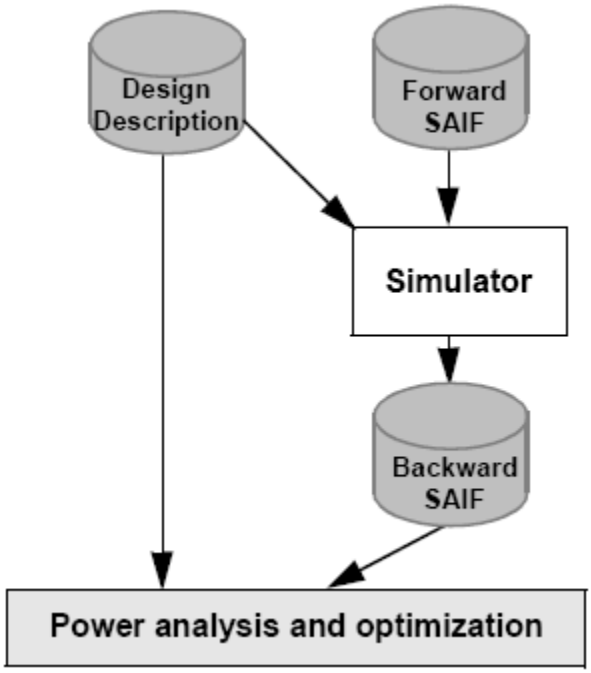

Figure 10—Forward SAIF file

This clause provides the syntax and semantics of the backward SAIF file and the following two kinds of forward SAIF files:

a) The library or gate-level forward SAIF file, which contains the directives for generating state-dependent and path-dependent switching activity.

b) The RTL forward SAIF file, which contains the directives for generating switching activity from the simulation of RTL hardware descriptions.

For examples of SAIF files, see [B7].

8.1 Syntactic conventions

The syntax of the SAIF file is described using the Backus-Naur Form (BNF), as follows:

Lowercase words (some containing underscores) are used to denote syntactic categories, e.g.,

backward_instance_info

Boldface words are used to denote the reserved keywords, operators and punctuation marks that are a required part of the syntax, e.g.,

INSTANCE * ( )

A non-boldface vertical bar (|) separates alternative items, e.g.,

```
binary_operator ::=
      * | ^ | |
```


Note that the last vertical bar is in **boldface** and therefore represents an actual operator rather than a separator between the alternative operators.

Non-boldface square brackets (**[]**) enclose optional items, e.g.,

date ::=
(DATE [string]**)**

Non-boldface braces (**{}**) enclose items that can be repeated 0 or more times, e.g.,

backward_saif_info ::=
{backward_instance_info}

8.2 Lexical conventions

SAIF files are a stream of lexical tokens that consist of one or more characters. Except for one-line comments (see below), the layout of SAIF files is free-format, i.e., spaces and newlines are only syntactically significant as token separators.

The following are types of *lexical tokens* in SAIF files:

- white space
- comments
- numbers
- strings
- parenthesis
- operators
- hierarchical separator character
- identifiers
- keywords

The rest of this subclause describes the lexical tokens used in SAIF files and their conventions.

8.2.1 White space

White spaces are sequences of spaces, tabs, newlines, and form-feeds. White spaces separate the other lexical tokens.

8.2.2 Comments

The SAIF format allows for both one-line comments and block comments. *One-line comments* start with the character sequence // and end with a newline. *Block comments* start with the character sequence /* and end with the first occurrence of the sequence */. Block comments are not nested.

8.2.3 Numbers

Numbers in SAIF files are either

- Non-negative decimal integers, which are represented by a sequence of decimal characters, e.g., `12`, `012`, or `1200`.
- Non-negative real numbers, which are non-negative IEEE standard double precision floating point number representations, e.g., `1`, `3.4`, `.7`, `0.3`, `2.4e2`, or `5.3e-1`.

8.2.4 Strings

A *string* in SAIF files is a possibly empty sequence of characters enclosed by the double quotes character ("") and contained on a single line, e.g.,`"SAIF version 2.0"` or `""`.

8.2.5 Parenthesis

Most of the constructs in SAIF files are enclosed between the left-parenthesis character (() and the right-parenthesis character ()).

8.2.6 Operators

An *operator* in SAIF files is one of the following characters: !, *, ^, and |. Operators are used in conditional expressions.

8.2.7 Hierarchical separator character

The *hierarchical separator* is a special character used in composing hierarchical port/pin/net/instance names from simple identifiers. The hierarchical separator character is defined in the header of SAIF files and can be either the / character or the . character.

8.2.8 Identifiers

A SAIF *identifier* is a non-empty sequence of alphanumeric characters, the underscore character (_) and escaped characters, followed by an optional decimal number enclosed in brackets ([]). *Escaped identifiers* consist of the \ character followed by a non-white space character. A SAIF identifier cannot start with a decimal digit (.) character and cannot contain the hierarchical separator character, unless it is escaped. The \ character used in an escaped character is not part of the identifier, so `abc` and `a\b\c` represent the same identifier. SAIF identifiers are case-sensitive, `abc` and `ABC` represent two different identifiers.

Examples:

`clk`, `clk_net`, `clk[4]`, `clk\#4`, `clk\(4\)`, `\1clk`, or `mod\/net`

where the hierarchical separator character is presumed to be `/`.

8.2.9 Keywords

A SAIF *keyword* is a special sequence of alphanumeric characters. SAIF keywords can be used as identifiers; to avoid possible ambiguity, escape the first character of identifiers that can be mistaken for keywords. SAIF keywords are case-sensitive. Table 8 shows the set of SAIF keywords.

Table 8—SAIF keywords

COND	**LEAKAGE**	**TB**
COND_DEFAULT	**LIBRARY**	**TC**
DATE	**MODULE**	**TG**
DESIGN	**NET**	**TIMESCALE**
DIRECTION	**PORT**	**TX**
DIVIDER	**PROGRAM_NAME**	**TZ**
DURATION	**PROGRAM_VERSION**	**VENDOR**
FALL	**RISE**	**VIRTUAL_INSTANCE**
IG	**RISE_FALL**	**fs**
IK	**SAIFILE**	**ms**
INSTANCE	**SAIFVERSION**	**ns**
IOPATH	**T0**	**ps**
IOPATH_DEFAULT	**T1**	**s**
		us

8.2.10 Syntactic categories for token types

The syntax of the SAIF files described in this document use the syntactic categories shown in Table 9 for token types.

Table 9—Token type categories

Syntactic category	Token type
dnumber	Non-negative integer numbers
rnumber	Non-negative real numbers
string	Strings
hchar	Possible hierarchical separator characters
identifier	Simple (non-hierarchical) identifiers
hierarchical_identifier	Hierarchical identifiers

8.3 Backward SAIF file

This subclause describes the format of the *backward SAIF file*, which contains hierarchical instance-specific switching activity information.

8.3.1 SAIF file

The backward SAIF file consists of a left-parenthesis (( ), the **SAIFILE** keyword, the backward SAIF header, the backward SAIF info, and a right-parenthesis ( )), as shown in Syntax 1.

```
backward_saif_file ::=
    (SAIFILE backward_saif_header backward_saif_info)
```

Syntax 1—backward_saif_file

8.3.2 Header

Syntax 2 defines the backward SAIF file header.

```
backward_saif_header ::=
    backward_saif_version
    direction
    design_name
    date
    vendor
    program_name
    program_version
    hierarchy_divider
    time_scale
    duration
```

Syntax 2—backward_saif_header

Each backward SAIF header construct is described in the following subclauses.

8.3.2.1 backward_saif_version

Syntax 3 defines the `backward_saif_version`.

```
backward_saif_version ::=
    (SAIFVERSION string)
```

Syntax 3—backward_saif_version

The *string* in this construct represents the version number of the SAIF file, i.e., **2.0**.

8.3.2.2 direction

Syntax 4 defines the `direction`.

```
direction ::=
    (DIRECTION string)
```

Syntax 4—direction

The *string* in this construct represents the type of the SAIF file, i.e., **backward**.

8.3.2.3 design_name

Syntax 5 defines the `design_name`.

```
design_name ::=
    (DESIGN [string])
```

Syntax 5—design_name

The optional *string* in this construct represents the design for which the switching activity in the SAIF file has been generated.

8.3.2.4 date

Syntax 6 defines the `date`.

```
date ::=
    (DATE [string])
```

Syntax 6—date

The optional *string* in this construct represents the date the SAIF file was generated.

8.3.2.5 vendor

Syntax 7 defines the `vendor`.

```
vendor ::=
    (VENDOR [string])
```

Syntax 7—vendor

The optional *string* in this construct represents the name of the vendor whose application was used to generate the SAIF file.

8.3.2.6 program_name

Syntax 8 defines the `program_name`.

```
program_name ::=
    (PROGRAM_NAME [string])
```

Syntax 8—program_name

The optional *string* in this construct represents the name of the application used to generate the SAIF file.

8.3.2.7 program_version

Syntax 9 defines the `program_version`.

The optional *string* in this construct represents the version number of the application used to generate the SAIF file.

```
program_version ::=
    (PROGRAM_VERSION [string])
```

Syntax 9—program_version

8.3.2.8 hierarchy_divider

Syntax 10 defines the `hierarchy_divider`.

```
hierarchy_divider ::=
    (DIVIDER [hchar])
```

Syntax 10—hierarchy_divider

The optional *hchar* in this construct represents the hierarchical separator character used in hierarchical identifiers. Only the **/** and **.** characters shall be specified as the hierarchical separator character; the default is the **.** character.

8.3.2.9 time_scale

Syntax 11 defines the `time_scale`.

```
time_scale ::=
    (TIMESCALE [dnumber timeunit])
timeunit ::=
    s | ms | us | ns | ps | fs
```

Syntax 11—time_scale

This construct specifies the units used for all time values in the SAIF file. The *dnumber* shall be **1**, **10**, or **100**; it represents the scaling factor of the time values. For example, if the `time_scale` of a SAIF file is

```
(TIMESCALE 100 us)
```

then all the time values in the SAIF file are specified in hundreds of microseconds. If the decimal number and time unit are not specified, the default time scale is `1 ns`.

8.3.2.10 duration

Syntax 12 defines the `duration`.

```
duration ::=
    (DURATION rnumber)
```

Syntax 12—duration

This construct specifies the total time duration applied to the switching activity in the SAIF file.

8.3.2.11 Example

This is an example of a valid backward SAIF file header.

```
(SAIFVERSION "2.0")
```

```
(DIRECTION "backward")
(DESIGN "alu")
(DATE "Fri Jan 18 10:30:00 PDT 2002")
(VENDOR "SAIF'R'US Corp.")
(PROGRAM_NAME "saifgenerator")
(PROGRAM_VERSION "1.0")
(DIVIDER /)
(TIMESCALE 1 ns)
(DURATION 5000)
```

8.3.3 Simple timing attributes

This construct specifies the total duration (in time values) that some particular design net/port/pin (specified elsewhere) has some particular value. Syntax 13 defines this construct.

```
simple_timing_attribute ::=
      (T0 rnumber)
    | (T1 rnumber)
    | (TX rnumber)
    | (TZ rnumber)
    | (TB rnumber)
```

Syntax 13—simple_timing_attribute

The different types of simple timing attributes are as follows:

— **T0** is the total time the design object has the value `0`.

