



TECHNICAL REPORT

EIA-548™

Issued 1988-03
Stabilized 2015-08

Electronic Design Interchange Format Version 200

NOTICE

This document has been taken directly from the original TechAmerica document and contains only minor editorial and format changes required to bring it into conformance with the publishing requirements of SAE Technical Standards. The release of this document is intended to replace the original with the SAE International document. Any numbers established by the original document remain unchanged.

The original document was adopted as an SAE publication under the provisions of the SAE Technical Standards Board (TSB) Rules and Regulations (TSB 001) pertaining to accelerated adoption of specifications and standards. TSB rules provide for (a) the publication of portions of unrevised specifications and standards without consensus voting at the SAE committee level, and (b) the use of the existing specification or standard format.

SAE Technical Standards Board Rules provide that: "This report is published by SAE to advance the state of technical and engineering sciences. The use of this report is entirely voluntary, and its applicability and suitability for any particular use, including any patent infringement arising therefrom, is the sole responsibility of the user."

SAE reviews each technical report at least every five years at which time it may be revised, reaffirmed, stabilized, or cancelled. SAE invites your written comments and suggestions.

Copyright © 2015 SAE International

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of SAE.

TO PLACE A DOCUMENT ORDER: Tel: 877-606-7323 (inside USA and Canada)
Tel: +1 724-776-4970 (outside USA)
Fax: 724-776-0790
Email: CustomerService@sae.org
<http://www.sae.org>

SAE WEB ADDRESS:

Full document details: SAE EIA548

SAE values your input. To provide feedback on this Technical Report, please visit <http://www.sae.org/technical/standards/EIA548>

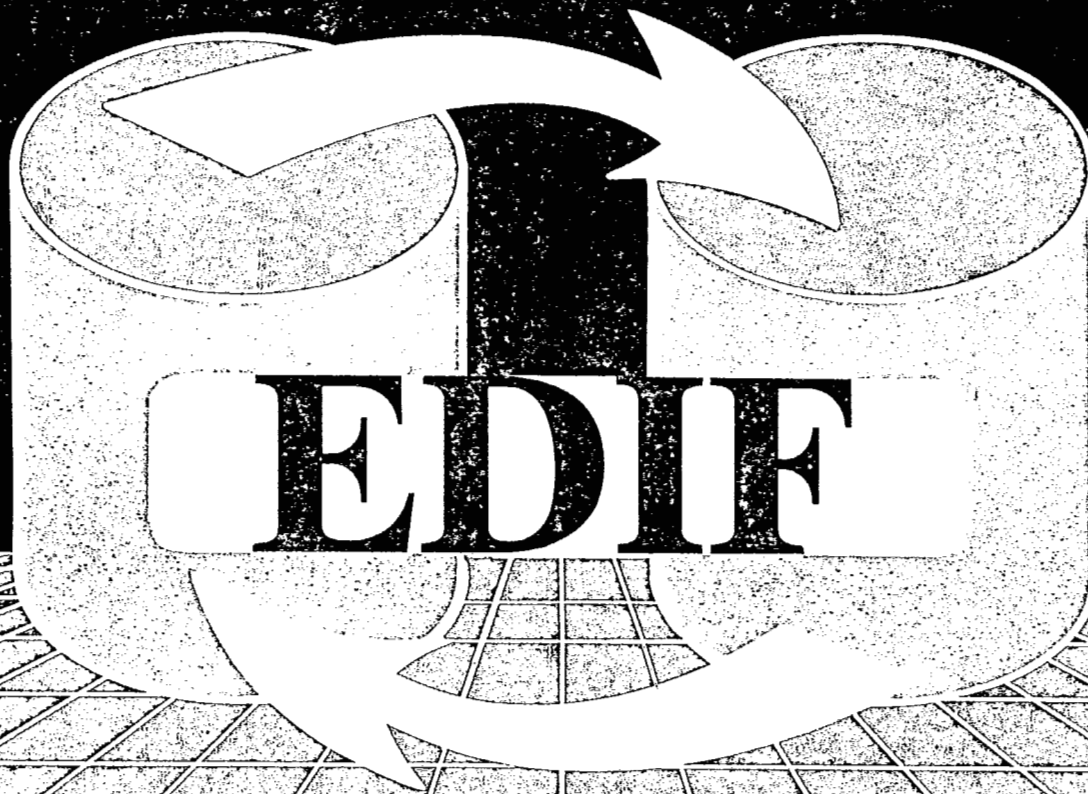
Documents

Document Sharing Hub

EIA-548-1988
APPROVED: MARCH 14, 1988

Electronic Design Interchange Format Version 200

Recommended Standard EIA-548



ELECTRONIC INDUSTRIES ASSOCIATION
ENGINEERING DEPARTMENT



Documents

Document Sharing Hub

Electronic Design Interchange Format

Notice

EIA Engineering Standards and Publications are designed to serve the public interest through eliminating misunderstanding between manufacturers and purchasers, facilitating interchangeability and improvements of products, and assisting the purchaser in selecting and obtaining with minimum delay the proper product for his particular need. Existence of such Standards and Publications shall not in any respect preclude any member or non-member of EIA from manufacturing or selling products not conforming to such Standards and Publications, nor shall the existence of such Standards and Publications preclude their voluntary use by those other than EIA members, whether the standard is to be used either domestically or internationally.

Recommended Standards and Publications are adopted by EIA without regard to whether or not their adoption may involve patents on articles, materials, or processes. By such action, EIA does not assume any liability to any patent owner, nor does it assume any obligation whatever to parties adopting the Recommended Standard or Publication.

This EIA Standard is considered to have International Standardization implication, but the International Electrotechnical Commission activity has not progressed to the point where a valid comparison between the EIA Standard and the IEC document can be made.

This standard does not purport to address all safety problems associated with its use or all applicable regulatory requirements. It is the responsibility of the user of this Standard to establish appropriate safety and health practices and to determine the applicability of regulatory limitations before its use.

The Electronic Industries Association (EIA) and the EDIF Committee do not possess full access to the data concerning the uses and applications for which someone may use the information contained within this document. Therefore, no responsibility is assumed by the EIA or EDIF Committee for the application of any design, nor for any infringements of patents or rights of others which may result from the use of this document. A user of EDIF assumes the risk that EDIF may be unsuitable for the user's purposes. The EIA and EDIF Committee reserve the right to change or amend this document at any time.

