SCALD LANGUAGE REFERENCE MANUAL

Manual Number: MN221 Rev A

10 March 1986

Valid Logic Systems, Incorporated
2820 Orchard Parkway
San Jose, CA 95134
(408)945-9400 Telex 371 9004

TABLE OF CONTENTS

SECTION 1
OVERVIEW

Logic design is supported on the SCALD system and PC-AT with a unique set of software tools and a proven methodology. One of the most important aspects of SCALD technology is the SCALD language that is used to express the logical design of an electronic circuit. Within this manual, the signal and path name syntax is defined and the concepts of properties and text macros are explained.

As with any language, the SCALD language has been developed to provide clear and concise communication between system and designer and specifically to allow logic design concepts to be expressed in a predictable and consistent manner. The language is explicitly designed to allow the definition of complex logic circuits while still retaining its user-comprehensible nature. Since the language is understood by the system, error detection, circuit analysis, and physical descriptions can be generated automatically.

In the development of the SCALD language, the following criteria were realized:

- The language is complete. That is, the language is capable of describing any logic circuit.

- The language is easy to understand. It is consistent, simple, and logical and includes no surprises.

- The language adapts to existing design conventions. Logic design conventions such as signal naming conventions and schematic layout are accommodated within the language – the language does not require the user to follow a specific design style.

- The language supports hierarchical and structured as well as flat designs.

- The language is error resistant. Constructs that are error-prone and that do not provide significant advantages in return are avoided.

- The language supports concise representations. Commonly encountered circuit elements are represented in as concise a manner as possible.

- The language does not require the addition of "special" information in a drawing that normally would not be placed on a vellum print. A drawing needs only enough information to describe the schematic.

1.1 DRAWINGS

Logic designs are entered into the SCALDsystem as drawings. A drawing is nothing more than a graphical schematic – it is the same as a schematic drawn by hand on paper. The Graphics Editor is used to create all drawings in the SCALDsystem. Drawings are used to specify all information about a schematic throughout the SCALDsystem.

To describe a schematic, the components or "parts" are specified, positioned, and interconnected with wires. Components come from libraries that define sets of parts within a logic family. Valid offers a wide range of libraries that include the TTL, ECL, and CMOS technologies. A schematic drawing is complete when all components and wires have been entered and the drawing has been written to the disk.

The Graphics Editor creates two descriptions of each drawing; a graphical description that shows the shape and placement of all parts and wires, and a description of the circuit's electrical connectivity that describes how the parts are interconnected, but contains no graphical information. The Graphics Editor is the only SCALDsystem analysis tool that knows what the drawing "looks like" and is the only tool that reads the graphical descriptions. The remainder of the analysis tools use the electrical connectivity descriptions.

The SCALD Compiler reads the drawings created by the Graphics Editor, performs error checking and hierarchical expansion, and outputs the connectivity files for use by the other analysis tools. A drawing is entered by the designer using the Graphics Editor, is compiled by the Compiler, and is packaged by the Packager into the net and parts lists required for circuit fabrication and documentation. The optional Timing Verifier and Logic Simulator analysis tools perform electrical verification of the design.

1.2 SIGNALS AND INTERCONNECTIONS

Every interconnection between two or more components represents a signal, and every signal has a name. Signal names can be assigned by the designer (using the Graphics Editor's SIGNAME command); unnamed signals automatically are given unique names by the Graphics Editor. Signal name assignment by the designer allows descriptive or mnemonic references to be used; signal names assigned by the Graphics Editor are more cryptic and are not as easily interpreted. A signal is referred to by name and can be referenced from many drawings.

The SCALD language recognizes interconnections in two ways: the direct connection of two (or more) points with a wire, or the designer's assignment of the same signal name to two or more wires (i.e., there is an implicit connection among wires of the same name). Implicit connections by signal name make it easy to interconnect components without having to use continuous wire connections that can add unnecessary complexity to a schematic. As an example, consider a clock signal that drives multiple components. While a single wire with multiple tie points can be used, labeling each clock input with the same signal name is logically and functionally identical and eliminates having to route the signal to each input.

SECTION 2
SIGNAL NAMING AND SYNTAX

2.1 SIGNAL NAMING CONVENTIONS

Signals in the SCALDsystem represent interconnections of parts. These interconnections are given names that serve to identify and distinguish them. Signals have several attributes that are specified within the signal name. These attributes are:

- The name by which the signal is known
- Its assertion level (high or low)
- The number of bits the signal represents
- The properties it possesses

A signal's name is a string of characters chosen to provide some descriptive or mnemonic reference for the signal. The name is used to identify the signal, and all signals with the same name are interpreted as being the same signal.

The assertion level describes the active state of the signal when asserted. By convention, a signal is active high for positive logic and is active low for negative logic. Two signals with the same name, but with different assertion levels are NOT the same signal.

A signal that represents a single bit is called a "scalar" signal. Within SCALDsystem, signals can represent multiple bits (i.e., a bus). Multiple-bit signals are called "vector" signals; the bit subscript portion of the signal name specifies the number of bits (and which bits) the vector signal represents. Scalar signals do not have bit subscripts. Vector signals always have bit subscripts even when the signal represents only a single bit. A signal cannot be a scalar in one instance and a vector in another; the use of a signal must be consistent.

Signals can be given properties that describe characteristics of the signal, control how the signal is interpreted by the Compiler, convey physical information, etc. Several properties are predefined by the Compiler and have special meanings. The designer can define additional properties that are passed through the Compiler to post processing programs to allow information to be added to the drawings that is not used by the other design tools, but has meaning in the user's design environment. Signals that have different properties, but are otherwise identical, are considered to be the same signal (except signals with different values for the SCOPE property; see Properties in section 3.

2.2 SIGNAL NAME SYNTAX

The complete signal name syntax can include the following parameters:

- negation symbol
- signal name
- bit subscript
- assertion symbol
- general properties

With the exception of the name, all of the other signal parameters are optional.

NEGATION SYMBOL

The negation symbol indicates that the entire signal is the negated form of a corresponding "base" signal (i.e., the complement of the signal without the negation symbol). As an example, the signal -CLOCK A is the negated form or complement of the signal CLOCK A. The default negation symbol is the "-" character (see section 2.4 and the *Library Reference Manual* for optional symbols).

SIGNAL NAME

The name portion of a signal is the "name" by which the signal is known. Within a name, a timing assertion may be included. A signal name may be made up of any characters except the following reserved characters:

bit subscript start character ('<')
general property prefix character ('\')
assertion character ('*')
signal concatenation character (':')

The following additional characters have special meanings within the signal name:

'	delimiter for a single quote string
"	delimiter for a double quote string
{}	delimiters for a comment
!	prefix for a timing assertion
$	signal class separator

A signal name is further divided into three parts: the signal class, the name string, and the timing assertion. The signal class and timing assertion are optional.