— **T1** is the total time the design object has the value `1`.

— **TX** is the total time the design object has an unknown value.

— **TZ** is the total time the design object is in a floating bus state. A *floating bus state* is the state when all drivers on a particular bus are disabled and the bus has a floating logic value.

— **TB** is the total time the design object is in a bus contention state. A *bus contention state* is the state when two or more drivers simultaneously drive a bus to different logic levels.

Example:

If the time scale is 100 µs, then the following three simple timing attribute constructs:

```
(T0 100)
(T1 92.5)
(TX 7.5)
```

specify a particular design object has the value `0` for a total `10,000 us`, the value `1` for a total of `9250 us`, an unknown value for a total of `750 us`, and it never reaches the floating bus and bus contention states.

8.3.4 Simple toggle attributes

This attribute construct specifies the number on a particular type of toggle registered on a particular design net/port/ pin (specified elsewhere). Syntax 14 defines this construct.

The different types of simple toggle attributes are as follows:

— **TC** is the number of `0` to `1` plus the number of `1` to `0` transitions. This is usually referred to as the *toggle count*.

```
simple_toggle_attribute ::=
      (TC rnumber)
    | (TG rnumber)
    | (IG rnumber)
    | (IK rnumber)
```

Syntax 14—simple_toggle_attribute

— **TG** is the number of transport glitch edges (see 8.3.4.1).

— **IG** is the number of inertial glitch edges (see 8.3.4.2).

— **IK** is the inertial glitch de-rating factor. To estimate this factor, see Annex E.

Example:

The following simple toggle attributes:

```
(TC 200)
(IG 6)
```

specify a total of 200 transitions between the `0` and `1` logic states, and a total of six inertial glitch edges are registered on some particular design object(s).

8.3.4.1 Transport glitch

*Transport glitche*s are extra transitions at the output of the gate before the output signal reaches its steady state and, unlike inertial glitches (see 8.3.4.2), can not be canceled by an inertial delay algorithm. A transport glitch consumes the same amount of power as a normal toggle transition does and is an ideal candidate for power minimization during the optimization process. Transport glitches at the output of the gate have a pulse width longer than the gate delay and do not contribute to the functional behavior of the circuit.

In general, the number of transport glitch transitions occurring in the circuit is the difference between the total number of toggle transitions obtained from a full-timing simulation and that from a cycle-based simulation, assuming all inertial glitches (see 8.3.4.2) have been filtered out by the timing simulator, i.e., the total number of toggles obtained from the timing simulator does not include inertial glitches. Figure 11 shows a possible way to have transport glitches in the circuit. While steady state analysis of the circuit indicates node `N`, the output of the `XOR` gate, should always remain logic `1` regardless of the primary input, the additional timing delay by the inverter causes a glitch at `N` whenever the input changes its state.

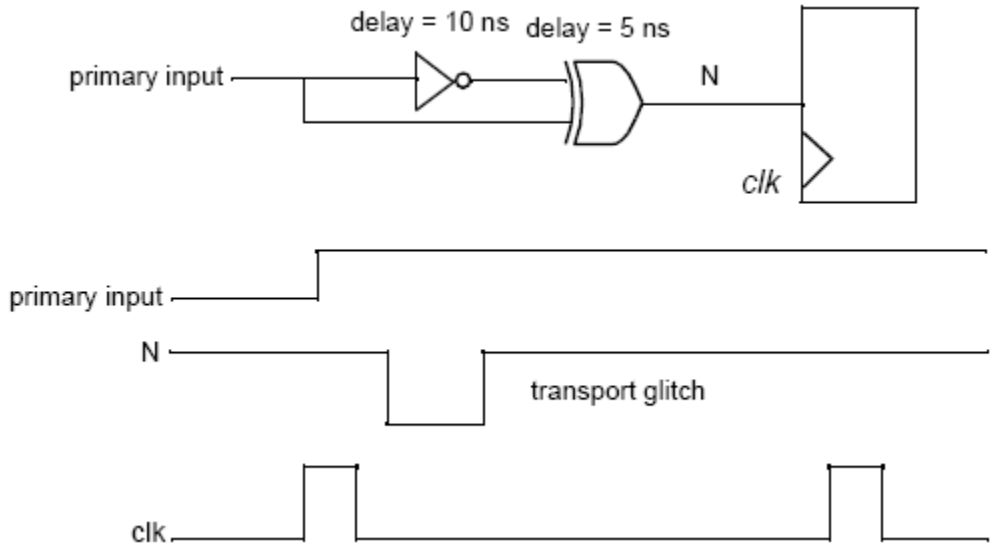

Figure 11—Transport glitch

8.3.4.2 Inertial glitch

Inertial glitches are signal transitions occurring at the output of the gate, which can be filtered out if an inertial delay algorithm is applied. A simple example (see Figure 12) best explains inertial glitches.

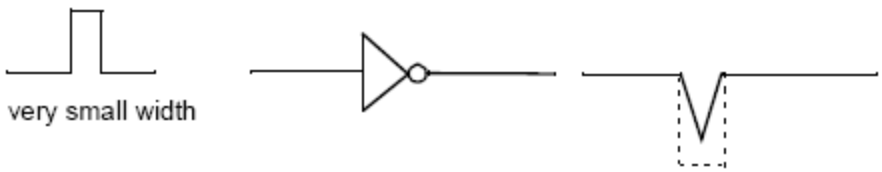

Figure 12—Inertial glitch

A VHDL description for this inverter looks something like:

`OUT` ← `not IN after 5 ns` (inertial delay is implicitly presumed)

If the input pulse has a width less than `5 ns`, the inertial delay algorithm shall cancel the signal transitions at the output of the inverter. However, some power is still consumed due to the two partial transitions at the output. Therefore, it is necessary to report these two inertial glitch transitions in a SAIF file.

NOTE—SAIF counts the number of glitches by signal edges, not signal pulses.

8.3.5 State-dependent timing attributes

State-dependent timing attributes specify the time duration when a cell is in particular states. The *state* of a cell is defined as the logic value of its pins. Syntax 15 defines this construct.

```
state_dep_timing_attributes ::=
      (state_dep_timing_item {state_dep_timing_item}
      [COND_DEFAULT sd_simple_timing_attributes])
state_dep_timing_item ::=
      COND cond_expr sd_simple_timing_attributes
cond_expr ::=
       port_name
      | unary_operator cond_expr
      | cond_expr binary_operator cond_expr
      | (cond_expr)
port_name ::=
      identifier
unary_operator ::=
      !
binary_operator ::=
      * | ^ | |
sd_simple_timing_attributes ::=
      {sd_simple_timing_attribute}
sd_simple_timing_attribute ::=
       (T1 rnumber)
      | (T0 rnumber)
```

Syntax 15—state_dep_timing_attributes

Here `cond_expr` represents conditional expressions on pin names; `sd_simple_timing_attribute` can only contain one of the following:

— **T1** is the total time duration in which the cell is in any of its associated states.

— **T0** is the total time duration in which the cell is not in any of its associated states.

A *conditional expression* specifies the set of states for which the condition holds. For example, given a cell with, three inputs, `A`, `B`, and `C`, and one output `Y`, the conditional expression

```
A | B
```

represents all the cell states when the input pin `A` is `1` or the input `B` is `1`, while `C` and `Y` can have any value.

The precedence of the operators in conditional expressions is by the following sequence: **!** (logical not), ***** (logical and), ^ (logical exclusive or), and | (logical or), where **!** has the highest precedence.

A state-dependent timing attribute construct

```
(COND expr1 attrs1
COND expr2 attrs2
...
COND exprn attrsn
COND_DEFAULT attrs_default)
```

determines a priority-encoded specification of the timing attributes `attrs1`, `...`, `attrs_default`, i.e., the attributes `attrs1` apply for the set of states for which the condition `expr1` holds, while the attributes `attrs2` apply for the set of states where the condition `expr2` holds and `expr1` does not hold, etc. The attributes `attrs_default` apply for all the states where none of the conditional expressions hold.

Example:

The state-dependent timing attributes of the cell given in Figure 13 during the time duration given in the wave diagram in Figure 14 can be specified as follows:

```
(COND (A * B * Y) (T1 1) (T0 8)
COND (!A * B * Y) (T1 1) (T0 8)
COND (A * !(B * C)) (T1 2) (T0 7)
COND B (T1 1) (T0 8)
COND C (T1 1) (T0 8)
COND_DEFAULT (T1 3) (T0 6))
```

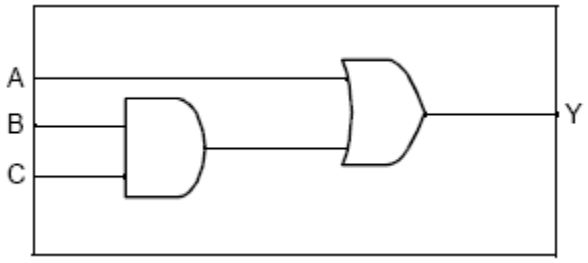

Figure 13—A cell and its internal behavior

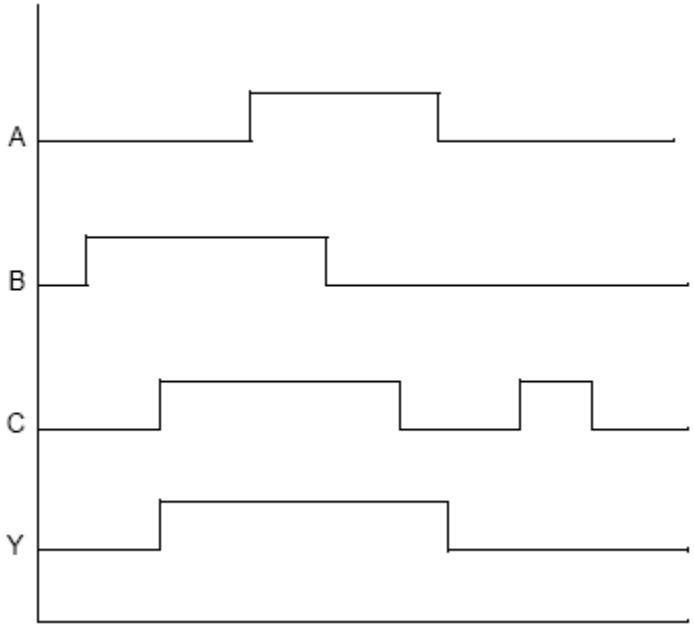

Figure 14—A wave diagram

8.3.6 State-dependent toggle attributes

The toggle attributes on cell pins can be *state dependent*, i.e., the attributes are relevant only to particular cell states. Syntax 16 defines this construct.

```
state_dep_toggle_attributes ::=
      (state_dep_toggle_item {state_dep_toggle_item}
      [state_dep_default_toggle_item])
state_dep_toggle_item ::=
      COND cond_expr [(edge_type)] simple_toggle_attribute
state_dep_default_toggle_item ::=
       COND_DEFAULT simple_toggle_attribute
      | COND_DEFAULT (edge_type) simple_toggle_attribute
       [COND_DEFAULT (edge_type) simple_toggle_attribute]
edge_type ::=
      RISE | FALL
```

Syntax 16—state_dep_toggle_attributes

Similar to state-dependent timing attributes, the state-dependent toggle attributes construct represents a priority-encoded attribute specification. The optional `edge_type` is used to further differentiate the toggle count between `0` to `1` (**RISE**) and `1` to `0` (**FALL**) transitions.