EDIF

Electronic Design

Interchange Format

Version 2 0 0

Edited by
Paul Stanford and Paul Mancuso

1989
Electronic Industries Association
EDIF Steering Committee

Published by
ELECTRONIC INDUSTRIES ASSOCIATION
Engineering Department
2001 Pennsylvania Ave., N.W.
Washington, D.C. 20006

Copyright 1989. All rights reserved.

Principal Editor: Paul Stanford
Assistant Editor: Paul Mancuso

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise without the prior written permission of the Electronic Industries Association.

ISBN: 0-7908-0000-4

Library of Congress Catalog Number: 87-81322

Price: \$60.00 (student price: \$40.00)

Second Edition 1990

Preface to the Second Edition

This edition of the EDIF Reference Manual incorporates various editorial corrections to the document but does not represent any change to the format itself.

Most of the corrections and clarifications are the result of Technical Committee replies to user questions. Many are of a simple typographical nature within the text or examples of the reference section. An appendix has also been added which summarizes the restrictions on *interface* and *contents* for each view type.

The following corrections and additions have been made since the first edition. Times are non-negative in *after*, *cycle*, and *loadDelay*. The expansion of arcs in *cornerType* and *endType* is explained. The temporary nature of *figureGroupOverride* is explained. Global port references in *globalPortRef* and *joined* are clarified. References within *instanceGroup* must be within the same view. Symbolic constants may not be aliased in *keywordAlias* and *keywordNameRef*. The contextual restrictions on *logicOneof* within *logicInput* and *logicWaveform* have been specified. Array sizes must match in *logicOneof* and *match*. Clarification of tree structure restrictions for *mustJoin*, *nonPermutable*, *permutable*, and *weakJoined* are given. The nature of object grouping taken together in *netGroup* and *portGroup* is given. The restrictions on *keywordDisplay* for *protectionFrame* and *symbol* are given. Clarification for the EDIF level restrictions within *parameterAssign* and *valueRef* are given.

There are also corrections to examples in *after*, *designator*, *difference*, *figure*, *follow*, *instanceMap*, *netMap*, *nonPermutable*, *parameterAssign*, *portGroup*, *portMap*, *string*, *userData*, *viewList*, *visible* and *when*.

Rewording for clarity has been done in *cell*, *enclosureDistance*, *figureGroupNameDef*, *intraFigureGroupSpacing*, *overhangDistance*, *portNameDef* and *transform*.

Typographical errors have been removed in *cellType*, *contents*, *cornerType*, *edif*, *follow*, *interface*, *iterate*, *joined*, *name*, *nameDef*, *netBackAnnotate*, *offPageConnector*, *portImplementation*, *property*, *protectionFrame*, *then*, *variable* and *viewType*.

Dallas, 1989

Paul Stanford



Documents
Document Sharing Hub

Foreward

This document is the authoritative reference document for EDIF Version 2 0 0. It defines the syntax and semantic content of every EDIF construct. It should be taken as correct when conflict or contradiction is found elsewhere. While great care has been taken to ensure accuracy and internal consistency, errors or ambiguous interpretations are always possible; if any such problems are noted the user is urged to contact the EDIF Users' Group at the following address:

EDIF Users' Group
P.O. Box 25542
Tempe, Arizona 85285-5542, USA

It is not expected that a new user of EDIF will use this document alone to gain an understanding of EDIF, nor to learn the correct application of EDIF. For this purpose an EDIF user guide and several EDIF style guides are planned which will further explain the overall structure of EDIF and the application of EDIF to specific areas of technology.

The reader is cautioned to exercise care in creating an EDIF file. Under no circumstances should syntax be guessed or invented to solve a problem or to be used as a shortcut.

In tests using previous versions of EDIF, some of the more common errors found were: (1) use of floating point numbers such as 100.000; (2) improper keyword abbreviations, i.e. use of keywords that were not given keyword definitions; (3) use of human-readable object names such as CMOS_Library to carry semantic weight in the EDIF file; and (4) missing EDIF Level and EDIF Version information. In each case the reader of the information is forced to guess at the meaning or interpretation or forced to write some form of special software just to read this particular file.

This manual is divided into the following sections:

- Introduction. The information within this section provides an overview of EDIF basics.
- Reference. This section describes in detail the constructs of all levels of EDIF. Once the reader is familiar with EDIF, this section may be used as a basic reference source.
- Grammar. This section describes the grammar of EDIF. It may be used by experienced EDIF programmers for initial coding of a scanner and parser, and for quick reference while developing translators.

For EDIF, upward compatibility is interpreted to mean that software which is designed to read a future EDIF version (e.g. a future EDIF Version 3 0 0) will also be able to read and correctly interpret a file which was written by software designed for an older EDIF version (such as EDIF Version 2 0 0). This capability requires compatibility at three levels of interpretation: the lexical level, the syntactic level, and the semantic level.

The most basic level of compatibility - semantic compatibility - already exists between EDIF Versions 1 0 0, 1 1 0 and the current version of EDIF, since the underlying data models of the earlier EDIF versions have been retained in the current version. Hence, anything which was expressible with the earlier EDIF versions is still expressible using EDIF Version 2 0 0. The EDIF technical contributors and reviewers considered carefully the requirements for upward compatibility at all three of these levels while preparing this version of EDIF. Everything which is included in EDIF Version 2 0 0 is expected to be completely upward compatible for the future, but this is not guaranteed at this time.

To guard against the possibility of uncovering some subtle problem in actual use which would violate upward compatibility, the EDIF reviewers have elected to wait until after some period of production use before attempting to identify the parts of EDIF which will be upward compatible. At that time certain keywords and groups of keywords will be identified as frozen or upwardly compatible. From this point on any changes to those keywords and their constructs will be made only in an upward compatible way. All future EDIF versions will maintain the syntax and semantics of those keywords, adding only optional or extended constructs within the keywords. The upward compatibility of the keywords which allow user-definable semantics (*userData* and *property*) cannot be controlled by the EDIF technical contributors but only by the originators of these constructs to which semantic interpretation is applied.

The capability to read a new EDIF file with obsolete software ("downward" compatibility) is not planned, since any significant changes to EDIF will involve the addition of new capabilities to EDIF with new and presumably significant information present in a file written using the new version. Obsolete software must be modified to handle the new information if only to know that the information may be safely ignored. An EDIF reader can be written in such a way that both the current EDIF version 2 0 0 and future versions could be read correctly without requiring modifications to the program, unless the new information is required.