Signal Class

Signal class is an optional character string prefix that is used to identify groups or sets of related signals. As part of the signal name, signals with the same name string, but with a different signal class, are NOT identical. A signal class string must be separated from the name string by the $ character. As an example, all signals in an ALU portion of a design would be placed in the same class (i.e., the "ALU" class) by prefixing each signal with "ALU."

ALU$A=B
ALU$BUS ENABLE
ALU$CARRY IN

The signal class of a signal is ignored by the Compiler; it is interpreted as part of the name. Signal class can be used by the designer both to sort signals and to improve the readability of signal names. As will be explained later in this section, unnamed signals automatically are assigned signal class "UN."

Name String

The name string is a string of characters that form the "name" of the signal; user-assigned names are usually descriptive or mnemonic. Names may be made up of any characters except for the special characters previous described (special characters can be used if enclosed in single or double quotes). For example, the name

A*B enable

is illegal while the following names are legal:

'A*B enable'
"A*B" enable
A'*'B enable

Timing Assertions

Timing assertions are used to define the periodic behavior of a signal over a clock period. The most common use is in defining the behavior of clock signals. Timing assertions provide important data to the Timing Verifier and the Logic Simulator. They are passed on to these programs by the Compiler without being checked for syntax errors. Timing assertions are ignored by the Packager. The form of a timing assertion is described in detail in the *Timing Verifier Reference Manual.* The general form is

! *assertion_type time_specifier*

where '!' is the timing assertion prefix character, *assertion_type* is C, P, D, or S, and *time_specifier* is a clock interval or range of clock intervals.

Here are some examples:

```
CLOCK !S 4-6
CLOCK !C2-4
```

The Compiler ignores timing assertions because they are part of the name portion of the signal name. The two signals in the above example are NOT the same signal because they have unique signal names.

BIT SUBSCRIPTS

Bit subscripts are part of the SCALD structured design methodology and are used with vector signals both to specify the number of bits that a signal represents (e.g., the bit range) and to identify the bits included. Bit subscripts can be of the following forms:

<bit>
<bit .. bit>
<bit .. bit : step>
<bit : width>
<bit : width : step>
<bit list>

where *bit* is some bit number. The bit number must be equal to or greater than zero (negative bit numbers are not allowed). The '<' character marks the beginning of a bit subscript, and the '>' character marks the end. If a bit subscript does not appear in a signal name, the signal is a scalar.

<bit> Subscript

Specifies a single bit of a vector signal. Note that although such a signal represents only a single bit, it is called a vector since it represents a specific bit of a multibit (vectored) signal. Some examples:

<31> <0> <6> <5334773>

<bit1 .. bit2> Subscript

Specifies a subrange of bits from *bit1* to *bit2* inclusive. The order of the bits is determined by the signal syntax being used; the default bit order is from right to left (i.e., *bit1* is greater than *bit2*). See also section 2.4 and the *Library Reference Manual.* Some examples:

<31..0> <9..2> <7..0>

<bit1 .. bit2 : step> Subscript

Specifies a subrange of bits beginning with *bit2* and including every bit that is *step* bits apart up to *bit1* (default right-to-left bit order). The *step* value is a positive integer (a negative integer can be specified to reverse the bit order signal syntax); a *step* value of "1" is equivalent to no step value. Some examples:

<31..0:2> results in 30 28 26 ... 6 4 2 0
<11..0:4> results in 8 4 0
<9..1:3> results in 7 4 1
<0..31:-1>results in 31 30 29 ... 3 2 1 0
<15..0:20>results in 0

<bit : width> Subscript

Specifies a field of *width* bits using *bit* as the high-order bit (default right-to-left bit order) or low-order bit (left-to-right bit order). Note that *width* must be a non-zero positive integer. Some examples:

<31:8> same as <31..24>
<15:16> same as <15..0>
<0:16> same as <0..15> (left-to-right bit order)

<bit : width : step> Subscript

Specifies a field of *width* bits using *bit* as the high-order bit (default right-to-left bit order) or low-order bit (left-to-right bit order) and including only those bits that are *step* bits apart. Note again that *width* must be a non-zero positive integer, and *step* can be either a positive or negative non-zero integer. Some examples:

<31:8:2> same as <31..24:2>, results in 30 28 26 24
<0:16:3> same as <0..15:3> (left-to-right bit order), results in 0 3 6 9 12 15
<31:8:-1> same as <24..31>

<bit list> Subscript

A *bit list* is a list of any of the above forms of subscript specifiers. Each subscript specifier must be separated by a comma, and any number of specifiers may be included in the list. An example:

<1,7..4,19:8:2> results in 18 16 14 12 7 6 5 4 1

ASSERTION

In the default signal syntax, the assertion level of a signal is determined by the presence or absence of the '*' low assertion character (i.e., the presence of the low assertion character indicates that the signal is active in it low state). For a high assertion character that must be explicitly specified to define a signal that is active in its high state, see the alternate signal syntax formats described in section 4.4.

SIGNAL PROPERTIES

Signal properties are used to add information to a signal that can be interpreted by the Compiler or Packager programs. A property is a name/value pair that is used to convey almost any kind of information. For a complete description of properties, see the Section 4.

The form of a signal property when it appears in a signal name is:

\ *property name* = '*property value*'

The \ is the property prefix character, *property name* is a character string identifier, and '*property value*' is a string of characters enclosed in single quotes. Some common signal properties have been given abbreviations (with text macros) to make them easier to use. These abbreviations are:

L - gives local scope to a signal
G - gives global scope to a signal
I - identifies a signal as an interface signal
R*n* - specifies signal replication
NWC - no width check directive
NAC - no assertion check directive
WD*n* - wire delay
CD*n* - chip delay
E*n* - evaluation directive

They are used as follows:

\L equivalent to \SCOPE='LOCAL'
\WD 2.0-3.0 equivalent to \WIRE_DELAY='2.0-3.0'
\R 2 equivalent to \REP='2'

For a more complete description of text macros and their use, see Text Macro Facility, later in this manual.

2.3 THE COMPLETE SIGNAL SYNTAX

The signal name parameters are combined to form a complete signal name. To summarize, a signal name may include the following parameters:

- negation character
- name portion
- bit subscript
- assertion statement
- general properties

With the default signal name syntax, the order in which the signal name parameters must appear is as follows:

negation name subscript assertion properties

The *negation*, *subscript*, *assertion*, and *properties* parameters are optional; *name* must appear in every signal name. The following examples demonstrate the default signal syntax.

CLOCK
Active high "CLOCK" scalar signal.

-CLOCK
Negated (complementary) active high "CLOCK" scalar signal.

ENABLE*
Active low "ENABLE" scalar signal.

-ENABLE*
Negated active low "ENABLE" scalar signal.

-DATA IN <15..0>*
Negated active low 16-bit DATA IN vector signal (DATA IN 0 through DATA IN 15).

DATA OUT <2>* \WD 2.0-3.0 \L
Active low "DATA OUT 2" single-bit vector signal with 2.0-3.0 nanosecond wire delay and "local" scope.

SYSINIT* \G
Active low "SYSINIT" scalar signal with global scope.