The state-dependent toggle attributes construct can end with an optional **COND_DEFAULT** specification that has no edge restrictions. Otherwise, it can end with up to two **COND_DEFAULT** specifications having different edge restrictions.

Example:

The following state-dependent toggle attributes construct:

```
(COND A (RISE) (TC 20)
COND A (FALL) (TC 15)
COND B (RISE) (TC 5)
COND B (FALL) (TC 10))
```

specifies a total toggle count of 50. Of the 25 rise transitions, 20 occur when pin `A` has a value of `1`, and 5 occur when pin `A` has a value of `0` and `B` is `1`. Of the 25 fall transitions, 15 occur when the pin `A` is `1`, and 10 occur when the pin `A` is `0` and `B` is `1`.

The state associated with an input pin transition is the cell state just before the time of the transition, e.g., in the wave diagram given in Figure 15, the state associated with the rise transition on input pin `A` at time `10` is represented by the expression `A * !B * !Y`.

The state associated with an output pin transition is the cell state just before the time of the input pin transition causing the output pin transition, e.g., in the wave diagram given in Figure 15, the rise transition on the output pin `Y` at time `13` is caused by the rise transition on the input pin `B` at time `10`. The state associated with the rise transition on `Y` is the cell state just before time `10` (not time `13`). This state is represented by the expression `!A * B * !Y`.

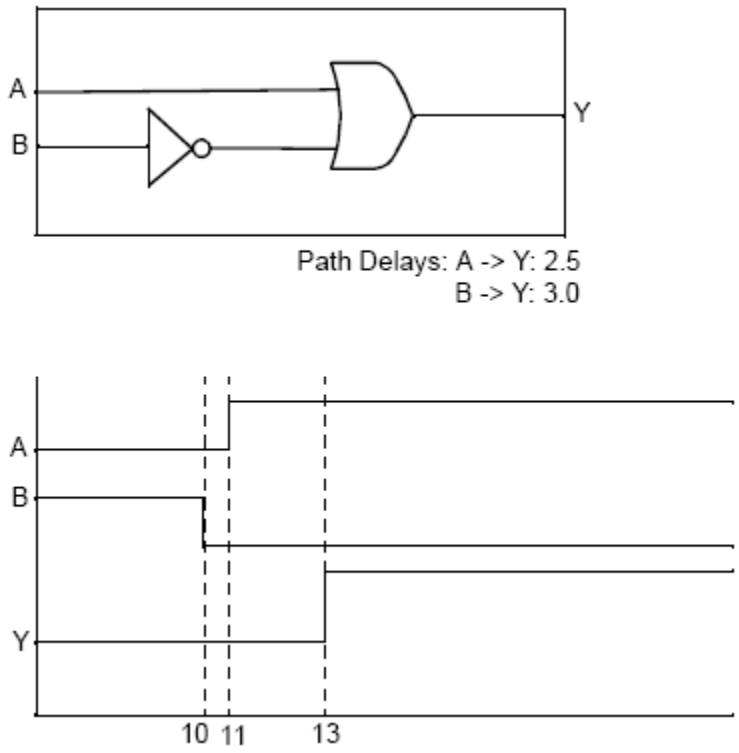

Figure 15—A cell and its wave diagram

8.3.7 Path-dependent toggle attributes

The toggle attributes on output cell pins can be *path dependent*, i.e., the attributes are relevant only to particular input pins causing the output toggles. Syntax 17 defines this construct.

```
path_dep_toggle_attributes ::=
    (path_dep_toggle_item {path_dep_toggle_item}
    [IOPATH_DEFAULT simple_toggle_attribute])
path_dep_toggle_item ::=
    IOPATH port_name {port_name} simple_toggle_attribute
```

Syntax 17—path_dep_toggle_attributes

Given a path-dependent toggle attributes construct:

```
(IOPATH pins1 attrs1
IOPATH pins2 attrs2
...
IOPATH pinsn attrsn
IOPATH_DEFAULT attrs_default)
```

the attributes `attrs1` represent toggles caused by the input pins in `pins1`, the attributes `attrs2` represent toggles caused by the input pins in `pins2`, etc. The pin lists `pins1`, `...`, `pinsn` are mutually exclusive. The attributes `attrs_default` represent toggles caused by the cell input pins not present in `pins1`, `...`, `pinsn`. The pin lists `pins1`, `...`, `pinsn` are also called the *path conditions* or *related pins*.

Example:

The following path-dependent toggle attributes construct:

```
(IOPATH A (TC 10)
IOPATH B (TC 20)
IOPATH C D (TC 5))
```

specifies a total of 35 toggle edges on a cell output port, of which 10 are caused by transitions on the input port `A`, 20 are caused by transitions on the input port `B` and 5 are caused either by a transition on the input port `C` or `D`.

8.3.8 SDPD toggle attributes

The toggle attributes on output cell pins can be both state and path dependent (*SDPD*). The syntax of state-dependent and path dependent toggle attributes is that of simple toggle attributes and path-dependent toggle attributes nested inside a state-dependent toggle attributes construct, as shown in Syntax 18.

Similarly to state-dependent toggle attributes and path-dependent toggle attributes, the SDPD toggle attributes construct represents a priority -encoded attribute specification.

```
sdpd_toggle_attributes ::=
      (sdpd_toggle_item {sdpd_toggle_item}
      [sdpd_default_toggle_item])
sdpd_toggle_item ::=
      COND cond_expr [(edge_type)] potential_pd_toggle_attributes
potential_pd_toggle_attributes ::=
       path_dep_toggle_attributes
      | simple_toggle_attribute
sdpd_default_toggle_item ::=
       COND_DEFAULT potential_pd_toggle_attributes
      | COND_DEFAULT (edge_type) potential_pd_toggle_attributes
       [COND_DEFAULT (edge_type) potential_pd_toggle_attributes]
```

Syntax 18—sdpd_toggle_attributes

Example:

This is an example of an SDPD toggle attributes construct.

```
(COND A (RISE) (IOPATH B (TC 1))
COND A (FALL) (IOPATH B (TC 2))
COND B (RISE) (IOPATH A (TC 1))
COND B (FALL) (IOPATH A (TC 0))
COND_DEFAULT (RISE) (IOPATH A (TC 1)
IOPATH B (TC 0))
COND_DEFAULT (FALL) (IOPATH A (TC 0)
IOPATH B (TC 1)))
```

8.3.9 Net, port, and leakage switching specifications

The constructs for net, port, and leakage switching specification associate switching activity (given in terms of timing and toggle attributes) to individual design nets, ports, and cells.

8.3.9.1 Net switching specifications

The net switching specification construct associates switching activity to individual nets. Syntax 19 defines the `backward_net_spec`.

```
backward_net_spec ::=
      (NET backward_net_info {backward_net_info})
backward_net_info ::=
      (net_name net_switching_attributes)
net_name ::=
      identifier
net_switching_attributes ::=
      {net_switching_attribute}
net_switching_attribute ::=
        simple_timing_attribute
      | simple_toggle_attribute
```

Syntax 19—backward_net_spec

The switching attributes that can be associated to nets are simple timing attributes and simple toggle attributes.

Example:

This is an example of a net switching specification assigning switching activity to the nets `clk`, `rst`, `in1`, `in2`, and `out`.

```
(NET
(clk (T0 100) (T1 100) (TC 50))
(rst (T0 180) (T1 20) (TC 2))
(in1 (T0 60) (T1 140) (TC 22))
(in2 (T0 80) (T1 120) (TC 12))
(out (T0 120) (T1 60) (TX 20) (TC 10))
)
```

8.3.9.2 Port switching specifications

The port switching specification construct associates switching activity to individual design ports and cell pins. Syntax 20 defines the `backward_port_spec`.

The toggle attributes that can be associated to input cell pins can be simple or state dependent. The toggle attributes that can be associated to output cell pins can be simple, state dependent, path dependent, or both state and path dependent. The toggle attributes that can be associated to design ports have to be simple. The timing attributes that can be associated to design ports and cell pins have to be simple.

Example:

This is an example of the port switching specification construct applied to an `AND` gate.

```
(PORT
(A (T0 8) (T1 7)
(COND B (RISE) (TC 1)
COND B (FALL) (TC 2)
COND_DEFAULT (TC 1)))
```

```
backward_port_spec ::=
    (PORT backward_port_info {backward_port_info})
backward_port_info ::=
    (port_name port_switching_attributes)
port_name ::=
    identifier
port_switching_attributes ::=
    {port_switching_attribute}
port_switching_attribute ::=
     simple_timing_attribute
    | simple_toggle_attribute
    | state_dep_toggle_attributes
    | path_dep_toggle_attributes
    | sdpd_toggle_attributes
```

Syntax 20—backward_port_spec

```
(B (T0 9) (T1 6)
(COND A (RISE) (TC 2)
COND A (FALL) (TC 1)
COND_DEFAULT (TC 3)))
(Y (T0 10) (T1 5)
(COND A (RISE) (IOPATH B) (TC 2)
COND A (FALL) (IOPATH B) (TC 1)
COND B (RISE) (IOPATH A) (TC 1)
COND B (FALL) (IOPATH A) (TC 2)
COND_DEFAULT (TC 0)))
)
```

8.3.9.3 Leakage switching specifications

The leakage switching specification construct specifies the duration that a particular cell spends in particular states. This construct is a list of state-dependent timing attributes, as shown in Syntax 21.

```
backward_leakage_spec ::=
    (LEAKAGE state_dep_timing_attributes {state_dep_timing_attributes})
```

Syntax 21—backward_leakage_spec

Example:

This is an example of a leakage switching specification.

```
(LEAKAGE
(COND (A * B) (T1 5) (T0 10))
COND (A | B) (T1 6) (T0 9))
(COND_DEFAULT (T1 4) (T0 11)))
)
```

8.3.10 Backward SAIF info and instance data

Design switching activity is organized hierarchically in the backward SAIF info construct (that follows the SAIF header in a backward SAIF file). The *backward SAIF info* is a list of backward instance info constructs, as shown in Syntax 22.


```
backward_saif_info ::=
      {backward_instance_info}
backward_instance_info ::=
       (INSTANCE [string] path {backward_instance_spec}  {backward_instance_info})
      | (VIRTUAL_INSTANCE string path backward_port_spec)
backward_instance_spec ::=
       backward_net_spec
      | backward_port_spec
      | backward_leakage_spec
```

Syntax 22—backward_saif_info

`backward_instance_info` contains the switching activity of a particular cell or design instance. The optional *string* following the **INSTANCE** keyword is the cell/design name that is instantiated and *path* is the hierarchical name of the actual instance. This is followed by a possibly empty list of instance switching specifications, which are the net, port, and leakage switching specifications described in 8.3.9. For design instances, the instance info can recursively contain the switching activity of its sub-design and library cell instances.