TABLE OF CONTENTS

Section 1

INTRODUCTION TO EDIF	1-1
1.1 EDIF Meta-Syntax	1-1
1.2 EDIF Syntax	1-2
1.3 Names	1-2
Denoting Names or Identifiers	1-2
The Role of Names in EDIF	1-3
Name Class	1-4
Name Scope	1-4
Renaming for External Use	1-5
1.4 Values	1-6
Numbers	1-6
Strings	1-7
Booleans	1-7
Points	1-7
Ranges of Numbers	1-7

Section 2

REFERENCE	2-1
-----------------	-----

Section 3

EDIF GRAMMAR	3-1
3.1 Introduction	3-1
3.2 Notations	3-1
3.3 Level 0	3-2
3.4 Level 1	3-30
3.5 Level 2	3-34
3.6 Keyword Level 1	3-36
3.7 Keyword Level 2	3-37
3.8 Keyword Level 3	3-39

Section 4

CHARACTER LIST	4-1
----------------------	-----

TABLE OF CONTENTS [continued]**Section 5**

SYMBOLIC CONSTANTS	5-1
--------------------------	-----

Section 6

VIEWTYPE MATRICES	6-1
-------------------------	-----

Index

EDIF Version 2 0 0

LIST OF FIGURES

2-1. Corner-type round	2-60
2-2. Corner-type extend for obtuse angles	2-61
2-3. Corner-type truncate	2-61
2-4. Corner-type extend for acute angles	2-62
2-5. End types: (a) truncate (b) extend (c) round	2-99
2-6. NonPermutable example	2-243
2-7. The eight orthogonal orientations	2-260
2-8. Nested permutable example	2-280
2-9. A typical symbol	2-360

April 27, 1987



Documents
Document Sharing Hub

INTRODUCTION TO EDIF

This section provides an overview of basic EDIF concepts. These include the levels of EDIF, and the syntax and the meta-syntax used throughout the document. All concepts described in this section apply to all levels of EDIF.

An interchange format for electronic design data must meet the needs of a wide range of users – from those wishing to transfer mask artwork data in its final form, to those desiring to transfer a behavioral description of a microcomputer. A procedural format could be used to describe the mask artwork data, but the data can be expressed as effectively using simple geometric primitives.

A primary requirement of an interchange format is unambiguous description of the data it is transmitting. Consequently, it is important that all readers and writers of the format possess exactly the same understanding of how the file is to be interpreted.

To avoid complicating the simple tasks and still provide the power of procedural descriptions for those who need them, EDIF descriptions can be specified in three different levels of complexity. These are:

- Level 0 : The basic level – only literal constants are permitted.
- Level 1 : Level 0 with the addition of parameters and expressions.
- Level 2 : Level 1 with the addition of control-flow constructs.

Since Level 1 is a proper subset of Level 2, and Level 0 is a proper subset of Level 1, it is possible to move down a level by evaluating functions and expressions and by replacing parametric values with a constant value. Such a conversion would be necessary to move from a sophisticated design environment to a more straightforward system and would involve the loss of parametric information.

Reference page titles are followed by EDIF level in chevrons, unless the construct is the same for all levels: <0>, <1>, or <2>. In a similar way, constructs involved in keyword abbreviations are separated into keyword levels, indicated by <k0>, <k1>, <k2>, or <k3>. These keyword levels allow no abbreviations, simple keyword abbreviations, noniterative keyword definitions and iterative keyword definitions respectively.

EDIF Meta-Syntax

The meta-syntax used in the following sections to describe EDIF constructs obeys the following rules:

- i. Syntactic categories (nonterminals) are printed in *italics*; literal words and characters (terminal strings) are **emboldened**. Symbolic constants are printed

in roman and are in **UPPERCASE**; single characters are represented as '+' and parentheses represent themselves.

- ii. If a construct is enclosed in [square brackets], it is optional.
- iii. If a construct is enclosed in {curly braces}, it may be repeated zero or more times in any order.
- iv. A choice is indicated with a vertical bar |. Only one of the options may be chosen. A list of items enclosed within {curly braces} and separated by vertical bars indicates that any number of each item may be present and that the items may occur in any order. Inside such a list, if an item is permitted to occur at most once, it is enclosed within <chevrons>.
- v. Notes on the right-hand side are comments and are not part of the syntax.
- vi. An ellipsis ... is used to indicate that further items have been omitted within an example.
- vii. All EDIF names must be defined before they are referenced.

EDIF Syntax

EDIF uses a Lisp-like syntax. The syntax was chosen for ease of parsing and provision of a uniform method of extension of the format. A construct begins with an opening parenthesis and a keyword; this is followed by a list of items ending with a closing parenthesis. Those items may be other constructs which build a nested structure, or they may be tokens consisting of data items, such as integer tokens or string tokens, or EDIF identifiers. Delimiters may occur on either side of parentheses in EDIF constructs but are not required. EDIF delimiters are the ASCII characters space, tab, carriage-return, and linefeed.

The list of legal identifiers that play the role of EDIF keywords is defined by this document. Users of the format can define their own keywords in terms of existing EDIF keywords with the *keywordAlias* and *keywordDefine* constructs. Case is not significant in keywords, symbolic constants, or identifiers.

The specification of a particular construct is summarized between horizontal lines and in the paragraphs following the entry.

Names

Denoting Names or Identifiers

Identifiers are used in an EDIF description to uniquely identify objects. EDIF attaches no special meaning to the choice of identifier spelling; the basic semantics of an EDIF file is not changed if one or more of the identifiers is uniformly changed

EDIF Version 2 0 0

for a new (and valid) identifier both in the original definition which names an object and in all uses of the identifier to refer to that object. All structural semantics should be identical; the only change would be in the appearance of the name itself. No meaning should be assumed or expected with respect to the actual choice of names. This sharply contrasts with native EDIF keywords (and symbolic constants), whose semantic meaning is fully defined by this manual.

The Role of Names in EDIF

There are two mechanisms in EDIF to express relationships between objects: containment and name reference.

Containment is a purely lexical mechanism and relates to the textual enclosure of object descriptions within the matching parentheses of another description (construct).