ADDR <15..0,18> \I\WD 3.0-5.6
Active high 17-bit "ADDR" vector signal (ADDR 0-15, ADDR 18) with "interface" scope and 3.0-5.6 nanosecond wire delay.

CLK !C 0-4, 5-7
Clock signal high for intervals 0-4 and 5-7.

DATA !S 2-4
Data signal high for intervals 2-4.

Note that in the signal examples with properties, a space is required to separate the property name from its associated value and that a space also is required to separate a property value from a subsequent property name (the Compiler uses spaces to determine the beginning and end of a text macro parameter; a space is not required between the \I and \WD properties in the last example since the \I text macro does not have an associated property value).

2.4 OPTIONAL SIGNAL NAME SYNTAX

The default syntax for signal names described in the previous sections is referred to as the Valid standard library format or "Library Format 1" and is defined as follows:

negation name subscript assertion general_props

Four other formats for the signal name syntax are supported. To use these formats, the component libraries must be translated from the Valid format to the library format desired. Note that library translations normally are performed by Valid field service personnel when the system is installed. The signal name syntax for each of the optional library formats is as follows; the negation, assertion, and bit subscript order and subrange indicator for each format are outlined in Table 2-1.

Library Format 2:

negation name subscript assertion general_props

Library Format 3:

negation name subscript assertion general_props

Library Format 4:

assertion name subscript general_props

Library Format 5:

negation name assertion subscript general_props

Table 2-1. Optional Library Formats

Format Number	Low Assertion Character	High Assertion Character	Bit Order	Subrange Indicator
1*	'*'	none	right to left	'..'
2	L	H	right to left	':'
3	L	H	right to left	'..'
4	'-'	'+'	left to right	':'
5	L	H	left to right	'..'

*Valid standard default format

Note that with the formats that use the 'L' and 'H' for assertion level, a space prefix is required to avoid signal name ambiguity with scalar signal names. Also note that since Library Formats 2 and 4 use a ':' as the subrange indicator, two consecutive colons must be used to indicate a step argument for a bit subscript (e.g. *<bit1:bit2::step>*).

2.5 CONCATENATED SIGNALS

Signals can be concatenated (linked) to form signal bus structures by separating each signal name with the concatenation character (':') For example, the signals A and B are concatenated as follows:

A:B

Concatenated signals are completely unrelated; concatenation is merely a shorthand notation for two or more signals that appear together and is the same as running the signals side by side. A concatenated signal is separated back into its individual signals with a "demerge" or "tap" body (see section 3).

2.6 CONSTANT SIGNALS

A constant is a special type of signal name. The SCALD language allows constant signals to be specified in binary, octal, hexadecimal, and other number systems. The syntax for a constant signal name is the same as for other signal names, except that bit subscripts are not allowed. These fields are allowed:

neg constant_name assertion properties

Constants may have assertions although no assertion checking is performed on them unless explicitly enabled (see the *Compiler Reference Manual* for a discussion of assertion checking).

The syntax for *constant_name* is

radix # constant_value [(width)]

where *radix* specifies the number system (or base) used to specify *constant_value.* Note that *radix* must be a base 10 integer between 2 and 16. If *radix* is not specified, binary (base 2) is assumed. The '#' character separates *radix* from *constant_value.*

Constant_value is a string of digits. The legal digits are determined by the radix specified as follows:

Radix	Legal Digits
2	0 1
3	0 1 2
4	0 1 2 3
5	0 1 2 3 4
6	0 1 2 3 4 5
7	0 1 2 3 4 5 6
8	0 1 2 3 4 5 6 7
9	0 1 2 3 4 5 6 7 8
10	0 1 2 3 4 5 6 7 8 9
11	0 1 2 3 4 5 6 7 8 9 A
12	0 1 2 3 4 5 6 7 8 9 A B
13	0 1 2 3 4 5 6 7 8 9 A B C
14	0 1 2 3 4 5 6 7 8 9 A B C D
15	0 1 2 3 4 5 6 7 8 9 A B C D E
16	0 1 2 3 4 5 6 7 8 9 A B C D E F

Width explicitly specifies the number of bits used to express the constant (in the specified radix) and must be greater than zero. If *width* is omitted, it is calculated from the number of bits per digit (radix) and the number of digits in the constant. Note that *width,* if specified, must be enclosed in parentheses.

Radix	Bits per Digit
2	1
3 4	2
5 6 7 8	3
9 10 11 12 13 14 15 16	4

Example constants:

Constant	Value (base 10)	Number of Bits
0101	5	4
2#0000	0	4
0000(3)	0	3
10#0	0	4
16#FFFF	65535	16
8#377	255	9
8#377(8)	255	8

2.7 PATH NAME SYNTAX

Path names are used to uniquely identify every component within a design. Just as every wire has a signal name, every component appearing within a design has a unique path name. Since the same component can be used many times within a design, the component path name must be more specific than just the name of the component. Path names are assigned exclusively by the SCALD system based on the path followed from the root drawing down through the hierarchy to each individual component. Users may not specify their own path names. Path names are used by the Compiler (the Compiler listing file *cmplst.dat* references path names in its error reports), the Logic Simulator (opening signals with the same name within a design are resolved with path names), and by the Packager.

The path name itself is made up of path element names for each of the bodies encountered in the path from the root drawing to the component. A path element name is created by the Compiler for each body within a design. The path element name syntax is:

page abbreviation path [*size*] [*unique_number*]

PAGE

The number of the drawing page on which the body appears. If the page number is '1,' the *page* reference is omitted from the path element name.

ABBREVIATION

An abbreviation for the drawing name. The abbreviation is normally assigned by the user by attaching the ABBREV property to the DRAWING body within the drawing. If an ABBREV property is not assigned to a drawing, the Compiler creates an abbreviation by truncating the drawing name. (Library components are preassigned an ABBREV property.) When *abbreviation* begins with a number, a period ('.') is prefixed to the abbreviation as a delimiter to separate *abbreviation* from *page* even if the page number is '1' (omitted). For example:

.1ALU = page 1 of drawing 1ALU
2.1ALU = page 2 of drawing 1ALU
2ALU = page 2 of drawing ALU

PATH

The value of the PATH property attached to the body. Unique PATH properties are automatically assigned to each body in a drawing by the Graphics Editor when the drawing is written. PATH properties can also be assigned by the user with the Graphics Editor's PROPERTY command. When PATH is automatically assigned by the Graphics Editor, it takes the form

*integer*P

as in:

37P 85P 4P

If *abbreviation* ends with a number, a period ('.') is prefixed to *path* as a delimiter.

SIZE

The value of the SIZE property attached to the body for size replication. If the SIZE value is greater than 0, the value is included in the path element and prefixed by a '#' character; if the SIZE value is 0 or not specified (i.e., no size replication is to be performed), *size* is omitted.

UNIQUE_NUMBER

If the combination of the *page*, *abbreviation*, *path*, and *size* values does not form a unique path name element (e.g., vectored signals with size replicated components), an incrementing number, prefixed by a colon (':') is added following the *size* value to make each path name element unique.