`backward_instance_info` can also be used to specify the switching activity of cell instances where the port names of the instance are not known, e.g., in design flows where switching activity generated by RTL simulation is annotated to the synthesized gate-level netlist of the RTL design.

In this case, the *string* following the **VIRTUAL_INSTANCE** keyword represents the type of cell instance; it needs to be recognized by the application reading the backward SAIF file. The *path* represents the name of the instance and `backward_port_spec` assigns switching activity to logical port names. The application reading the SAIF file needs to map the logical port names to the actual cell instance port names.

Example:

For example, the following virtual instance construct:

```
(VIRTUAL_INSTANCE "sequential" A_reg
(PORT
(Q (T0 220) (T1 370) (TC 122))
)
)
```

gives the switching activity of the positive output pin of a sequential element; the actual name of the output pin depends on the library cell that is used to implement the sequential cell, i.e., it can have a different name than `Q`.

8.4 Library forward SAIF file

The *Library forward SAIF file* contains the state-dependent and path-dependent (SDPD) directives needed by simulators and other applications generating backward SAIF files that contain state-dependent and path-dependent switching activity. The SDPD directives can be generated from cell libraries with SDPD power characterization by using the appropriate tools.

For a description of state and path dependency, see 8.3.

8.4.1 The SAIF file

The library forward SAIF file consists of a left-parenthesis ((), the **SAIFILE** keyword, the library forward SAIF header, the library forward SAIF info, and a finishing right-parenthesis ()), as shown in Syntax 23.

```
lforward_saif_file ::=
    (SAIFILE lforward_saif_header lforward_saif_info)
```

Syntax 23—lforward_saif_file

8.4.1.1 Header

Syntax 24 defines the *library forward SAIF file header*.

```
lforward_saif_header ::=
    lforward_saif_version
    direction
    design_name
    date
    vendor
    program_name
    program_version
    hierarchy_divider
```

Syntax 24—lforward_saif_header

Each library forward SAIF header construct is described in the following subclauses.

8.4.1.2 lforward_saif_version

Syntax 25 defines the `lforward_saif_version`.

```
lforward_saif_version ::=
    (SAIFVERSION string [string])
```

Syntax 25—lforward_saif_version

The first *string* in the this construct represents the version number of the SAIF file, i.e., **2.0**.

The second *string* is optional and is either the string **"lib"** or **"LIB"**; this is used to specify that the SAIF file is a library forward SAIF file.

8.4.1.3 direction

Syntax 26 defines the `direction`.

```
direction ::=
    (DIRECTION string)
```

Syntax 26—direction

The *string* in the this construct represents the type of the SAIF file, i.e., **forward**.

8.4.1.4 design_name

Syntax 27 defines the `design_name`.

```
design_name ::=
    (DESIGN [string])
```

Syntax 27—design_name

The optional *string* in this construct represents the design for which the forward SAIF file has been generated.

8.4.1.5 date

Syntax 28 defines the `date`.

```
date ::=
    (DATE [string])
```

Syntax 28—date

The optional *string* in this construct represents the date the SAIF file was generated.

8.4.1.6 vendor

Syntax 29 defines the `vendor`.

```
vendor ::=
    (VENDOR [string])
```

Syntax 29—vendor

The optional *string* in this construct represents the name of the vendor whose application was used to generate the SAIF file.

8.4.1.7 program_name

Syntax 30 defines the `program_name`.

```
program_name ::=
    (PROGRAM_NAME [string])
```

Syntax 30—program_name

The optional *string* in this construct represents the name of the application used to generate the SAIF file.

8.4.1.8 program_version

Syntax 31 defines the `program_version`.

The optional *string* in this construct represents the version number of the application used to generate the SAIF file.

```
program_version ::=
    (PROGRAM_VERSION [string])
```

Syntax 31—program_version

8.4.1.9 hierarchy_divider

Syntax 32 defines the `hierarchy_divider`.

```
hierarchy_divider ::=
    (DIVIDER [hchar])
```

Syntax 32—hierarchy_divider

The optional *hchar* in this construct represents the hierarchical separator character used in hierarchical identifiers. Only the **/** and **.** characters shall be specified as the hierarchical separator character; the default is the **.** character.

Example:

This is an example of a valid library forward SAIF file header.

```
(SAIFVERSION "2.0" "lib")
(DIRECTION "forward")
(DESIGN)
(DATE "Fri Jan 18 10:00:00 PDT 2002")
(VENDOR "SAIFíRíUS Corp.")
(PROGRAM_NAME "libsaifgenerator")
(PROGRAM_VERSION "1.0")
(DIVIDER /)
```

8.4.2 State-dependent timing directive

State-dependent timing directives instruct the backward SAIF generator on the state conditions required in state-dependent timing attributes. Syntax 33 defines the `state_dep_timing_directive`.

```
state_dep_timing_directive ::=
    (state_dep_timing_directive_item
    {state_dep_timing_directive_item}
    [COND_DEFAULT])
state_dep_timing_directive_item ::=
    COND cond_expr
```

Syntax 33—state_dep_timing_directive

A state-dependent timing directive is a list of directive items. The state-dependent timing attributes generated using such a timing directive contain switching activity assigned to a number of the states given in the directive. The order of any states in the timing attribute shall be the same as that in the timing directive.

Example:

This is an example of a state-dependent timing directive.

```
(COND (A * B * C)
```

```
COND (!A * B * C)
COND (A * !(B * C))
COND B
COND C
COND_DEFAULT)
```

8.4.3 State-dependent toggle directive

State-dependent toggle directives instruct the backward SAIF generator on the state and rise/fall conditions required in state-dependent toggle attributes. Syntax 34 defines the `state_dep_toggle_directive`.

```
state_dep_toggle_directive ::=
    (state_dep_toggle_directive_item
    {state_dep_toggle_directive_item}
    [COND_DEFAULT [RISE_FALL]])
state_dep_toggle_directive_item ::=
    COND cond_expr [RISE_FALL]
```

Syntax 34—state_dep_toggle_directive

A state-dependent toggle directive is a list of directive items, each followed by an optional **RISE_FALL** keyword. The item list is followed by an optional **COND_DEFAULT** keyword, which can also be followed by an optional **RISE_FALL** keyword.

The state-dependent toggle attributes generated using such a toggle directive contain switching activity for a number of the states given in the directive. The order of any states in the toggle attribute shall be the same as that in the toggle directive. The **RISE_FALL** keyword instructs the backward SAIF generator that rise and fall edges can be differentiated and state-dependent toggle attribute items with **RISE** and/or **FALL** keywords can be generated.

Example:

This is an example of a state-dependent toggle directive construct.

```
(COND (A*B) RISE_FALL
COND A RISE_FALL
COND B RISE_FALL
COND_DEFAULT)
```

8.4.4 Path-dependent toggle directive

Path-dependent toggle directives instruct the backward SAIF generator on the path conditions required in path-dependent toggle attributes for cell output pins. A *path condition* is a list of input port pins. Syntax 35 defines the `path_dep_toggle_directive`.

```
path_dep_toggle_directive ::=
    (path_dep_toggle_directive_item
    {path_dep_toggle_directive_item}
    [IOPATH_DEFAULT])
path_dep_toggle_directive_item ::=
    IOPATH port_name {port_name}
```

Syntax 35—path_dep_toggle_directive

A path-dependent toggle directive is a list of directive items. The path-dependent toggle attributes generated using such a toggle directive contain switching activity for a number of the path conditions (input pin lists) given in the directive. The order of the path conditions in the toggle attribute shall be the same as that in the toggle directive.

Example:

This is an example of a path-dependent toggle directive construct.

```
(IOPATH A
IOPATH B
IOPATH C D)
```

8.4.5 SDPD toggle directives

SDPD toggle directives instruct the backward SAIF generator on the state and path conditions required in SDPD toggle attributes for cell output pins. The syntax of this construct is that of the path-dependent toggle directive embedded in the state-dependent toggle directive, as shown in Syntax 36.

```
sdpd_toggle_directive ::=
    (sdpd_toggle_directive_item {sdpd_toggle_directive_item}
    [COND_DEFAULT [RISE_FALL] [path_dep_toggle_directive]])
sdpd_toggle_directive_item ::=
    COND cond_expr [RISE_FALL] [path_dep_toggle_directive]
```

Syntax 36—sdpd_toggle_directive

The SDPD dependent toggle attributes generated using such a toggle directive contain switching activity for a number of the state and path conditions given in the directive. The order of the conditions in the toggle attribute shall be the same as that in the toggle directive.

Example:

This is an example of an SDPD toggle directive construct.

```
(COND A RISE_FALL (IOPATH B)
COND B RISE_FALL (IOPATH A)
COND_DEFAULT RISE_FALL
(IOPATH A
IOPATH B
IOPATH_DEFAULT))
```

8.4.6 Module SDPD declarations

Module SDPD declarations instruct the backward SAIF generator on the type and structure of the required switching activity for particular cells. Syntax 37 defines this construct.

The module name *identifier* represents the library cell name.

A *port declaration* assigns port directives to the individual cell pins. Port directives are either state-dependent toggle directives, path-dependent toggle directives, or SDPD toggle directives.

A *leakage declaration* consists of the **LEAKAGE** keyword followed by a state-dependent timing directive, which instructs the backward SAIF generator on the state conditions for the state-dependent timing attributes in backward leakage specifications.


```
module_sdpd_declaration ::=
      (MODULE module_name {module_sdpd_directive})
module_name ::=
      identifier
module_sdpd_directive ::=
       port_declaration
      | leakage_declaration
port_declaration ::=
      (PORT port_name {port_directive})
port_directive ::=
       state_dep_toggle_directive
      | path_dep_toggle_directive
      | sdpd_toggle_directive
leakage_declaration ::=
      (LEAKAGE {state_dep_timing_directive})
```

Syntax 37—module_sdpd_declaration

Examples:

This is an example of a port declaration.

```
(PORT
(A
(COND B RISE_FALL
COND_DEFAULT))


(B
(COND A RISE_FALL
COND_DEFAULT))
(Y
(COND A RISE_FALL (IOPATH B)
COND B RISE_FALL (IOPATH A)
COND_DEFAULT))
)
```

This is an example of a leakage declaration.

```
(LEAKAGE
(COND (A * B)
COND (A | B)
COND_DEFAULT)
)
```

8.4.7 Library SDPD information

The *SDPD declarations* for each library cell are listed in the library SDPD info constructs (that follow the SAIF header in the library forward SAIF file). Syntax 38 defines the `library_sdpd_info`.

```
library_sdpd_info ::=
      (LIBRARY string [string]
      {module_sdpd_declaration})
```

Syntax 38—library_sdpd_info

The first *string* following the **LIBRARY** keyword represents the name of the library. The second (optional) *string* sets the path of the directory containing the library and can be used for locating it.

8.5 The RTL forward SAIF file

The *RTF forward SAIF file* lists the synthesis invariant points of an RTL design and provides a mapping from the RTL identifiers of these design objects to their synthesized gate-level identifiers. *Synthesis invariant points* are design objects (nets, ports, etc.) in the RTL description that are mapped directly to equivalent design objects in the synthesized gate-level descriptions. Examples of such points are the design ports and RTL identifiers (variables, signals, wires, etc.) that are mapped to the outputs of sequential cells.