Many relationships are expressed using simple containment. A *cell* definition being part of a *library*, and a *view* being within a particular *cell* are examples of this type. However, containment alone is only adequate to describe simple, one-dimensional structures. There are many relationships between objects in a typical CAD database which require more flexibility, such as the instantiation of one cell within another and netlist connectivity.

The details of naming structures within EDIF descriptions are derived partially from the need to represent such relationships and partly from common identification schemes used in typical systems. As far as possible, minimum real restrictions are placed on originating naming schemes. In particular, there are no reserved identifiers in EDIF.

To make the format as explicit as possible, there are no assumed relationships between the names of distinct classes of objects such as cells, ports, and nets. For example, a port may be declared with the same name as a cell without any implied significance or semantic implication: it is simply a coincidence of spelling.

Another important aspect of the naming mechanism is that each name *must* be defined before it may be referenced.

Very few design systems employ flat naming schemes: names are typically valid within a well-defined scope. For example, the name chosen for a port in one cell cannot be confused with a port of another cell that happens to have the same name spelling. Such local names require some form of qualification before the local object can be referenced from another context. For example, to reference an instance port requires both the instance name and the name given to the corresponding port within the definition of the cell.

Name Class

The concept of *name class* is an essential aspect of object referencing in EDIF. The type of object being named determines the class of its name, such as cell name or logic-value name. Any identifier used as a reference is associated with an object class by its immediate context.

The various name classes are all subject to the same scoping rules and are parallel, independent naming schemes. There is no connection or interference between names of different class, even when such names happen to involve the same identifier spelling and have overlapping ranges. In addition, none of these classes have any conflict with identifiers used as keywords or symbolic constants: a cell could be named **and** or even **cell** without implying any connection with the corresponding keywords.

Some name classes are used for more than one keyword. For example, *external* and *library* both define library names.

Name Scope

The *scope* of a name is always defined by lexical containment within a certain construct, such as *library*, *cell*, or *view*. The range of direct accessibility of a named object extends from just after its description to just before the end of the smallest enclosing name scope.

It is illegal for two objects of the same class to be given the same name if they also have the same scope. For example, no two cells in the same library may be given the same name.

The major name scopes created within EDIF are illustrated by the following example:

EDIF Version 2 0 0

```

(edif example
  (edifVersion 2 0 0)
  (edifLevel 0)
  (keywordMap (keywordLevel 0))
  (library lib1
    (edifLevel 0)
    (technology
      (numberDefinition ...)
      (simulationInfo (logicValue low ...))
      (figureGroup m2 ...))
    (cell c1 (cellType Generic)
      (view v1 (viewType Netlist)
        (interface (port data ...))
        (contents
          (instance inv12 ...)
          (net q ...))))))

```

In this example the scopes of **low**, **m2**, and **c1** extends from just after their definitions to the end of the library. The scope of **v1** extends from just after its definition to the end of the cell. The scope of **data** extends from just after its definition to the end of the view. The scopes of **inv12** and **q** extend from just after their definitions to the end of the contents.

Renaming for External Use

When the sending system allows more freedom in spelling than EDIF, there is a mechanism for recording the original spelling of the name. This is necessary when the external identifier does not meet the naming restrictions of EDIF. For this situation, the *rename* construct is provided. The *rename* provides a legal EDIF name for the illegal identifier. That name is used throughout the scope of the identifier and can be mapped back to the original identifier on output, if desired. For example, rather than:

```
(cell dollar ...)
```

the rename would be used as:

```
(cell (rename dollar "$") ...)
```

This defines a cell named **dollar**, and records the fact that it used to be called **\$** in the originating system. The legal identifier **dollar** is then the only name used within the rest of the file. A receiving translator should attempt to preserve the original names if possible; this could improve human communication. This has to be an individual design choice: few systems accept unrestricted identifier spellings completely.

April 27, 1987

Values

A second important general category of object is the *value*. In the following sections, the primitive value types supported by EDIF are described.

Numbers

A number is either an integer or scaled-integer quantity, and either one may be used in any EDIF description that expects a number. The integer portion of a number is restricted to lie in the range $+2^{31}-1 \geq m \geq -2^{31}+1$. This restriction guarantees that a program processing the file can represent any integer, or mantissa, using a 32-bit one's or two's complement integer. All of the following are legal integer tokens:

```
100
0
-123454321
+64
```

A scale factor can be applied to an integer to represent fractional numbers or numbers greater than can be expressed using an integer alone. These scaled integers are represented in the form: (*e integerToken integerToken*). The first integer is the mantissa; the second integer is the exponent of the scale factor, and the resulting number has a value of mantissa $\times 10^{\text{exponent}}$. While a restriction is not applied to the values of the scale factor itself, the resulting scaled integer must lie within the range $\pm 1 \times 10^{35}$, so that it may be expressed as a floating-point real number on most conventional computer systems.

Real numbers are represented as *scaledIntegers*; therefore, real-number notation is not allowed in EDIF. Integers of 32 bits can be used to store any mantissa values for both *integerTokens* and *scaledIntegers*. A 64-bit (double) floating-point number can store any *scaledInteger* within the range specified above. In a system that does not have double-precision floating point, a 32-bit integer can be used to store the mantissa, and a separate byte can be used to store the exponent. The following examples show numbers expressed in real-number notation and the *scaledInteger* equivalent:

100.0 →	(e 1000 -1)
0.8 →	(e 8 -1)
-2321. →	(e -2321 0)
+6E12 →	(e 6 12)

Numbers are used to represent a variety of types of data. For example, numbers are used for the width of paths (a distance) and the delay of a logic gate (a time). Numbers may be translated upon output to dimensioned quantities via the *scale*

EDIF Version 2 0 0

construct. That is, while a path width may be expressed as 3, it is converted to 3 μ m or 3 meters depending on the value of the distance scale factor for the current library.

Strings

String tokens are delimited by quotation marks (") and can contain legal EDIF characters. A string has no limit to its length and there is no implication that adjacent strings are concatenated. Within strings, arbitrary ASCII characters can be included by using the *asciiCharacter* mechanism.

Booleans

Boolean constants can take only one of two values: (true) or (false). This is extended in Level 1 descriptions to include Boolean expressions.

Points

A point value is the data type used to describe all geometric figures, including both line drawings and filled areas. It is defined by the keyword *pt* followed by two integer values. The first integer is the x-location and the second integer is the y-location of the point. For example: (pt 100 200).

Ranges of Numbers

Another possible value for some constructs is the *mmm* construct. It is used to specify minimum, nominal and maximum values. For example: (mmm 95 100 105).