PATH ELEMENT NAME EXAMPLES

The path element name for a 74LS74 with an assigned PATH property of 34P that appears on page 3 of a drawing is:

3LS74.34P

In the above example, note the period delimiter between LS74 (the abbreviation for an 74LS74 device) and the PATH property (34P). A more complex path element name is:

.2DAC8P#8:3

This path element name is for a body on page 1. Note the omission of a page number at the beginning; the leading period indicates that the body drawing abbreviation begins with a numeral (2DAC). The body has an assigned PATH property of 8P (a period delimiter is not required between the ABBREV and PATH properties since the ABBREV property ends with a character). The '#8' indicates a size replicated part, and the ':3' makes the path name element unique among the eight size-replicated bodies.

PATH NAME

In a hierarchical design, the path name for any component within a design is constructed by concatenating each path element name for each body in the hierarchical path from the root drawing down to the component. Path names are enclosed in parentheses ( ). Each individual path element name is separated by a space. An example of a path name using the two path element names described above is:

(CONVT .2DAC8P#8:3 3LS74.34P)

This example describes a 74LS74 that appears on page 3 of the drawing "2nd DAC stage" (abbreviated as 2DAC) which itself appears as a body on page 1 of the root drawing "converter" (abbreviated as CONVT). Note that since "CONVT" is the root drawing, there is no corresponding body and no page number or PATH or SIZE property.

2.8 SIGNAL SYNONYMS

When a signal has more than one signal name, the signal names are said to be synonymous (i.e., the names all reference the same signal). Synonyms are useful for creating locally-meaningful names for signals known throughout a design. Synonyms also provide a means of interconnecting nets. As an example, the two nets A and B can be connected together by simply synonyming the signals A and B.

The easiest way to create a synonym is to assign two signal names to a single wire. Synonyms also are created whenever more than one signal is connected to the same pin. A SYNONYM body is provided in the Standard Library (see the *Library Reference Manual*) that can be used to create a synonym. The SYNONYM body appears as three parallel lines; the center line has two common pins (one at each end). Every signal connected to the pins of the synonym body will be synonymed together.

When two signals are synonymed, the Compiler selects one of the signal names as the "base" signal. The Compiler's expansion file (cmpexp.dat) contains only base signals. The synonyms file (cmpsyn.dat) lists all of the signals in the design and their corresponding base signal name. A signal is its own base signal if is is not synonymed to any other signals or if it is selected as the base signal. The rules for selecting a base signal are as follows. The rules are applied in the order listed; if base signal name selection cannot be determined by one rule, the next rule is applied.

1. Select the lower bit number of two signals with the same name (e.g., X<0> is selected over X<3>).

2. Select a constant signal over a non-constant signal.

3. Select a signal with name properties over a signal without name properties (e.g., CLOCK !C 0-4 is selected over CLOCK).

4. Select the signal with the most global scope.

5. Select the root-level interface signal.

6. Select a user-assigned signal name over an "unnamed" or 'NC' signal.

7. Select a scalar signal over a vector signal.

8. Select the signal that is lexicographically smaller (e.g., CLK is selected over CLOCK).

The synonyms file is described in detail in the *Compiler Reference Manual.*

2.9 SIGNALS OF UNDETERMINED WIDTH

Some signals in a design do not have an explicit width specification. The width of these signals must be determined from context; that is, the widths are determined by how they are used. The most common example of signals with undetermined widths are NC and unnamed signals.

NC and unnamed signals are special signals. An NC signal is used to specify an unconnected pin. The use of this signal serves to make the drawing easier to understand. It is inconvenient to have to specify the width of an unconnected pin so the Compiler coerces the width of the signal NC to the width of whatever pin it is connected to. Unnamed signals are created by the Graphics Editor.

For both NC and unnamed signals, the Compiler determines the signal width from context. Most of the time, the Compiler sets the width of the signal to the width of the pin to which it is connected. In some cases, however, the pin itself has no width (as in the case of a MERGER or NOT body). In these cases, the Compiler must search further to find the width of the signal. Typically, all the "plumbing" bodies in a drawing must be processed before the widths of all the unnamed and NC signals can be determined. If the width of a signal cannot be determined, an error message is printed.

Signals with unknown widths are given the bit subscript <UNDEFINED> when printed to indicate to the designer that the width could not be determined. In such cases, the designer can specify the width in one of the following ways:

1. Give the signal a name with the width specified (in the bit subscript).

2. Use a SLASH body to specify the width of the signal.

2.10 SIGNALS OF UNDETERMINED ASSERTION

The assertion of unnamed signals must be determined from context as is the width. The assertion of the signal is determined whenever the signal is connected to a pin with a known assertion or when the signal is synonymed to a signal with a known assertion. Some pins do not have known assertions. These pins have the NAC (no assertion check) property which causes the assertion of the pin to be inherited from the signals connected to it. The existence of the NAC property forces the Compiler to look at another body to find the assertion of a signal.

Most "plumbing" bodies have pins with the NAC property. For this reason, the Compiler processes all of the "plumbing" bodies first in order to determine the assertions of the unnamed signals in the design. If an unnamed signal is connected to pins of conflicting assertions, the Compiler generates an error message.

2.11 ADVANCED SIGNAL NAME TOPICS

All nets in a design are named. Many of these names are assigned by the user while others are assigned by the Graphics Editor. The Compiler uses the name given a net to refer to the net. If there is more than one name for a net, the Compiler selects one name for the net and outputs a list of synonyms (or aliases) for that name so that the other names are known. Special cases of signal names will be described below.

UNUSED PIN NAMES

In a number of instances, a pin on a body will be intentionally left unconnected by design. Any unconnected pins are tied to the signal NC (no connect). There are cases where this can lead to problems. For instance, assume a drawing has two bodies connected with a signal that happens to also be an interface signal (a pin of the body corresponding to the drawing). Normally, the Compiler replaces the pin name as used in the drawing with the name of the signal connected to the pin. This means that a global signal name is propagated down into the design via the pins it is connected to. This makes it very easy to trace a signal and reduces confusion caused by preserving pin names (which aren't real signals in any case). In the case mentioned above, the real signal is NC (no connect). Substituting NC for the pin name in the drawing disconnects the two bodies since the signal is now a non-connected signal. To prevent this from happening, the Compiler creates a new signal name when it encounters an unconnected pin name.

The new signal name is created as follows:

1. Start with PINNAME$.

2. Append the pin name.

3. If the name is not unique, make it unique by appending a number preceded by a '$'.

For example, the pin name SETA would be transformed into the signal PINNAME_$SETA. This signal has the same scope as the pin name; that is, it is known throughout the subtree below the drawing.

NC SIGNALS

Occasionally, a signal (or pin) is to be left unconnected. To make this clear in a drawing, it can be given the special signal name NC. The Compiler (and the rest of the SCALD-system programs) understand that NC is a special signal name. It has the following characteristics:

- Unlike other signals, nets with the name NC are not connected together; each one is considered to be a unique net.