8.5.1 The SAIF file

The RTF forward SAIF file consists of a left-parenthesis ((), the **SAIFILE** keyword, the RTL forward SAIF header, the RTL forward SAIF info, and a finishing right-parenthesis ()), as shown in Syntax 39.

```
rforward_saif_file ::=
    (SAIFILE rforward_saif_header rforward_saif_info)
```

Syntax 39—rforward_saif_file

8.5.1.1 Header

Syntax 40 defines the *RTL forward SAIF file header*.

```
rforward_saif_header ::=
    rforward_saif_version
    direction
    design_name
    date
    vendor
    program_name
    program_version
    hierarchy_divider
```

Syntax 40—rforward_saif_header

Each RTL forward SAIF header construct is described in the following subclauses.

8.5.1.2 rforward_saif_version

Syntax 41 defines the `rforward_saif_version`.

```
rforward_saif_version ::=
    (SAIFVERSION string)
```

Syntax 41—rforward_saif_version

The *string* in the this construct represents the version number of the SAIF file, i.e., **2.0**.

8.5.1.3 direction

Syntax 42 defines the `direction`.

```
direction ::=
    (DIRECTION string)
```

Syntax 42—direction

The *string* in the this construct represents the type of the SAIF file, i.e., **forward**.

8.5.1.4 design_name

Syntax 43 defines the `design_name`.

```
design_name ::=
    (DESIGN [string])
```

Syntax 43—design_name

The optional *string* in this construct represents the design for which the forward SAIF file has been generated.

8.5.1.5 date

Syntax 44 defines the `date`.

```
date ::=
    (DATE [string])
```

Syntax 44—date

The optional *string* in this construct represents the date the SAIF file was generated.

8.5.1.6 vendor

Syntax 45 defines the `vendor`.

```
vendor ::=
    (VENDOR [string])
```

Syntax 45—vendor

The optional *string* in this construct represents the name of the vendor whose application was used to generate the SAIF file.

8.5.1.7 program_name

Syntax 46 defines the `program_name`.

```
program_name ::=
    (PROGRAM_NAME [string])
```

Syntax 46—program_name

The optional *string* in this construct represents the name of the application used to generate the SAIF file.

8.5.1.8 program_version

Syntax 47 defines the `program_version`.

```
program_version ::=
    (PROGRAM_VERSION [string])
```

Syntax 47—program_version

The optional *string* in this construct represents the version number of the application used to generate the SAIF file.

8.5.1.9 hierarchy_divider

Syntax 48 defines the `hierarchy_divider`.

```
hierarchy_divider ::=
    (DIVIDER [hchar])
```

Syntax 48—hierarchy_divider

The optional *hchar* in this construct represents the hierarchical separator character used in hierarchical identifiers. Only the **/** and **.** characters shall be specified as the hierarchical separator character; the default is the **.** character.

Example:

The following is an example of a valid library forward SAIF file header:

```
(SAIFVERSION "2.0")
(DIRECTION "forward")
(DESIGN "alu")
(DATE "Fri Jan 18 11:00:00 PDT 2002")
(VENDOR "SAIFíRíUS Corp.")
(PROGRAM_NAME "rtlsaifgenerator")
(PROGRAM_VERSION "1.0")
(DIVIDER /)
```

8.5.2 Port and net mapping directives

The *port and net mapping directives* in the RTL forward SAIF file contain a list of synthesis invariant port and net identifiers and their corresponding synthesized gate-level identifiers. Syntax 49 defines these constructs.

Here, the `rtl_name` is mapped into the gate-level identifier `mapped_name`. Both the RTL name and mapped name in these constructs are represented by hierarchical identifiers.

In `port_mapping_directives`, the optional *string* is used for generating virtual instance data in the backward SAIF file and represents the type of the virtual instance.


```
port_mapping_directives ::=
      (PORT {(rtl_name mapped_name [string])})
rtl_name ::=
      hierarchical_identifier
mapped_name ::=
      hierarchical_identifier
net_mapping_directives ::=
      (NET {(rtl_name mapped_name)})
```

Syntax 49—Port and net mapping directives

8.5.3 Instance declarations

The port and net mapping directives in the RTL forward SAIF file are organized hierarchically in RTL forward instance declarations, which comprise the RTL forward SAIF instance info that follows the header in the forward SAIF file. Syntax 50 defines the RTL forward SAIF info constructs.

```
rforward_saif_info ::=
      {rforward_instance_declaration}
rforward_instance_declaration ::=
      (INSTANCE [string] instance_name {rforward_instance_directive}
      {rforward_instance_declaration})
instance_name ::=
      hierarchical_identifier
rforward_instance_directive ::=
       port_mapping_directives
      | net_mapping_directives
```

Syntax 50—RTL forward SAIF info constructs

The *RTL forward SAIF info* is a list of instance declarations. The optional *string* following the **INSTANCE** keyword represents the design name and the *hierarchical_identifier* following it is the actual instance name. The port and net mapping directives follow the instance name. The instance declarations of any sub-design instances can be included recursively in this construct.

Annex A

(informative)

Bibliography

[B1] IEEE 100, *The Authoritative Dictionary of IEEE Standards Terms,* Seventh Edition. New York: Institute of Electrical and Electronics Engineers, Inc.

[B2] IEEE Std 1364™, IEEE Standard for Verilog Hardware Description Language.[7]

[B3] ISO/IEC 8859-1, Information technology—8-bit single-byte coded graphic character sets—Part 1: Latin Alphabet No. 1.[8]

[B4] For a summary of Tcl language syntax, see the following Internet location: http://www.tcl.tk/man/tcl8.4/TclCmd.

[B5] For more details on using the Tcl language, see the following Internet location: http://sourceforge.net/projects/tcl/.

[B6] For more details on using the Liberty library format, see the following Internet location: http://synopsys.com/cgi-bin/tapin/login1.cgi.

[B7] Coding examples are available from the UPF *WG* World Wide Web site http://www.accellera.org/upf/references.html.

[7]The IEEE standards or products referred to in this clause are trademarks of the Institute of Electrical and Electronics Engineers, Inc.

[8]ISO/IEC publications are available from the ISO Central Secretariat, Case Postale 56, 1 rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse (http://www.iso.ch/). ISO/IEC publications are also available in the United States from Global Engineering Documents, 15 Inverness Way East, Englewood, CO 80112, USA (http://global.ihs.com/). Electronic copies are available in the United States from the American National Standards Institute, 25 West 43rd Street, 4th Floor, New York, NY 10036, USA (http://www.ansi.org/).

Annex B

(normative)

Supply net logic type

These functions are required for any implementations supporting VHDL and/or SystemVerilog simulation.

The `real` typed value parameter to the **supply_on** and **supply_partial_on** functions is the voltage value in units of volts. This voltage value shall be converted into a signed 32-bit integer value in units of microvolts.