April 27, 1987

**Documents**

Document Sharing 147

REFERENCE

<i>edif</i>	<i>\$goal</i>
	<i>\$goal</i>

The grammatical goal of an EDIF description is the *edif* form, possibly surrounded by *whiteSpace* characters.

Such a file need not be logically complete, since external library declarations may be used. The design or designs may also be partially described or not fully developed.

A legal EDIF file may be created to communicate a design that contains design errors (such as violations of physical design rules), possibly for verification or correction at another location.

See also:

design, *edif*, *edifLevel*, *external*, *form*, *identifier*, *integerToken*, *keywordLevel*, *library*, *nameDef*, *nameRef*, *stringToken*, *viewType*, and *whiteSpace*.

(abs <i>numberValue</i>)	<i>abs</i>
	<i>abs</i>

Abs is a number function that returns the absolute value of its argument; it may be used only in Level 1 and Level 2 descriptions. In the special case where *abs* is applied to an *integerValue* the expression may also be used as an *integerValue*.

Example:

(pt (abs -5) (abs 4))

In this example the integer value **5** is returned as the x-coordinate of the point; the integer value **4** is returned as the y-coordinate of the point.

Used in:

integerValue, and *numberValue*.

See also:

divide, *edifLevel*, *max*, *min*, *mod*, *negate*, *subtract*, and *sum*.

(acLoad miNoMaxValue | miNoMaxDisplay)

*acLoad**acLoad*

AcLoad is an attribute of a port and is used to express the external load capacitance of the port. This quantity has the units of **CAPACITANCE**. The total capacitance seen at the output port of a gate is the sum of the *acLoad* values of all connected ports.

Values of *acLoad* scale between *libraries* using the factors defined for **CAPACITANCE** values.

Example:

(acLoad (e 25 -15))

This example specifies an *acLoad* value of **25 fF**, assuming a scale factor of **1**.

Used in:

port, *portBackAnnotate*, and *portInstance*.

See also:

keywordDisplay, *loadDelay*, *scale*, and *unit*.

actual <k2>

EDIF Version 2 0 0

(actual <i>formalNameRef</i>)	<i>actual</i>
	<i>actual</i>

Actual generates the actual token or form that corresponds in position within the use of the keyword being defined, to the formal parameter definition within the *keywordDefine* form. If the *formal* parameter is bound to several forms, the *actual* expands to several forms – unless it is used within a *forEach* statement which references the same formal parameter.

This is only appropriate within descriptions expressed at keyword level 2 or higher.

Example:

```
(keywordDefine EG
  (keywordParameters
    (formal layer)
    . . . )
  (generate
    (build figure
      (actual layer)
      . . . ) ) )
```

This will, if given the usage (**EG POLY . . .**), generate the form (**figure POLY . . .**) during the keyword expansion.

Used in:

build, *forEach*, *generate*, and *optional*.

See also:

formal, *keywordDefine*, *keywordLevel*, and *literal*.

```
(after miNoMaxValue
  { logicAssign /
    follow /
    maintain /
    comment /
    userData }
)
```

after

after

After is used to specify a time delay prior to execution of a series of actions in logic modeling. The time delay may not be negative. *After* differs from *delay*. Its actions are evaluated after the specified time; *delay* simply delays propagation of the result of an action. This is useful for checking values a specified time after a *trigger* has occurred.

Example:

```
(when
  (trigger (change CLK (becomes T)))
  (logicAssign OUT (logicRef X) (delay 2))
  (after 5
    (logicAssign OUT (portRef IN))))
```

In the above example the *logicAssign* statement takes the logic value **X** at the time of the *trigger* and will apply the value to **OUT**, 2 time units later. The *after* statement causes a delay of 5 time units after the *trigger* before taking the value of **IN** and applying it to **OUT** immediately. By that time, the value of port **IN** may have changed.

Used in:

when.

See also:

delay, follow, logicAssign, and maintain.

$\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline \text{'A'} & \text{'B'} & \text{'C'} & \text{'D'} & \text{'E'} & \text{'F'} & \text{'G'} & \text{'H'} & \text{'I'} & \text{'J'} & \text{'K'} & \text{'L'} & \text{'M'} & \\ \hline \text{'N'} & \text{'O'} & \text{'P'} & \text{'Q'} & \text{'R'} & \text{'S'} & \text{'T'} & \text{'U'} & \text{'V'} & \text{'W'} & \text{'X'} & \text{'Y'} & \text{'Z'} & \\ \hline \end{array}$

$\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline \text{'a'} & \text{'b'} & \text{'c'} & \text{'d'} & \text{'e'} & \text{'f'} & \text{'g'} & \text{'h'} & \text{'i'} & \text{'j'} & \text{'k'} & \text{'l'} & \text{'m'} & \\ \hline \text{'n'} & \text{'o'} & \text{'p'} & \text{'q'} & \text{'r'} & \text{'s'} & \text{'t'} & \text{'u'} & \text{'v'} & \text{'w'} & \text{'x'} & \text{'y'} & \text{'z'} & \\ \hline \end{array}$

Alpha is a character of the set A..Z or a..z which is used within string tokens and identifiers. Case is significant within string tokens but not in identifiers (for the purpose of comparison).

Example:

```
netA
neta
"a string"
"A string"
```

In this example, the two identifiers are equivalent, but the two string tokens are distinct.

Used in:

identifier, and *stringToken*.

See also:

digit, and *specialCharacter*.

(and { *booleanValue* })

and

and

And is a Boolean function that returns the logical conjunction of its arguments. It is *false* if any of its arguments are *false*; otherwise it returns *true*. If used with no arguments, it returns *true*.

Example:

```
(constant x (integer 2))  
(constant y (integer 2))  
(constant z (integer 3))  
...  
(and (equal x y) (equal y z))
```

In the above example, $x = y = 2$, and $z = 3$; therefore, *(equal x y)* is *true*, and *(equal y z)* is *false*. The *and* thus returns the Boolean value *false*.

Used in:

booleanValue.

See also:

equal, *not*, *or*, and *xor*.

(annotate *stringValue* | *stringDisplay*)

*annotate**annotate*

Annotate is used in *commentGraphics* for associating comment text with an object. The string value has an association with the object which the *commentGraphics* is contained within. The string has no interpretation and is equivalent to a comment in a programming language.