- The signal NC has no particular width: it assumes the width of whatever it is connected to.

- The signal NC has no particular assertion: it can be connected to bubbled as well as non-bubbled pins.

- NC can be given an explicit width through the use of the replication operator (\R n).

If a pin is left unconnected, the Graphics Editor automatically assigns the signal NC to it. See the section on undetermined width signals for a complete discussion of NC signal width.

UNNAMED SIGNALS

The Graphics Editor names each net that is not given a name by the designer. The name is selected as follows:

1. Start with UN$.

2. Append the page number of the drawing in which the net is found followed by a '$'.

3. Find all the bodies that the net connects to. Sort the names of the bodies alphabetically and select the first in the list. Append this name followed by a '$'.

4. Append the value of the PATH property attached to the body (selected above) followed by a '$'.

5. Append the name of the pin of the body (selected above) that the net connects to.

6. If the signal so constructed is not unique within the drawing, append a number, prefixed with a '$', to make it unique.

For example, an unnamed signal on the third page of a drawing that is connected to an LS00, an LS138, a DAC1, and a MUX3 would be named as follows:

UN3$DAC1$

If the pin that the signal connects to on the DAC1 body is ENABLE and the DAC1 body has the PATH property 31P, the signal would be named as follows:

UN3$DAC1$31P$ENABLE

Finally, if this signal name is not unique, it is made unique:

UN3$DAC1$31P$ENABLE$2

This is the complete form of the signal name. If the body name or the pin name have special characters (those with special meanings in signal names such as '<', '>', '\', ':') the body or pin name is placed in quotes. For instance, if the pin on the DAC1 body was ENABLE\1, the signal name would be formed as:

UN3DAC1$31P$"ENABLE\1"$2

The Graphics Editor does not assign an assertion to the unnamed signal. This is because, in general, the assertion of a signal cannot be determined from the drawing alone; it can only be determined by processing the synonyms and "plumbing" bodies.

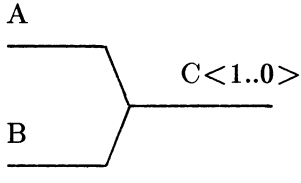
Unnamed signals have no particular width; they assume the width of the pins of the bodies to which they are connected. See the section describing signals of undetermined width for details.

SECTION 3
SPECIAL SCALD BODIES

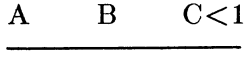
There are several special bodies that can be used in SCALD drawings for signal manipulation. These bodies are used to make it easier to express a design concisely. The bodies can be found in the SCALD Standard library in the *Library Reference Manual.* The SCALD system does not treat these bodies as special; each one is defined in terms of more basic concepts. The designer may create special bodies that behave in exactly the same manner as the ones provided.

3.1 PLUMBING BODIES

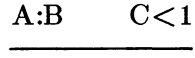
Plumbing bodies are used to combine (concatenate) signals into a single (vectored) signal or to separate a vectored signal into individual signals. A "plumbing" body looks like a wire or wires and simulates interconnections. For example, if two signals are to be combined into a bus, the user might draw a structure as follows:



The signals A and B are combined into the bus C<1..0>. Combining signals in this manner is called merging. If the above structure is drawn with wires, the result is the same as drawing:



The signals A, B, and C<1..0> are synonymed together. This is not what was meant at all, and is an error besides (since the widths of the signals are different). The intended function could be drawn as:



In this example, the signal A:B (the concatenation of the signals A and B) is synonymed to the signal C<1..0>. This is precisely the function desired for merging.

The last example, however, does not give a good graphical representation of the function of merging. What is needed is to define a body that looks like the merge function and has the same function. This is done in a family of bodies called 2 MERGE, 3 MERGE, 4 MERGE, etc. that merge 2, 3, or 4 etc. signals into one signal. The definition of the MERGE is a synonym of the concatenation of all of the input signals to the output bus. MERGERs are described further below.

"Plumbing" bodies are special in that the Compiler processes them before processing other bodies in a drawing to resolve widths and assertions of signals whose widths or assertions are unknown (this is discussed in detail elsewhere). A "plumbing" body is identified by the presence of the NWC (no width check) property on a pin of the body. The NWC property indicates that the pin has no known width; the width of the pin is determined by the widths of the signals connected to it. Another method of specifying a "plumbing" body is the presence of the BODY_TYPE="PLUMBING" body property. A "plumbing" body can not contain any primitives.

The designer may create "plumbing" bodies that follow design conventions already being used. In this manner, SCALD system drawings can be customized

3.2 BODIES IN THE STANDARD LIBRARY

The bodies in the following sections are included in the Standard Library; for additional information, see the *Library Reference Manual.*

MERGERS

Mergers are used to combine several signals into one signal. The result of merging several signals is a single signal that is the concatenation of the input signals. An equivalent signal can be created by explicitly specifying the concatenation of the input signals. For instance, the signals A, B, and C can be merged together to form the signal A:B:C with a merger. The output signal is equivalent to A:B:C. If the output signal were named A:B:C, the merger would not be needed. The merger provides a graphical representation for combining signals. For a more detailed explanation, see under Plumbing Bodies below.

DEMERGERS

Demergers are simply mergers turned around. The input signals are now output signals, and the output signal is now the input signal. A demerger is used to separate a signal into several pieces. In fact, a demerger is exactly the same as a merger. Its use determines its meaning; that is, if a merger is used to combine signals, it is a merger; but if it is used to separate signals, it is a demerger. For a more detailed explanation, see under Plumbing Bodies below.

NOT

The NOT body is used to change the logic convention of a signal. If a signal is asserted low, it is considered to be a negative logic signal. If a signal is asserted high, it is considered to be a positive logic signal. The NOT body is used to change the logic convention of a signal without introducing an actual logical inversion. That is, the state of the signal is not changed, it is just considered to be of the opposite logic convention. The consistent use of logic

conventions makes designs easier to read and understand. The NOT body is used as an escape in those cases where strict adherence to a logic convention is not possible. The NOT body is a notational assistance and does not affect the physical implementation of the circuit. The NOT body must be used if bubble checking is to be performed.

SLASH

A SLASH body performs two useful functions. First, it is used to document the widths of signals. Normally, a signal's width is apparent from its bit subscript, but occasionally the signal name is not present or visible where the signal is used. In these cases, it may not be clear what the signal's width is. The SLASH body is used to remind the designer. The Compiler always checks the signal's actual width with that specified on the SLASH body.

The second use of the SLASH body is to specify signal widths that are not otherwise specified. NC and unnamed signals, for instance, have no particular widths. The Compiler assigns widths to these signals, but, when the Compiler is unable to determine the width, the SLASH body can be used to specify it.

SYNONYM

The SYNONYM body is used to tell the Compiler that two signals with different names are to be considered the same signal. For more information, see below under Signal Synonyms.