B.1 VHDL

The following defines the VHDL package for UPF. This package shall be located in the IEEE library.

```
Library IEEE;
Use IEEE.std_logic_1164.all;
Use IEEE.numeric_bit.all;
package UPF is

  type state is (OFF,
                 UNDETERMINED,
                 PARTIAL_ON,
                 FULL_ON);

  -- The provided routines shall be used to ensure
  -- the HDL code is independent of the details of the supply net
  -- type implementation.  This ensures portability and forward
  -- compatibility of the HDL.
  -- The supply net type implementation is openly specified for
  -- the following reasons:
  --   1.  Users know how supply net and port values will visually
  --       appear in tools such as wave windows.
  --   2.  C language access by user or 3rd party tools can depend
  --       on existing functionality to read and write supply
  --       values.
  --
  -- Tools implementing this package may optimize the supply data
  -- type as long as the 2 items above are preserved and the
  -- supply value set and get routines are supported.
  type supply_net_type is record
    state   : state;  -- UPF reserves 32 bits
    -- Voltage in microvolts
    voltage : signed(31 downto 0);
  end record;

  -- Types used to navigate and to find UPF objects in
  -- the design hierarchy
  subtype upf_object_handle is Integer;

  type object_kind is (ERROR_KIND,
                       SWITCH, ISOLATION_CELL, LEVEL_SHIFTER,
                       SUPPLY_SET, SUPPLY_NET, SUPPLY_PORT,
                       ROOT_SUPPLY_DRIVER,
                       LOGIC_NET, LOGIC_PORT,
```

```
                    INSTANCE,
                    POWER_DOMAIN,
                    UPF_POWER_STATE,
                    ITERATOR,
                    OTHER );

-- NOTE: UNDETERMINED is not defined as a power state kind as
--       it is replaced during simulation with a determined state
type power_state_kind is
  (ERROR_PS, OPERATING, ILLEGAL, TRANSIENT);

type power_state_simstate is
  (NORMAL, CORRUPT, CORRUPT_ON_ACTIVITY, CORRUPT_STATE_ON_CHANGE,
   CORRUPT_STATE_ON_ACTIVITY);

subtype supply_kind is object_kind
  range SUPPLY_NET to ROOT_SUPPLY_DRIVER;

-- Voltage is a real value in volts that is converted into
-- an integer value normalized to microvolts
function supply_on (
  supply_name : STRING;   -- Path name to supply net, port or
                          -- root supply driver
  voltage     : REAL := 1.0 )
return BOOLEAN;

function supply_off (
  supply_name : STRING )
return BOOLEAN;

-- Voltage is a real value in volts that is converted into
-- an integer value normalized to microvolts
function supply_partial_on (
  supply_name : STRING;
  value       : REAL := 1.0 )
return BOOLEAN;

function get_supply_value (
  supply_name : STRING )
return supply_net_type;

function get_supply_voltage (
  value : supply_net_type )
return REAL;

function get_supply_on_state (
  value : supply_net_type )
return BOOLEAN;

function get_supply_on_state (
  value : supply_net_type )
return BIT;

function get_supply_state (
  value : supply_net_type )
return state;

-- Routines to navigate and find UPF objects in the design hierarchy
```



```
-- The initial scope shall be the root of the simulation
-- which allows access to the testbench as well as design
-- under verification.
-- If inst_path is valid for the active scope, then
-- the function changes the scope to that instance.
-- The function returns TRUE on success, FALSE if the
-- the scope cannot be set as requested.
function set_scope( inst_path : STRING )
return Boolean;

-- This function returns the active scope's complete
-- instance path from the root of the simulation.
function get_scope
return STRING;

-- Tests the handle and returns TRUE if the handle is valid
-- and FALSE if it is invalid
function is_valid_handle( handle : in upf_object_handle )
return Boolean;

-- Get a handle to a design object (either HDL or UPF
-- created).  If a qualifier is required to differentiate
-- objects in different name spaces at the same scope, then
-- the qualifier string shall not be null.  See section 7.x
-- for a list of valid qualifier strings.
-- Returns a valid handle on success; invalid handle on failure
function get_object( inst_path : STRING;
                     qualifier : STRING := "" )
return upf_object_handle;

-- Returns the kind of object that the handle refers
-- to.
-- If the handle is not valid, ERROR object kind is
-- returned.
function get_object_kind( handle : upf_object_handle )
return object_kind;

-- Returns TRUE if the kind of object referenced by
-- handle is a supply_net, supply_port or
-- root_supply_driver.
-- Returns FALSE otherwise.
function is_supply_kind ( handle : upf_object_handle )
return Boolean;

-- For a supply kind of object referenced by handle,
-- return the state of that object.
-- It is the caller's responsibility to ensure that
-- the handle passed references a supply kind of object.
-- If the object is not a supply kind, the value returned
-- is UNDETERMINED
function get_supply_state( handle : upf_object_handle )
return state;

-- For a supply kind of object referenced by handle,
-- return the voltage of that object.
-- It is the caller's responsibility to ensure that
-- the handle passed references a supply kind of object.
-- If the object is not a supply kind, the value returned
-- is -1.0.
```

```
function get_supply_voltage( handle : upf_object_handle )
return REAL;

-- For a handle that references a supply kind object, sets
-- the net state and voltage of the supply.
-- Returns TRUE on success.
-- Returns FALSE if the supply state cannot be set or
-- if the object that handle references is not a supply
-- kind of object.
function assign_supply_state( handle  : upf_object_handle;
                              state   : state := OFF;
                              voltage : REAL := 0.0,
                              after   : TIME := 0 ns )
return Boolean;

-- Quick checks for the information specified by the
-- function name.
-- All functions return TRUE if the information/state
-- specified is true for the object referenced by handle.
-- Returns FALSE if it is not true or if the information/
-- state being compared or check is not applicable to the
-- kind of object that handle references.
function is_supply_full_on ( handle : upf_object_handle )
return Boolean;

function is_supply_off ( handle : upf_object_handle )
return Boolean;

function is_supply_partial_on ( handle : upf_object_handle )
return Boolean;

function is_supply_undetermined ( handle : upf_object_handle )
return Boolean;

function is_supply_equal ( handle  : upf_object_handle;
                           state   : state;
                           voltage : real )
return Boolean;

-- Both handles must reference a supply kind of object.
-- Returns TRUE if states are the same and, if
-- state is not OFF, the voltages are the same.
-- This function does not check the root supply drivers of
-- the supplies or any other connectivity aspects of the supplies
function are_supplies_equivalent ( handle1 : upf_object_handle;
                                   handle2 : upf_object_handle )
return Boolean;

-- Assigns the source supply to the destination supply.
-- For purposes of supply net resolution, the destination
-- will be sourced by the same root supply driver as the source.
-- (The source may be a root supply driver.)
-- Returns TRUE on success, FALSE on failure.
function assign_supply2supply( destination : upf_object_handle;
                               source      : upf_object_handle;
                               after       : TIME := 0 ns )
return Boolean;

-- Creates a root supply driver than can be used to drive
```



```
-- one or more supply nets from within an HDL model of a supply
-- network component (HDL model of a bias generator, for example).
-- The root supply driver is created within the scope of the parent.
-- The parent and driver name information may be used for error reporting.
-- Returns a valid object handle on success and an invalid object handle
-- on failure.
function create_root_supply_driver (
  driver_name : STRING;
  parent      : upf_object_handle )
return upf_object_handle;

-- Routines to query and set power states on various objects.

-- There can be 0, 1 or many power states defined for a given
-- object.  The iterator provides a mechanism to retrieve a
-- an opaque list handled by the tool.
-- If there are 0 power states, then the handle returned is
-- an invalid handle.
function get_iterator_for_all_ps ( handle : upf_object_handle )
return upf_object_handle;

-- Returns an iterator referencing all power states of the
-- specified object that are active when the call is made.
-- The returned handle is invalid if there are no power states
-- defined for the specified handle or if none of the power
-- states defined are active.
function get_iterator_for_all_active_ps (
  handle : upf_object_handle )
return upf_object_handle;

-- If there are more items in the iterator, this routine
-- will return the next item in the iterator.
-- Otherwise, an invalid handle will be returned if there
-- are no more objects to iterate over or if the iterator is
-- invalid
function iterate( iterator : upf_object_handle )
return upf_object_handle;

-- Returns the name of a power state kind of object.
-- Returns the null string if the handle does not reference
-- a power state object.
function get_ps_name( power_state : upf_object_handle )
return STRING;

-- For a handle referencing a power state object,
-- return the kind of power state.
-- Returns ERROR if the handle is invalid or does
-- not reference a power state object
function get_ps_kind( power_state : upf_object_handle )
return power_state_kind;

-- For a handle referencing a power state object,
-- return the simulation state associated with the power state.
function get_ps_simstate( power_state : upf_object_handle )
return power_state_simstate;

-- Returns TRUE if the object to which this power state is
-- attributed is in a state consistent with being in this
-- power state.
```

```
-- Returns FALSE otherwise (including if the power_state
-- handle is invalid)
function is_active( power_state : upf_object_handle )
return Boolean;

-- Returns TRUE if the object referenced by handle
-- is in the power state referenced by the power state
-- handle.  If either handle is invalid, it returns FALSE.
function is_in ( handle      : upf_object_handle;
                 power_state : upf_object_handle )
return Boolean;

-- Set the object to the specified power state.
-- This function returns TRUE on success.
-- It returns FALSE on failure.
-- The function will fail if any aspect of the power state definition
-- that is fully implemented is not in the state required to match
-- the specified power state.
-- For example, if the power state is defined in terms of only
-- symbolic supply sets, domain names (implied primary supply set of
-- the domain) and isolation and retention strategy names (implied
-- isolation and retention supply sets) and no supply nets have
-- been created and associated with those supply sets, then this
-- function can be freely used to change the symbolic power state
-- of the object.
-- If however, the power state is also defined in terms of:
--   1. The state of a logic signal or port (e.g., for use to control
--      a power switch), then that logic signal/port must already
--      have the value required by the power state.  That is, a check
--      is made to ensure the logic net has the required value.  The
--      logic net or port is not changed as a result of this command.
--   2. The state of a supply port or net.  Then that supply must
--      already have the required net state as defined by the power
--      state.  A power state at this level of specification
--      indicates that the low power design has progressed to the
--      point of explicitly creating the supply net or port.
--      Therefore, the state of such explicitly created supply nets
--      and ports must be set outside of this command
function set_power_state( object      : upf_object_handle;
                          power_state : upf_object_handle;
                          after       : TIME := 0 ns )
return Boolean;

-- Routines to facilitate type conversion of a supply net state to a
-- logic value; specifically, for use in connecting a supply net to a
-- logic port that is tied high or tied low.

-- Returns 1 if the supply net is ON at any voltage level > 0.0.
-- Returns X if the supply net is OFF or PARTIAL_ON.
-- It is up to the user to ensure that a proper supply net is
-- connected to a power net.
function tie_hi ( supply_net : supply_net_type )
return std_logic;

-- Returns 0 if the supply net is OFF.
-- Returns X if the supply net is ON or PARTIAL_ON.
-- It is up to the user to ensure that a proper supply net is
-- connected to a ground net.
function tie_lo ( supply_net : supply_net_type )
```



```
  return std_logic;
end package UPF;
```