Annotate may also be used within a *symbol* to associate textual display that should appear within contexts that use the symbol.

The string can be displayed using *stringDisplay*. The *stringDisplay* contains *display* forms which are used to define the location and system attributes necessary to display the string value.

Example:

```
(commentGraphics
  (annotate
    (stringDisplay "This is an example of annotate"
      (display textDisplay (origin (pt 12 13))))))
```

In this example the string "This is an example of annotate" will be displayed at (12, 13), according to the display attributes associated with the figure group named *textDisplay*, which must have been defined previously.

Used in:

commentGraphics, and *symbol*.

See also:

display, *figureGroup*, *property*, and *stringDisplay*.

	<i>apply</i>
(<i>apply</i> <i>cycle</i>	
{ <i>logicInput</i>	inputs and their values
<i>logicOutput</i>	outputs and expected results
<i>comment</i>	user comments
<i>userData</i> }	user-defined constructs
)	
	<i>apply</i>

Apply is used within *simulate* to describe input stimuli and expected responses over a certain time interval. The *cycle* specifies the number of input and output cycles described and the interval between each application of a logical pattern. The remainder of the construct consists of logic input and output statements, which individually describe the respective behavior of their subject ports.

Example:

```
(apply
  (cycle 7 (duration (e 5 -1)))
  (logicInput dataIn (logicWaveform T T F T F F T))
  (logicOutput dataOut (logicWaveform F T F F T F T)))
```

This example indicates that 7 test cycles are desired, each of which will take half a time unit.

During the first cycle, T is applied to **dataIn** and F is expected from **dataOut**.

During the Last cycle T is applied to **dataIn** and T is expected from **dataOut**.

Used in:

simulate.

See also:

logicInput, and *logicOutput*.

arc

EDIF Version 2 0 0

(arc *pointValue pointValue pointValue*)

*arc**arc*

Arc is used in describing shapes with curved edges. The first *pointValue* is the starting point of the arc, the second *pointValue* is any point on the *arc*, and the third *pointValue* is the end point of the arc.

The three points of the arc specification must be distinct.

Example:

```
(shape
  (curve
    (pt 20 100)
    (arc
      (pt 40 100)
      (pt 60 110)
      (pt 40 120))
    (pt 20 120)))
```

This example shows a *shape* describing a curved figure and the use of an *arc* to construct such a shape.

Used in:

curve.

See also:

figure, *openShape*, and *shape*.

(array <i>nameDef</i>	<i>array</i>
<i>integerValue</i> { <i>integerValue</i> }	name of arrayed object
)	range for each index
	<i>array</i>

Array is used in place of *nameDef* in contexts in which array structures are appropriate for the class of object involved. The class of array name is the same as that of a simple name in the same context. The *array* specifies the size of each index dimension of the created array, in order. Such values must be positive integers. There is no limit to the number of dimensions.

The use of *array* causes several objects of the given class to be created. Individual members may be accessed or referenced by using the *member* construct, which must contain the same number of indices as those in the *array*; each must be greater than or equal to zero and less than the corresponding dimension size. The first index varies the most slowly for the purpose of any expansion.

Example:

(array X 32)

This example defines an array X of size 32, running from (member X 0) through (member X 31).

(array Q 4 16)

The second example defines a two-dimensional array Q of size 4 by 16, running from (member Q 0 0) through (member Q 0 15) through (member Q 3 15).

Used in:

instanceNameDef, *netNameDef*, *portNameDef*, and *valueNameDef*.

See also:

member, and *nameDef*.

(arrayMacro *plug*)

*arrayMacro**arrayMacro*

ArrayMacro identifies a *plug* which describes the customizing patterns used to create a predefined function when placed at a valid site location on a gate array. The construct also includes a *plug* definition which may be matched to a *socket* for placement of the macro cell.

Example:

```
(arrayRelatedInfo
  (arrayMacro
    (plug ...)))
```

This indicates that the current cell is a macro and contains necessary *plug* information.

Used in:

arrayRelatedInfo.

See also:

plug, *site*, and *socket*.

```

(arrayRelatedInfo
  arrayMacro | arraySite | baseArray
  { comment |
  userData }
)

```

*arrayRelatedInfo**arrayRelatedInfo*

ArrayRelatedInfo is used to contain the EDIF constructs for describing physical placement rules which define the placement of customizing macrocells in a gate array. It may be used in the *interface* of views of type MASKLAYOUT, PCBLAYOUT, STRANGER, and SYMBOLIC.

Example:

```

(cell M299 (cellType generic)
  (view Site_definition (viewType Symbolic)
    (interface
      (arrayRelatedInfo
        (arrayMacro
          (plug
            (socketSet (symmetry)
              (site
                (viewRef Site (cellRef Site_cell))
                (Transform (orientation R90))))))))))

```

This example specifies **cell M299**, which may be placed in site **Site_cell** after being rotated **270** degrees.

Used in:

interface.

See also:

cellType, plug, socket, and viewType.

(arraySite socket)

arraySite

arraySite

ArraySite is used to define a site location in a gate array which may be satisfied by a macrocell placement during implementation of a gate array. A cell of this type consists of geometry which defines the underlying base pattern for the generic portion of a gate array. Different sites may be defined for different classes of underlying circuitry or for different physical realizations of similar functions such as high-power and low-power sites.

Example:

```
(Cell Site_cell (cellType generic)
  (View Site_definition (viewType Symbolic)
    (Interface
      (ArrayRelatedInfo (ArraySite (Socket))))))
```

This example specifies cell **Site_cell**, which may be instantiated in a base array to provide the required sites for macro placement.

Used in:

arrayRelatedInfo.

See also:

arrayMacro, plug, and socket.

<i>'%'</i>	<i>asciiCharacter</i>
<i>{ integerToken }</i>	
<i>'%'</i>	<i>asciiCharacter</i>

AsciiCharacter represents a sequence of characters within a *stringToken*; the value of each *integerToken* is interpreted as an ASCII code, and must lie between 0 and 127 inclusive. *WhiteSpace* may exist within this construct to separate consecutive integer tokens, but is optional before the first value and after the last value.

None of the special properties associated with various characters (such as quote and percent) should be attributed to these characters. This enables any ASCII character to be explicitly incorporated into a string token.

Example:

"abc%9%def"

This string contains seven characters.