TAP

The TAP body is used to select a portion of a signal while leaving the original signal unchanged. In this fashion it is different from a merger which splits a signal into several pieces. A tap can be used to select either the most significant or least significant portion of the input signal. The number of bits selected by the tap is specified with a SIZE property attached to the TAP body.

SECTION 4
PROPERTIES IN THE SCALD LANGUAGE

This section introduces the concept of a property. Properties serve important and varied functions in the SCALD system. They are used to convey a wide range of information about the design and to control analysis processes. A property is a name/value pair that can be attached to certain objects in a design to convey almost any information. A number of predefined properties are used by the SCALD system to record information needed by the Timing Verifier, the Simulator, and the Packager. Other properties can be defined by the user to convey information to design programs, or to be passed through the SCALD system to other systems (such as simulators, physical design systems, etc.).

Properties also provide a mechanism for adding physical information to drawings (which represent only a logical design), that can be passed on to the Packager and other physical design systems. With the ability to define and use properties, the designer can customize the SCALD system to fit into an existing or evolving CAD system.

A property consists of a name by which the property is known (property name) and an associated value (property value). Properties can be attached to certain objects on any drawing in the Graphics Editor. Property name/value pairs can be attached to bodies, signals, and pins. Properties can also be attached to an entire drawing by attaching them to a DEFINE body or a DRAWING body on that drawing.

Properties are then passed along to the Compiler in the editor output files. From the Compiler they are passed to all the other SCALD system programs (as well as programs written by the user) in the Compiler expansion output file. (Some properties can be filtered out by the Compiler to reduce file size.)

4.1 WHAT IS A PROPERTY?

A property is a name/value pair assigned to a particular object. The property name is an identifier, that is,

> a string of not more than 16 characters that includes letters, digits, and '_'(underscores) and starts with a letter.

Some examples of property names are:

SIZE
TIMES
MY_PROP_NAME
THE_40TH_NAME
SATURDAY1027
COST_OF_PART
PIN_NUMBER
PART_NAME

Notice that the underscore is used instead of a space. Spaces are not allowed in property names because a space delimits a property name from a property value.

Many properties have been defined by Valid for use in the SCALD system and have a specific meaning. Each of these are described, one to a page, in alphabetical order, later in this section.

A property value is associated with each property name. The property value is a string of up to 255 printing characters. Property values can be empty. Property values should be enclosed in single quotes when a property is added to a signal name (and text macros are not used). Property values need no quotes when a property is added using the PROPERTY command.

Here are some representative property values:

1
25oct82 10:31:46.03
(SIZE + 4) / 5 + 35 MOD A
This is a long property value
Property value with special chars !@ #$%*( ) ~}{[] ><

A property always consists of the property name and its value.

4.2 SPECIFYING PROPERTIES

Properties are specified with the Graphics Editor. The Graphics Editor ignores double quotes ("), it does not pass them on to the Compiler. Where a double quote is needed, use two single quotes ('') instead.

There are basically two ways of adding properties to drawings; using the PROPERTY command or including the property in a signal name. The two methods are used in different situations. Body properties are always added using the PROPERTY command. Signal properties are usually included in a signal name, but can also be added to the signal using the PROPERTY command. Pin properties are usually included in the pin name, but can also be added using the PROPERTY command. A pin property can also be inherited by a pin from a signal connected to the pin.

The meaning of the properties is the same regardless of the method used to assign them.

THE PROPERTY COMMAND

The PROPERTY command of the Graphics Editor is used to specify a property name and its value and to attach the property to an object in the drawing. Properties are attached to the origin of an object. Any property value can be entered, except one with leading spaces. Double quotes are ignored by GED.

A property specification appears as

name value

where *name* is the property name and *value* is the property value. When displayed on the drawing, the property appears as:

NAME=VALUE

PROPERTIES WITHIN SIGNALS

The other method of specifying properties is to include them as part of signal names. Commonly used properties can be added to a signal name using predefined text macros. Other properties included in a signal name take the form:

\NAME='VALUE'

For more information on the exact syntax used for adding properties to signal names, see below under Signal Properties.

The user can define additional text macros to support other properties. See "Text Macro Facility," in Section 5.

SIGNAL PROPERTIES

The syntax for a property in a signal name is

\NAME='VALUE'

where the backslash (\) denotes the start of a property. Note that the property value appears in single quotes. This is recommended to unambiguously identify the beginning and end of the property value. Do not use double quotes (GED restriction).

Text macros are used to make it easier to specify properties. For example, the property "SCOPE" specifies the scope of a signal. It can assume the values "LOCAL," "GLOBAL," or "INTERFACE." Instead of adding the property SCOPE='LOCAL' or SCOPE='GLOBAL' to a signal, text macros for each are predefined for the Compiler in a file. These text macros are:

```
L = 'SCOPE="LOCAL"'
G = 'SCOPE="GLOBAL"'
I = 'SCOPE="INTERFACE"'
```

When used in a signal name, the text macros are expanded by the Compiler into the proper form. For example

```
CLOCK * \I
```

is expanded to:

```
CLOCK * \SCOPE="INTERFACE"
```

To define additional global text macros for property name/value pairs, see section 5. The designer may define additional text macros with a DEFINE body in a drawing (see section 5).

Text macro parameters can be used to create property name/value pairs whose values need to be assigned on an instance by instance basis. For example, assume a property called LENGTH which can take on many values. A global text macro definition of the property might be

```
LEN = 'LENGTH="%1"'
```

where %1 refers to the first text macro parameter (text macro parameters are separated by spaces). When used in a signal name, the value of the property is placed after the text macro as follows

```
\LEN 2
```

which expands to:

```
\LENGTH=2
```

This is the manner in which Timing Verifier properties are supported. For example, a wire delay may be added to a wire as follows

```
SIGNAL* \WD 2.0-5.6
```

which is equivalent to:

```
SIGNAL* \WIRE_DELAY='2.0-5.6'
```

This text macro is globally defined as follows

```
WD = 'WIRE_DELAY="%1"'
```

where %1 references the first parameter of the text macro.

Other standard signal name properties are supported with built-in, reserved text macros. See the text macro section for a complete description.

For a complete description of the syntax for a SCALD signal name, see section 2, *Signal Naming and Syntax.*

Properties on signal names are passed through the Compiler to the output expansion file. In this way, properties are available to the Packager, the Timing Verifier, the Logic Simulator, and any user provided programs.

Properties are associated with specific bits of the signal. Different bits of a multi-bit signal can have different properties.

4.3 PIN PROPERTIES

Properties can be attached to pins three ways. They may be attached by adding the property to the pin with the PROPERTY command of the Graphics Editor. They may be included as properties of the pin name for the pin. They may also be inherited from signals connected to the pin.

ADDING PIN PROPERTIES WITH THE PROPERTY COMMAND

PIN properties can be added to pins of bodies with the Graphics Editor PROPERTY command. The user points to the pin and specifies the property name and value to be assigned to the pin. Default properties may be attached to pins in the body drawing. The most common example of a default pin property is PIN_NAME which is used to specify the logical name of the pin.