B.2 SystemVerilog

The following defines the SystemVerilog package for UPF.

```
package UPF;

  // Bit encoding of the state type is provided
  // for backward compatibility to UPF 1.0.
  typedef enum {OFF = 0,
                UNDETERMINED,
                PARTIAL_ON,
                FULL_ON} state;

  // The provided routines shall be used to ensure
  // the HDL code is independent of the details of the supply net
  // type implementation.  This ensures portability and forward
  // compatibility of the HDL.
  // The supply net type implementation is openly specified for
  // the following reasons:
  //   1.  Users know how supply net and port values will visually
  //       appear in tools such as wave windows.
  //   2.  C language access by user or 3rd party tools can depend
  //       on existing functionality to read and write supply
  //       values.
  //
  // Tools implementing this package may optimize the supply data
  // type as long as the 2 items above are preserved and the
  // supply value set and get routines are supported.
  typedef struct packed {
    state state;   // UPF reserves 32 bits
    int       voltage; // voltage in ?V
  } supply_net_type;


  // Types used to navigate and to find UPF objects in
  // the design hierarchy
  typedef chandle upf_object_handle;

  typedef enum {ERROR_KIND,
                SWITCH, ISOLATION_CELL, LEVEL_SHIFTER,
                SUPPLY_SET, SUPPLY_NET, SUPPLY_PORT,
                ROOT_SUPPLY_DRIVER,
                LOGIC_NET, LOGIC_PORT,
                INSTANCE,
                POWER_DOMAIN,
                UPF_POWER_STATE,
                ITERATOR,
                OTHER } object_kind;

  // NOTE:  UNDETERMINED is not defined as a power state kind as
  //        it is replaced during simulation with a determined state
  typedef enum
    {ERROR_PS, OPERATING, ILLEGAL, TRANSIENT} power_state_kind;
```

```
typedef enum
  {NORMAL, CORRUPT, CORRUPT_ON_ACTIVITY, CORRUPT_STATE_ON_CHANGE,
   CORRUPT_STATE_ON_ACTIVITY} power_state_simstate;

// SystemVerilog does not support subtype definitions
// Therefore, there is no equivalent to the VHDL subtype
// definition of supply_kind.

// Voltage is a real value in volts that is converted into
// an integer value normalized to microvolts
// SystemVerilog does not support function overloading by
// input parameter type.  Therefore, a 2nd version of functions
// is specified.
function bit supply_on( string pad_name, real value = 1.0);
endfunction
function bit supply_on_from_handle(
  upf_object_handle supply, real value = 1.0);
endfunction

function bit supply_off( string pad_name );
endfunction

function bit supply_partial_on( string pad_name, real value = 1.0 );
endfunction

function supply_net_type get_supply_value( string name );
endfunction
function supply_net_type get_supply_value_from_handle(
  upf_object_handle supply );
endfunction

function real get_supply_voltage( supply_net_type arg );
endfunction

function bit get_supply_on_state( supply_net_type arg );
endfunction

function state get_supply_state( supply_net_type arg );
endfunction

// Routines to navigate and find UPF objects in the design
// hierarchy

// The initial scope shall be the root of the simulation
// which allows access to the testbench as well as design
// under verification.
// If inst_path is valid for the active scope, then
// the function changes the scope to that instance.
// The function returns TRUE on success, FALSE if the
// the scope cannot be set as requested.
function bit set_scope( string inst_path );
endfunction

// This function returns the active scope's complete
// instance path from the root of the simulation.
function string get_scope( );
endfunction

// Tests the handle and returns TRUE if the handle is valid
```



```
// and FALSE if it is invalid
function bit is_valid_handle( upf_object_handle handle );
endfunction

// Get a handle to a design object (either HDL or UPF
// created).  If a qualifier is required to differentiate
// objects in different name spaces at the same scope, then
// the qualifier string shall not be null.  See section 7.x
// for a list of valid qualifier strings.
// Returns a valid handle on success; invalid handle on failure
function upf_object_handle get_object(
  string inst_path, string qualifier = "" );
endfunction

// Returns the kind of object that the handle refers
// to.
// If the handle is not valid, ERROR object kind is
// returned.
function object_kind get_object_kind( upf_object_handle handle );
endfunction

// Returns TRUE if the kind of object referenced by
// handle is a supply_net, supply_port or
// root_supply_driver.
// Returns FALSE otherwise.
function bit is_supply_kind ( upf_object_handle handle );
endfunction

// For a supply kind of object referenced by handle,
// return the state of that object.
// It is the caller's responsibility to ensure that
// the handle passed references a supply kind of object.
// If the object is not a supply kind, the value returned
// is UNDETERMINED
function state get_supply_state_from_handle(
  upf_object_handle handle );
endfunction

// For a supply kind of object referenced by handle,
// return the voltage of that object.
// It is the caller's responsibility to ensure that
// the handle passed references a supply kind of object.
// If the object is not a supply kind, the value returned
// is -1.0.
function real get_supply_voltage_from_handle( upf_object_handle handle );
endfunction

// For a handle that references a supply kind object, sets
// the net state and voltage of the supply.
// Returns TRUE on success.
// Returns FALSE if the supply state cannot be set or
// if the object that handle references is not a supply
// kind of object.
function bit assign_supply_state(
  upf_object_handle handle,
  state state = OFF,
  real voltage = 0.0,
  time after := 0ns );
endfunction
```

```
// Quick checks for the information specified by the function name.
// All functions return TRUE if the information/state
// specified is true for the object referenced by handle.
// Returns FALSE if it is not true or if the information/
// state being compared or check is not applicable to the
// kind of object that handle references.
function bit is_supply_full_on ( upf_object_handle handle  );
endfunction

function bit is_supply_off ( upf_object_handle handle );
endfunction

function bit is_supply_partial_on ( upf_object_handle handle );
endfunction

function bit is_supply_undetermined ( upf_object_handle handle );
endfunction

function bit is_supply_equal (
  upf_object_handle handle,
  state         state,
  real              voltage );
endfunction

// Both handles must reference a supply kind of object.
// Returns TRUE if states are the same and, if
// state is not OFF, the voltages are the same.
// This function does not check the root supply drivers of
// the supplies or any other connectivity aspects of the supplies
function bit are_supplies_equivalent (
  upf_object_handle handle1,
  upf_object_handle handle2 );
endfunction

// Assigns the source supply to the destination supply.
// For purposes of supply net resolution, the destination
// will be sourced by the same root supply driver as the source.
// (The source may be a root supply driver.)
// Returns TRUE on success, FALSE on failure.
function bit assign_supply2supply(
  upf_object_handle destination,
  upf_object_handle source,
  time              after := 0ns );
endfunction

// Creates a root supply driver than can be used to drive
// one or more supply nets from within an HDL model of a supply
// network component (HDL model of a bias generator, for example).
// The root supply driver is created within the scope of the parent.
// The parent and driver name information may be used for error
// reporting.
// Returns a valid object handle on success and an invalid object
// handle on failure.
function upf_object_handle create_root_supply_driver (
  string            driver_name,
  upf_object_handle parent );
endfunction
```



```
// Routines to query and set power states on various objects.
// There can be 0, 1 or many power states defined for a given
// object.  The iterator provides a mechanism to retrieve a
// an opaque list handled by the tool.
// If there are 0 power states, then the handle returned is
// an invalid handle.
function upf_object_handle get_iterator_for_all_ps (
  upf_object_handle handle );
endfunction

// Returns an iterator referencing all power states of the
// specified object that are active when the call is made.
// The returned handle is invalid if there are no power states
// defined for the specified handle or if none of the power
// states defined are active.
function upf_object_handle get_iterator_for_all_active_ps (
  upf_object_handle handle );
endfunction

// If there are more items in the iterator, this routine
// will return the next item in the iterator.
// Otherwise, an invalid handle will be returned if there
// are no more objects to iterate over or if the iterator is
// invalid
function upf_object_handle iterate( upf_object_handle iterator );
endfunction

// Returns the name of a power state kind of object.
// Returns the null string if the handle does not reference
// a power state object.
function string get_ps_name( upf_object_handle power_state );
endfunction

// For a handle referencing a power state object,
// return the kind of power state.
// Returns ERROR if the handle is invalid or does
// not reference a power state object
function power_state_kind get_ps_kind(
  upf_object_handle power_state );
endfunction

// For a handle referencing a power state object,
// return the simulation state associated with the power state.
function power_state_simstate get_ps_simstate(
  upf_object_handle power_state );
endfunction

// Returns TRUE if the object to which this power state is
// attributed is in a state consistent with being in this
// power state.
// Returns FALSE otherwise (including if the power_state
// handle is invalid)
function bit is_active( upf_object_handle power_state );
endfunction

// Returns TRUE if the object referenced by handle
// is in the power state referenced by the power state
// handle.  If either handle is invalid, it returns FALSE.
```

```
function bit is_in (
    upf_object_handle handle,
    upf_object_handle power_state );
  endfunction

  // Set the object to the specified power state.
  // This function returns TRUE on success.
  // It returns FALSE on failure.
  // The function will fail if any aspect of the power state definition
  // that is fully implemented is not in the state required to match
  // the specified power state.
  // For example, if the power state is defined in terms of only
  // symbolic supply sets, domain names (implied primary supply set of
  // the domain) and isolation and retention strategy names (implied
  // isolation and retention supply sets) and no supply nets have
  // been created and associated with those supply sets, then this
  // function can be freely used to change the symbolic power state
  // of the object.
  // If however, the power state is also defined in terms of:
  //   1. The state of a logic signal or port (e.g., for use to control
  //      a power switch), then that logic signal/port must already
  //      have the value required by the power state.  That is, a check
  //      is made to ensure the logic net has the required value.  The
  //      logic net or port is not changed as a result of this command.
  //   2. The state of a supply port or net.  Then that supply must
  //      already have the required net state as defined by the power
  //      state.  A power state at this level of specification
  //      indicates that the low power design has progressed to the
  //      point of explicitly creating the supply net or port.
  //      Therefore, thestate of such explicitly created supply nets
  //      and ports must be set outside of this command
  function bit set_power_state(
    upf_object_handle object,
    upf_object_handle power_state,
    time              after = 0ns  );
  endfunction

  // Routines to facilitate type conversion of a supply net state to a
  // logic value; specifically, for use in connecting a supply net to a
  // logic port that is tied high or tied low.

  // Returns 1 if the supply net is ON at any voltage level > 0.0.
  // Returns X if the supply net is OFF or PARTIAL_ON.
  // It is up to the user to ensure that a proper supply net is
  // connected to a power net.
  function logic tie_hi ( supply_net_type supply_net );
  endfunction

  // Returns 0 if the supply net is OFF.
  // Returns X if the supply net is ON or PARTIAL_ON.
  // It is up to the user to ensure that a proper supply net is
  // connected to a ground net.
  function logic tie_lo ( supply_net_type supply_net );
  endfunction

endpackage : UPF
```


Annex C

(normative)

Value conversion tables (VCTs)

The predefined value conversion tables (VCTs) are as follows:

C.1 VHDL_SL2UPF

```
create_hdl2upf_vct VHDL_SL2UPF
-hdl_type vhdl
-table { {'U' UNDETERMINED}
              {'X' UNDETERMINED}
              {'0' OFF}
              {'1' FULL_ON}
              {'Z' UNDETERMINED}
              {'L' OFF}
              {'H' FULL_ON}
              {'W' UNDETERMINED}
              {'-' UNDETERMINED}}
```

C.2 UPF2VHDL_SL

```
create_upf2hdl_vct UPF2VHDL_SL
-hdl_type vhdl
-table {{UNDETERMINED 'X'}
             {PARTIAL_ON 'X'}
             {FULL_ON '1'}
             {OFF '0'}}
```

C.3 VHDL_SL2UPF_GNDZERO

```
create_hdl2upf_vct VHDL_SL2UPF_GNDZERO
-hdl_type vhdl
-table { {'U' UNDETERMINED}
              {'X' UNDETERMINED}
              {'0' FULL_ON}
              {'1' OFF}
              {'Z' UNDETERMINED}
              {'L' FULL_ON}
              {'H' OFF}
              {'W' UNDETERMINED}
              {'-' UNDETERMINED}}
```

C.4 UPF_GNDZERO2VHDL_SL

```
create_upf2hdl_vct UPF_GNDZERO2VHDL_SL
-hdl_type vhdl
```

```
-table {{UNDETERMINED 'X'}
             {PARTIAL_ON 'X'}
             {OFF '1'}
             {FULL_ON '0'}}
```

C.5 SV_LOGIC2UPF

```
create_hdl2upf_vct SV_LOGIC2UPF
-hdl_type sv
-table {{'X UNDETERMINED}
            {'1 PARTIAL_ON }
            {'1 FULL_ON }
            {'0 OFF }}
```

C.6 UPF2SV_LOGIC

```
create_upf2hdl_vct UPF2SV_LOGIC
-hdl_type sv
-table {{UNDETERMINED 'X}
            {PARTIAL_ON 'X}
            {FULL_ON '1}
            {OFF '0}}
```

C.7 SV_LOGIC2UPF_GNDZERO

```
create_hdl2upf_vct SV_LOGIC2UPF_GNDZERO
-hdl_type sv
-table {{'X' UNDETERMINED}
              {'0' FULL_ON}
              {'1' OFF}
              {'Z' UNDETERMINED}}
```

C.8 UPF_GNDZERO2SV_LOGIC

```
create_upf2hdl_vct UPF_GNDZERO2SV_LOGIC
-hdl_type sv
-table {{UNDETERMINED 'X}
             {PARTIAL_ON 'X}
             {OFF '1}
             {FULL_ON '0}}
```

C.9 VHDL_TIED_HI

```
create_upf2hdl_vct VHDL_TIED_HI
-hdl_type vhdl
-table {{UNDETERMINED 'X'}
            {FULL_ON '1'}
            {PARTIAL_ON 'X'}
            {OFF 'X'}}
```


C.10 SV_TIED_HI

```
create_upf2hdl_vct SV_TIED_HI
-hdl_type sv
-table {{UNDETERMINED 'X}
            {FULL_ON '1}
            {PARTIAL_ON 'X}
            {OFF 'X}}
```

C.11 VHDL_TIED_LO

```
create_upf2hdl_vct VHDL_TIED_LO
-hdl_type vhdl
-table {{UNDETERMINED 'X'}
            {FULL_ON '0'}
            {PARTIAL_ON '0'}
            {OFF 'X'}}
```

C.12 SV_TIED_LO

```
create_upf2hdl_vct SV_TIED_LO
-hdl_type sv
-table {{UNDETERMINED 'X}
            {FULL_ON '0}
            {PARTIAL_ON 'X}
            {OFF 'X}}
```

Annex D

(informative)

UPF procs

This annex contains Tcl procs developed to support the use of UPF.