"% 8 9 10 13 32 34 37 %"

The second example illustrates a *stringToken* which represents the characters: back-space, horizontal-tab, line-feed, carriage-return, space, quote, and percent.

Used in:

stringToken.

See also:

identifier, and *integerToken*.

<pre>(assign valueNameRef typedValue)</pre>	<i>assign</i> variable to assign to
	<i>assign</i>

After a variable has been defined by using the *variable* construct in Level 2, it is possible to assign a new value to it using the *assign* statement. The type of value must match the type of variable.

The *valueNameRef* may be an array, in which case the size and structure of the *boolean*, *integer*, *miNoMax*, *number*, *point*, or *string* construct must match the array dimensions as specified by the *array* in the *variable*.

The use of *assign* is not restricted by *viewType*.

Example:

```
(variable x (number 12))
...
(assign x
  (number
    (subtract (product b b) (product 4 a c))))
```

This example reassigns the value of the variable **x** to $b^2 - 4ac$.

```
(assign xLoc
  (integer (xCoord Corner)))
```

In the above example the variable **xLoc** is assigned the x coordinate of the identifier named **Corner** which must be of type *point*.

Used in:

contents, *interface*, *page*, and *statement*.

See also:

boolean, *constant*, *integer*, *miNoMax*, *number*, *parameter*, *point*, *string*, *typedValue*, and *variable*.

(atLeast *numberValue*)

atLeast

atLeast

The *atLeast* construct is used to specify that the range of values in the enclosing *between* or *range* constructs is greater than or equal to *numberValue*. This is needed in the context of most physical design rules to describe acceptable values for various geometric measurements.

Example:

(figureWidth metal_width
 (figureGroupObject metal)
 (atLeast 2))

This example restricts the width of all **metal** figures to be at least **2** distance units.

Used in:

between, and *range*.

See also:

atMost, *multipleValueSet*, *numberValue*, and *physicalDesignRule*.

(atMost <i>numberValue</i>)	<i>atMost</i>
	<i>atMost</i>

AtMost is used to specify that the range of values in the *between* or *range* constructs is less than or equal to *numberValue*. It is used in the context of most physical design rules.

Example:

```
(figureWidth metal_width
  (figureGroupObject metal)
  (atMost 2))
```

This example restricts the width of all **metal** figures to be at most **2** distance units.

Used in:

between, and *range*.

See also:

atLeast, *multipleValueSet*, *numberValue*, and *physicalDesignRule*.

(author *stringToken*)

*author**author*

Author is used to identify the person or organization responsible for writing the file. This field is often used for identifying ownership of the design or for identifying a contact for more information about the design.

Example:

```
(written
  (timeStamp 1987 3 4 12 34 54)
  (author "Kiki")
  (program "EDIF_$Net" (Version "5.3")))
```

The above example specifies that the block of information was produced on **March 4, 1987** at **12:34:54 UTC** by the program named **EDIF_\$Net** version **5.3**, and that the person named **Kiki** is responsible for the creation or modification of the data.

Used in:*written.***See also:***dataOrigin, program, status, timeStamp, version, and written.*

(baseArray)*baseArray**baseArray*

BaseArray is used to identify a view which describes the top level of the hierarchy defining the uncustomized portion of a gate array. A *baseArray* cell view will contain instances of *arraySite* cells which define the sites for customization as well as other information relating to the basic array.

Example:

```
(cell Cell1
  (cellType generic)
  (view View1 (viewType symbolic)
    (interface
      (arrayRelatedInfo
        (baseArray)))
    (contents ...)))
```

The above example identifies view **View1** of cell **Cell1** as a base-array cell.

Used in:

arrayRelatedInfo.

See also:

arrayMacro, and *arraySite*.

(becomes *logicNameRef* | *logicList* | *logicOneOf*)

*becomes**becomes*

Becomes is used to describe a logic state transition from any value to the given logic value.

LogicList is used in conjunction with lists or arrays of ports. When specified the size of the logic list must match the size of the port list.

LogicOneOf is used to provide a choice of logic values or logic lists. Any match in the *logicOneOf* is considered a match in the *becomes* statement.

Example:

(becomes L)

The first example specifies any transition to the logic value L.

(becomes (logicOneOf H L))

The second example specifies any transition to either the logic value H or to the logic value L.

Used in:

change, *event*, *netDelay*, *portDelay*, and *steady*.

See also:

logicList, *logicOneOf*, *pathDelay* and *transition*.

between

EDIF Version 2 0 0

between

(*between*
 atLeast | *greaterThan*
 atMost | *lessThan*
)

between

The *between* construct is used to specify a range of real number values with both an upper and lower limit for use in physical design rules as a *range*.

Example:

(*between* (*atLeast* 5) (*lessThan* 10))

This example specifies the range of numbers that are greater than or equal to **5** and less than **10**.

Used in:

range.

See also:

atLeast, *atMost*, *greaterThan*, *lessThan*, *multipleValueSet*, and *physicalDesignRule*.

(block { statement })

*block**block*

The *block* construct allows a number of EDIF statements to be collected into a group. *Block* is not restricted by *viewType*, but the enclosed *statements* must be appropriate within the context of the *block*. It may only be used in Level 2 descriptions.

Names defined within a block are local to that block. *Escape* may be used to prematurely terminate a *block*. Other than that, the meaning of the *block* is the same as the individual statements, taken in the same sequential order.

Example:

```
(block
  (variable n (integer 1))
  (while (increasing n chainSize)
    (figure POLY (dot (pt n 0)))
    (assign n (integer (sum 1 n))))))
```

This example shows the use of *block* to collect Level 2 statements which could be part of a procedural layout.

Used in:

contents, interface, page, statement, and technology.

See also:

else, escape, if, iterate, nameDef, then, and while.

```
(boolean { booleanValue | booleanDisplay | boolean })
```

*boolean**boolean*

Boolean declares a *constant*, *parameter*, *property*, or *variable* value to be of type Boolean. An optional *booleanValue* can be specified, in which case the value serves as a default.

An array of boolean values can be defined by specifying multiple *booleanValues* within a *boolean*. Nesting of the *boolean* construct is required to specify array values of two or more dimensions. The number of items within the outermost *boolean* should match the size of the first dimension of the *array*. Each nested *boolean* must have structure corresponding to the dimension of the *array*.