ADDING PIN PROPERTIES AS PART OF THE PIN NAME

Each pin of a body has a pin name which serves to identify that pin. A pin name can have signal properties just like any other signal; they are included in the signal name in exactly the same manner. Some examples of pin names with pin properties:

```
DATA INPUT \NAC
CLOCK* \PIN_NUMBER='2'
OUTPUT \OUTPUT_TYPE='(TS,TS)'
```

PIN PROPERTIES INHERITED FROM SIGNALS

Properties can be inherited from the signals that are connected to them. The most common example of a property of this type is WIRE_DELAY. This property is assigned as a signal property, but since it has a special inheritance attribute (see the section on property attributes), it is copied from the signal to the pin to which the signal is attached.

4.4 PROPERTY ATTRIBUTES

A property attribute is used to control property processing within the Compiler. Every property is given some attributes by default, and the user can modify or add to these. Attributes are assigned in property attributes files read by the Compiler. There is a Valid supplied attributes file that is always read by the Compiler to assign attributes to

standard SCALD properties. The user may supply an additional property attributes file specified by the PROPERTY_FILE directive.

The attributes file contains a list of property names and associated attributes. The file has the following form

```
FILE_TYPE = ATTRIBUTES;
property name : attribute specification ;
      .
      .
      .
END.
```

where *property name* is the name of the property, and *attribute specification* is a list of attributes for the property.

The attributes that can be assigned to properties are:

parameter	→	used on body properties only
inherit	→	controls property inheritance
permit	→	permission for property attachment
filter	→	removes property from output files

These are described in the following sections.

PARAMETER ATTRIBUTE

The PARAMETER attribute is used to make the name and value of a body property known within the drawing corresponding to the body. Normally, body properties are attached to bodies and pass through the Compiler to other analysis tools. For example, the LOCATION property is used to specify the LOCATION name for a physical component. This property means nothing to the Compiler; it passes it through to the Packager, which uses it to guide its package allocation.

Some properties, however, are used to pass information into the drawing. The most common example of such a property is SIZE. This property is used to convey information about the number of bits the body represents. This information is needed by the drawing. To make it available, the SIZE property is given the PARAMETER attribute. This causes two things to happen. First, the name of the property, and its value, are available in the drawing. It can be used as though it were a text macro. Second, any text macros within the property value are expanded.

For example, assume the WIDTH=45*X property is attached to the ELAN body. If the WIDTH property has the PARAMETER attribute, its value will be known within the ELAN.LOGIC drawing. It can be used, for example, in signal names:

SIGNAL_WITHIN_ELAN <WIDTH-1..0>

Further, the value of the WIDTH property is expanded by the Compiler. Its value is '45*X' where 'X' is some text macro (assume, for the purpose of example, that X=2). The WIDTH property value is expanded to be: "45*2." The signal name shown above then becomes:

SIGNAL_WITHIN_ELAN <45*2-1..0>

If the WIDTH property does not have the PARAMETER attribute, the Compiler will generate an error since WIDTH will be undefined.

Some properties that are to be parameters are used to pass information about a primitive component through to some analysis tool. For example, the TIMES property is attached to components to specify to the Packager that additional versions of the component should be generated. It may be necessary for the value of the TIMES property to be related to other design information; for example, the SIZE of the component:

TIMES=SIZE*2

If the TIMES property has the PARAMETER attribute, the Compiler substitutes the value of the SIZE parameter (the SIZE property has the PARAMETER attribute by default). If SIZE=1, the TIMES property value becomes "1*2." The Packager, on the other hand, is expecting the TIMES property to have an integer value; it won't accept "2*1." The Compiler can be told to completely evaluate the property's value by giving it the PARAMETER(INTEGER) attribute. The Compiler expands text macros within the property value and evaluates the property value as an integer expression (an error is generated if the property value is a malformed integer expression). The TIMES property shown above is then output with the value "2." In all other respects, the PARAMETER(INTEGER) attribute behaves just like the PARAMETER attribute.

A property can be given the PARAMETER attribute in one of two ways: the property can be specified as a PARAMETER when creating the body in the Graphics Editor, or the property can be given the PARAMETER attribute in the property attributes file. To give the property the PARAMETER attribute in a body drawing, append a \PARAMETER to the end of the property value. For example, the following property (attached to a body) has the PARAMETER attribute:

ELAN="value for ELAN\PARAMETER"

The property ELAN has the PARAMETER attribute only for this particular instance. If the property ELAN appears anywhere else (and does not have the \PARAMETER appended), it is not a PARAMETER. The second way to give a property the PARAMETER attribute is with the property attributes file. When given the attribute via the attributes file, the property has that attribute everywhere it is used (regardless of whether it has the \PARAMETER appended or not).

INHERIT ATTRIBUTE

A property may appear on an object automatically when objects become related or attached in some manner. This copying of properties from one object to another is called property inheritance. Inheritance can be controlled with the INHERIT attribute. Inheritance behavior for a particular property can be controlled for three independent contexts: BODYs (DRAWINGs), SIGNALs, and PINs. These will be discussed separately.

Body Property Inheritance

Inheritance of body properties is controlled with the INHERIT(BODY) attribute. When a property is attached to a body, it may be inherited down the hierarchy to appear on all of the bodies within the drawing corresponding to the body. For example, if the X property is attached to the ELAN body, and the X property has the INHERIT(BODY) attribute, every body within the ELAN.LOGIC drawing will have the X property. If the X property does not have the INHERIT(BODY) attribute, the X property only appears on the bodies to which it is attached (the ELAN body in this example).

Properties attached to the DRAWING body within the ELAN drawing will also inherit to every body within the drawing if the properties have the INHERIT(BODY) attribute. In this manner, properties attached to the body for ELAN and those attached DRAWING body within ELAN are processed in the same manner.

Care should be taken when using properties on bodies. If the properties all have the INHERIT(BODY) attribute (which all properties do by default), they all appear in the Compiler's output files. For example, if each drawing is given the property ENGINEER to specify the responsible engineer, and there are seven levels of hierarchy and five pages to every drawing, the primitives produced by the Compiler will each have 35 ENGINEER properties attached. This is probably not desirable. It can be corrected by removing the INHERIT(BODY) attribute from the property ENGINEER.

Signal Property Inheritance

Inheritance of signal properties is controlled with the INHERIT(SIGNAL) attribute. When a property is attached to a signal, it may be inherited by other signals synonymed to it. For example, if the signal RESET has the X property and RESET is synonymed to the MCLEAR signal, the MCLEAR signal will get the X property if the X property has the INHERIT(SIGNAL) attribute. Since MERGERs, NOTs, and all other plumbing bodies are implemented with synonyms, this attribute allows properties to move along a net within a drawing.