load_protect_upf

```
proc load_protected_upf args {
  #
  # Default is to allow globals to be modified
  #
  set load_protected_hideGlobals 0

  #
  # Parse the command arguments
  #
  for {set i 0} {$i < [llength $args]} {incr i 1} {
    set arg [lindex $args $i]
    #
    # Handle options
    #
    if { [string index $arg 0] == "-" } {
      if { [string match "$arg*" "-version"] } {
        incr i 1
        set load_protected_version [lindex $args $i]
      } else {
        if { [string match "$arg*" "-scope"] } {
          incr i 1
          set load_protected_scope [lindex $args $i]
        } else {
          if { [string match "$arg*" "-params"] } {
            incr i 1
            set load_protected_params [lindex $args $i]
          } else {
            if { [string match "$arg*" "-hide_globals"] } {
              set load_protected_hideGlobals 1
            } else {
              puts "Error : load_protected_upf :
                    Unrecognised option $arg to load_protected_upf"
              return 0
            }
          }
        }
      }
    } else {
      #
      # There must be exactly one fileName argument
      #
      if { [info exists load_protected_fileName] } {
        puts "Error : load_protected_upf : File name
          $load_protected_fileName already specified when parsing $arg"
        return 0
      }
```

```
      set load_protected_fileName $arg
    }
  }

  #
  # Must specify file to load
  #
  if { ![info exists load_protected_fileName] } {
    puts "Error : load_protected_upf : No file name specified"
    return 0
  }

  #
  # Problem in Tcl if the errorInfo global variable is unset
  #
  set load_protected_keepGlobalsList [list errorInfo]

  #
  # Add the names of any globals that the tool relies on
  #
#  lappend load_protected_keepGlobalsList toolGlobal

  #
  # Handle any params
  #
  if { [info exists load_protected_params] } {
    foreach load_protected_param $load_protected_params {
      set load_protected_par [split $load_protected_param]
      if { [llength $load_protected_par] > 2 } {
        puts "Error : load_protected_upf : Bad param '$load_protected_param'"
        return 0
      }
      if { [llength $load_protected_par] == 1 } {
        #
        # Just paramName so make it global
        #
        lappend load_protected_keepGlobalsList [lindex $load_protected_par 0]
        global [lindex $load_protected_par 0]
      } else {
        #
        # paramName and paramValue so initialise local variable
        #
        set [lindex $load_protected_par 0] [lindex $load_protected_par 1]
      }
    }
  }

  #
  # Are we protecting globals from modification ?
  #
  if { $load_protected_hideGlobals } {
    #
    # Save the active values of all the globals
    # Unset all the globals apart from the ones in keepGlobalsList
    #
    set load_protected_globalList [load_protected_save_globals
   $load_protected_keepGlobalsList]
  }
```



```
#
# Set the UPF version if requested
#
set load_protected_origVersion [set_upf_version]
if { [info exists load_protected_version] } {
  set load_protected_origVersion [set_upf_version $load_protected_version]
}

#
# Set the scope if requested
#
set load_protected_origScope [set_scope .]
if { [info exists load_protected_scope] } {
  set load_protected_hierarchy_eparator [set_hierarchy_separator]
  set_scope $load_protected_scope
  if { [set_scope .] == $load_protected_hierarchy_separator &&
 $load_protected_scope != $load_protected_hierarchy_separator } {
    puts "Error : load_protected_upf : failed to set scope to
 $load_protected_scope"
    set_scope $load_protected_origScope
    return 0
  }
}

#
# Source the UPF file
#
if [catch {source $load_protected_fileName} load_protected_mssg] {
  #
  # Some error detected during sourcing the file
  #
  puts "Error : load_protected_upf : $load_protected_mssg"
  if { $load_protected_hideGlobals } {
    #
    # Restore the global variables if we unset them
    #
    load_protected_restore_globals $load_protected_globalList
 $load_protected_keepGlobalsList
  }
  set_scope $load_protected_origScope
  set_upf_version $load_protected_origVersion
  return 0
}

#
# Restore the global variables if we unset them
#
if { $load_protected_hideGlobals } {
  load_protected_restore_globals $load_protected_globalList
 $load_protected_keepGlobalsList
}

#
# Restore the scope in case it got changed
#
set_scope $load_protected_origScope

#
# Restore the version in case it got changed
```

```
  #
set_upf_version $load_protected_origVersion
  return 1
}

#
# Demonstration implementation of load_protected_save_globals
#
# Purpose :
#    Capture the active value of all global variables and unset them
#
# Usage :
#   load_protected_save_globals keepGlobalsList
#
#     keepGlobalsList list of global variable names that will not be unset
#
#    Returns list of names of variables that were unset plus a string that can
#    be evaluated to restore its original value
#
proc load_protected_save_globals
    {load_protected_save_globals_keepGlobalsList} {

  set load_protected_save_globals_globalList [list]

  #
  # Get list of all the global variable names
  #
  set load_protected_save_globals_infoList [info globals]
  for {set load_protected_save_globals_i 0} {$load_protected_save_globals_i <
    [llength $load_protected_save_globals_infoList]} {incr
    load_protected_save_globals_i 1} {
     set load_protected_save_globals_name [lindex
    $load_protected_save_globals_infoList $load_protected_save_globals_i]
     #
     # Skip the variable if it is in the keepGlobalsList
     #
     if { [lsearch $load_protected_save_globals_keepGlobalsList
    $load_protected_save_globals_name] >= 0 } {
       continue
     }

     #
     # Access its active value
     #
     global [lindex $load_protected_save_globals_infoList
    $load_protected_save_globals_i]

     #
     # Only need to save it if it exists
     #
     if { ![info exists $load_protected_save_globals_name] } {
       continue
     }

     if { [array exists $load_protected_save_globals_name] } {
       #
       # Array value. Convert to a list
       #
```



```
      set load_protected_save_globals_str "array set
    $load_protected_save_globals_name \[list [array get
    $load_protected_save_globals_name]]"
     } else {
      if { [llength [set [lindex $load_protected_save_globals_infoList
    $load_protected_save_globals_i]]] > 1 } {
        #
        # List value
        #
        set load_protected_save_globals_str "set
    $load_protected_save_globals_name \[list [set [lindex
    $load_protected_save_globals_infoList $load_protected_save_globals_i]]]"
      } else {
        #
        # Simple value
        #
        set load_protected_save_globals_str "set
    $load_protected_save_globals_name \"[set [lindex
    $load_protected_save_globals_infoList $load_protected_save_globals_i]]\""
      }
     }
     lappend load_protected_save_globals_globalList [list
    $load_protected_save_globals_name $load_protected_save_globals_str]

     #
     # Destroy the existing global variable
     #
     unset $load_protected_save_globals_name
   }
   return $load_protected_save_globals_globalList
}
#
# Demonstration implementation of load_protected_restore_globals
#
# Purpose :
#    Restore original values of global variables
#
# Usage :
#   load_protected_restore_globals globalList keepGlobalsList
#
#    globalList list of global variable names and string to be evaluated to set
#     the variables value
#     keepGlobalsList list of globals to be left untouched
#
proc load_protected_restore_globals
    {load_protected_restore_globals_globalList
    load_protected_restore_globals_keepGlobalsList} {

  # Destroy all the globals except the ones on the keepGlobalsList list
  set load_protected_restore_globals_infoList [info globals]
  for {set load_protected_restore_globals_i 0}
    {$load_protected_restore_globals_i < [llength
    $load_protected_restore_globals_infoList]} {incr
    load_protected_restore_globals_i 1} {
     set load_protected_restore_globals_name [lindex
    $load_protected_restore_globals_infoList
    $load_protected_restore_globals_i]
     #
```

```
    # Skip the variable if it is in the
    load_protected_restore_globals_keepGlobalsList
    #
    if { [lsearch $load_protected_restore_globals_keepGlobalsList
    $load_protected_restore_globals_name] >= 0 } {
      continue
    }

    #
    # Access its active value
    #
    global [lindex $load_protected_restore_globals_infoList
    $load_protected_restore_globals_i]

    #
    # Destroy the existing global variable
    #
    unset $load_protected_restore_globals_name
  }

  # Re-create the original globals
  foreach load_protected_restore_globals_globalVariable
    $load_protected_restore_globals_globalList {
    #
    # Make the variable global
    #
    global [lindex $load_protected_restore_globals_globalVariable 0]
    #
    # Set its value
    #
    eval [lindex $load_protected_restore_globals_globalVariable 1]
  }
}
```


Annex E

(informative)

De-rating factor for inertial glitch

In 8.3.4, glitching activities are categorized into two types, transport glitches and inertial glitches, and number of the glitch transitions are reported in the SAIF file. Transport glitches consume the same amount of power as normal toggles, so power consumption can be accurately calculated based on the number of transition. For inertial glitches, however, the number of transitions is not enough to accurately estimate the inertial glitching power dissipation.

To improve the accuracy for inertial glitching power estimation, it is recommended that a simulator provide a de-rating factor for each node in the circuit that has inertial glitches. As described below, this de-rating factor can be used to scale the inertial glitch count to an effective count of normal toggle transition. Power analysis tools can use the adjusted inertial glitch count to improve estimation accuracy.

Assume a gate has a total number of k delays, with a delay value of T_i ($i = 1...k$) for each delay.

Define N_i ($i = 1...k$) as the total number of inertial glitch pulses due to the delay T_i, and δ_{ij} as the timing difference of the input events that cause glitch j ($j = 1...N_i$) due to the delay T_i.

Define N_e as the total number of inertial glitch edges of the gate. It is easy to see that N_i and N_e satisfy Equation (E.1).

$$\sum_{i=1}^{k} N_i = \frac{N_e}{2} \tag{E.1}$$

NOTE—The total number of the glitch pulses is half of the total number of the glitch edges.

With the parameters defined above, a de-rating factor can be defined as shown in Equation (E.2).

$$K = 2 \times \frac{\sum_{i}^{k} \sum_{j}^{N_i} \frac{\delta_{ij}}{T_i}}{N_e} \tag{E.2}$$

Here is an example of how to use the de-rating factor. Consider again the example of the inverter shown in Figure E.1.

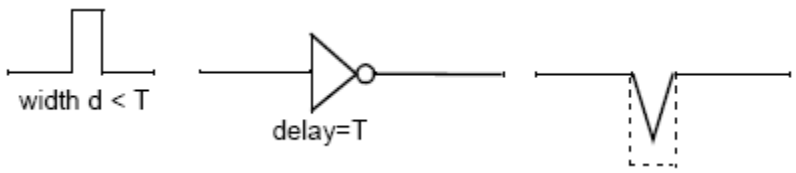

Figure E.1—Inverter

The power consumption at the output can be approximated as shown in Equation (E.3).

$$P = \frac{\delta}{T} \times 2 \times P0 \qquad 0 \le \delta \le T \tag{E.3}$$

where

$P0$ is the power consumption of the gate during one normal full-level transition
δ is the timing difference of the two input events that cause the glitch
T is the delay of the inverter

This equation indicates that the inertial glitching power dissipation can be roughly modeled by the timing difference of the input events that causes the glitch, and the delay of the gate beyond which there is no inertial glitch.

Accordingly, for a node with a total of N_i number of inertial glitch pulses due to the delay T_i ($i = 1...k$), the total power consumption can be estimated as shown in Equation (E.4).

$$P = \sum_{i=1}^{k} \sum_{j=1}^{N_i} \frac{\delta_{ij}}{T_i} \times 2 \times P0 \tag{E.4}$$

Replace it with the de-rating factor K, the power consumption can be simplified as shown in Equation (E.5).

$$P = K \times N_e \times P0 \tag{E.5}$$

This suggests that the inertial glitching power can be calculated by converting the number of glitching transitions into the number of normal transitions by applying a de-rating factor.