Example:

```
(parameter refresh (boolean (false)))
```

In this example the parameter **refresh** is declared to be of type Boolean. A default Boolean value of **(false)** is also specified.

```
(constant (array BA1 2) (boolean (true) (false)))
(constant (array BA2 2 3)
  (boolean
    (boolean (true) (false) (true))
    (boolean (false) (true) (true))))
```

In the above example the constant **BA1** is defined to be a one-dimensional Boolean array with the following value assignments:

```
(member BA1 0) = (true),
(member BA1 1) = (false)
```

and **BA2** is defined to be a two-dimensional Boolean array with the following value assignments:

```
(member BA2 0 0) = (true),
(member BA2 0 1) = (false),
(member BA2 0 2) = (true),
(member BA2 1 0) = (false),
(member BA2 1 1) = (true),
(member BA2 1 2) = (true).
```

EDIF Version 2 0 0

boolean

Used in:*boolean, borderPattern, fillPattern, and typedValue.***See also:***array, booleanValue, false, integer, miNoMax, number, point, string, and true.*

April 27, 1987

**Documents**

Document Sharing Hub

```
(booleanDisplay booleanValue
  { display }
)
```

*booleanDisplay**booleanDisplay*

BooleanDisplay is used for displaying Boolean design data. The actual form that a Boolean value has while being displayed is system-dependent and is not specified by EDIF. *BooleanDisplay* is allowed in constructs that normally contain design data and which exists in a coordinate space.

BooleanDisplay contains the data to be displayed, which must be a Boolean value and any number of *display* forms.

Example:

```
(net N1
  (joined (portRef A) (portRef B))
  (property WidthFrozen
    (boolean
      (booleanDisplay (true) (display text (origin (pt 4 6))))))...))
```

In the above example the *property* named **WidthFrozen** is in a coordinate space, and the value (**true**) can be displayed at point (4,6).

Used in:

boolean.

See also:

booleanValue, *display*, *false*, *integerDisplay*, *miNoMaxDisplay*, *numberDisplay*, *pointDisplay*, *stringDisplay*, and *true*.

(booleanMap booleanValue)

booleanMap

booleanMap

BooleanMap is an attribute of a logic value which declares whether the logic value is associated with Boolean values of *true* or *false*. Not all logic values will map into Boolean values.

Example:

```
(simulationInfo
  (logicValue T (booleanMap (true)))
  (logicValue WT (booleanMap (true)))
  (logicValue F (booleanMap (false)))
  (logicValue WF (booleanMap (false)))
  (logicValue X))
```

In the above example the logic values **T** and **WT** map to *true*. The logic value **F** and **WF** map to *false*. The logic value **X** does not correspond to any Boolean value.

Used in:

logicValue.

See also:

false, *simulationInfo*, and *true*.

April 27, 1987



Documents
Document Sharing Hub

2-27

*false | true**booleanValue**booleanValue*

BooleanValue is used to describe the primitive Boolean set. The set for Level 0 includes the Boolean values *true* and *false*. This is extended in Level 1.

Example:

```
(parameterAssign triState (boolean (true)))
```

This statement could be used in a Level 0 instance of a higher-level view to assign the value *true* to the parameter named **triState**.

Used in:

boolean, *booleanDisplay*, *booleanMap*, and *visible*.

See also:

false, *parameterAssign*, *property*, and *true*.

false | *true* | *valueNameRef* |
and | *or* | *not* | *xor* |
equal | *increasing* | *strictlyIncreasing*

*booleanValue**booleanValue*

In Level 1 EDIF descriptions, *booleanValue* is extended to incorporate any Boolean expression. This includes Boolean constants, a value reference, and functions that return Boolean values. The *valueNameRef* may refer to a *constant*, *parameter*, or *variable* definition of type *boolean*.

Example:

```
(constant b (boolean (true)))
...
(and b ...)
```

In the above example the **b** in the *and* function is a constant value reference. The *and* construct is a Boolean function.

Used in:

and, *boolean*, *booleanDisplay*, *booleanMap*, *constraint*, *if*, *not*, *or*, *visible*, *while*, and *xor*.

See also:

constant, *integerValue*, *numberValue*, *parameter*, *pointValue*, *stringValue*, and *variable*.

<i>borderPattern</i>	<i>borderPattern</i>
<i>integerValue</i>	number of pixels in X direction
<i>integerValue</i>	number of pixels in Y direction
<i>boolean</i>	boolean array containing pixel pattern
)	<i>borderPattern</i>

BorderPattern is used to specify the stipple or pixel pattern for a border or perimeter of a figure. The first integer value represents the number of pixels required for the pattern in the x direction, and the second integer value represents the number of pixels required for the pattern in the y direction. The two integer values also specify the dimensionality of the Boolean array in the same manner as the *array* construct.

The *boolean* construct is used to define an array of pattern values. It must define a Boolean value for each point of the matrix defined by the two integer values. The pixel size is not defined since it is dependent on the type of display used. The origin of the pattern is in the lower left corner, the x dimension increasing toward the right and the y dimension increasing toward the top.

The pattern values are defined such that (**member 0 0**) of the Boolean array is the pixel value at the pattern origin, (**member 1 0**) is the pixel value for the next adjacent pixel from the pattern origin in the positive x direction, (**member 0 1**) is the pixel value for the next adjacent pixel from the pattern origin in the positive y direction, and the array value for the member with maximum indices is the pixel value for the upper right corner of the pattern. A Boolean *true* indicates that the pixel is to be displayed in the specified color; a Boolean *false* indicates that the pixel is either unchanged or is displayed in the background color.

BorderPattern may be specified in the *figureGroup* form in the technology block of a library. In this case the pattern is the default border pattern to be used to outline all figures in the figure group. *BorderPattern* may also be specified in the *figureGroupOverride* form within the *figure* itself. In this case the border pattern specified is used to override the default border pattern.

The pattern only affects the display of figures, not their fabrication.

Example:

```
(borderPattern 4 6
  (boolean
    (boolean (true) (true) (false) (false) (false) (false))
    (boolean (true) (false) (false) (false) (true) (false))
    (boolean (true) (true) (true) (true) (true) (false))
    (boolean (false) (false) (false) (false) (true) (false))))
```

The above example specifies a border pattern which would fill the border with J character patterns.

Used in:

figureGroup, and *figureGroupOverride*.

See also:

array, *boolean*, *booleanValue*, *borderWidth*, *color*, *fabricate*, *false*, *figure*, *fillPattern*, *integerValue*, *true*, and *visible*.