Properties with the INHERIT(SIGNAL) attribute are considered to be properties of the net (since all of the signal names for the net will have the properties). These properties are output by the Compiler as properties of the net and are available for processing by the Packager, DIAL, etc. Properties without the INHERIT(SIGNAL) attribute are properties of a particular signal and not the entire net. These properties are not output from the Compiler. SCOPE is one such property. It is a property of a particular signal and should not be inherited by the entire net. In general, the user will never create a signal property without the INHERIT(SIGNAL) attribute.

All properties are given the INHERIT(SIGNAL) attribute by default.

Pin Property Inheritance

Inheritance of pin properties is controlled with the INHERIT(PIN) attribute. When a property is attached to a pin, it becomes a property of that pin. If it has the INHERIT(PIN) attribute, it inherits to the interface signal for the pin. Once on a signal, an INHERIT(PIN) property is copied to other pins. A property with the INHERIT(PIN) attribute is automatically given the INHERIT(SIGNAL) attribute.

For example, suppose the X property is attached to the A pin of the ELAN body. Within the ELAN.LOGIC drawing, the signal A\I appears and connects to the B pin of a EREG

body. If the X property has the INHERIT(PIN) attribute, it will first appear as a property of the A\I signal and finally a property of the B pin of the EREG body.

The WIRE_DELAY property has the INHERIT(PIN) attribute by default (assigned in the system-wide property attributes file). When used as a signal property (with the WD text macro) as follows:

CLOCK \WD 5.0-6.0

It appears as a signal property, but is copied to each pin connected to the CLOCK signal. It is then available as a pin property and can inherit deeper into the hierarchy. Properties with the INHERIT(PIN) attribute are stripped from all signals at the end of processing.

Properties are not assigned the INHERIT(PIN) property by default; this attribute must be assigned in the property attributes file.

Summary of the INHERIT Attribute

The INHERIT attribute is assigned as INHERIT(PIN), INHERIT(SIGNAL), and/or INHERIT(BODY). If a property can be inherited by more than one object, a list can be used:

INHERIT(PIN,SIGNAL)

INHERIT() can be used to remove all inheritance attributes for a property.

PERMIT ATTRIBUTE

The PERMIT attribute is used to control the objects to which a property may be attached. It is possible to accidentally attach a property to the wrong object. If the property does not have permission to be attached to that object, an error is generated by the Compiler. Permission can be granted for a property to be attached to a BODY, PIN, or SIGNAL (WIRE) with the PERMIT(BODY), PERMIT(PIN), or PERMIT(SIGNAL) attributes.

The SIZE property, for example, has the PERMIT(BODY) attribute since it is an error for it to be attached to any other object. The SCOPE property has the PERMIT(SIGNAL) attribute. By default, a property has all three attributes.

If a property is to be given permission to be attached to more than one object, the PERMIT attribute can be specified with a list:

PERMIT(PIN,SIGNAL)

Permission for a property to be attached to all objects can be removed with PERMIT(). Such a property cannot be attached to any object and is, therefore, useless.

FILTER ATTRIBUTE

The FILTER attribute prevents a property from appearing in the Compiler's output files. For example, the LAST_MODIFIED property, which specifies the date on which the drawing page was last written, is filtered from the output by default since it is normally of no interest to analysis programs. The intent of supporting filtering is to reduce the size of design files and to reduce superfluous properties.

The FILTER attribute is assigned in the property attributes file. The user can override this attribute with the PASS_PROPERTY directive (which causes the property to be output regardless of its attributes). The FILTER_PROPERTY directive can be used to filter properties from the output even if they do not have the FILTER attribute.

USER PROPERTY ATTRIBUTE FILE

The user can supply a property attributes file by using the PROPERTY_FILE directive in the Compiler command file. The attributes assigned in the user attribute file override the attributes assigned in the system-wide attributes file. Care should be taken to not override important attributes.

The only safe attribute is the FILTER attribute. All other attributes should be left unchanged; they are assigned as required for the SCALD system.

DEFAULT PROPERTY ATTRIBUTES

The default attributes for a property are:

PERMIT(SIGNAL,PIN,BODY),INHERIT(SIGNAL,BODY);

These may be changed in the property attributes file. The following SCALD properties have attributes assigned (as shown) within the Compiler and cannot be changed by the user (if the property is normally used via a text macro, the text macro name appears as a comment):

SIZE:	inherit(),permit(body),parameter(integer);
TIMES:	inherit(),permit(body),parameter(integer);
PATH:	inherit(),permit(body);
REPLICATION:	inherit(),permit(signal); { \R }
TITLE:	inherit(),permit(body);
EXPR:	inherit(),permit(body);
VERSION:	inherit(),permit(body);
ABBREV:	inherit(),permit(body);
SCOPE:	inherit(),permit(signal); { \I \L \G }
PART_NAME:	inherit(),permit(body);
TERMINAL:	inherit(),permit(body);
NEEDS_NO_SIZE:	inherit(),permit(body);
HAS_FIXED_SIZE:	inherit(),permit(body);
WIRE_DELAY:	inherit(pin),permit(pin, signal);
NOWIDTH:	inherit(),permit(signal), { \NWC } filter;
NOASSERT:	inherit(),permit(signal), { \NAC } filter;
BODY_TYPE:	inherit(),permit(body);
X:	inherit(),filter;
X_FIRST:	inherit(),filter;
X_STEP:	inherit(),filter;

AN EXAMPLE PROPERTY ATTRIBUTES FILE

The following example is used to demonstrate the syntax and form for the property attributes file. The list of standard property attributes in the previous section is also, except for FILE_TYPE, a legal attributes file.

```
FILE_TYPE = ATTRIBUTES;

CLOCK: inherit(signal);
STABLE: inherit(signal);
WIRE_DELAY: inherit(pin);
EVAL: inherit(pin);
CHIP_DELAY: inherit(pin);

END.
```

4.5 DRAWING PROPERTIES

Properties may be "attached" to an entire drawing by attaching them to a special body called DRAWING. These properties are used to convey standard information about the drawing itself. The Compiler understands a few standard property names. These properties are:

TITLE	= title of the drawing.
ABBREV	= abbreviation for the drawing.
EXPR	= selection expression (if the drawing has >1 version).
PART_NAME	= name of the primitive part if this is a primitive drawing.
TERMINAL	= indicates that the drawing is a terminal drawing in the expansion but is not a primitive component.

These properties are discussed in detail later in this section.

Properties of the DRAWING are inherited by all bodies within the drawing if they have the INHERIT(BODY) attribute. Drawing properties behave exactly as body properties except they are common for all instances of the drawing.

4.6 TEXT MACRO PROCESSING WITHIN PROPERTIES

It is convenient to be able to use text macros within property values. One mechanism for doing this was described above in the section about the PARAMETER attribute. The problems with this method are:

1. It only works for body properties.

2. The syntax of the property value must be severely restricted so that the text macros can be found.

3. If there is any text in the property value that coincidentally matches a text macro name, it is expanded. This can result in very strange results.

To solve these problems, another mechanism has been implemented that is much more flexible. Each of the problems above has been addressed as follows:

1. It works for all properties regardless of where they are attached.

2. The content of the property value can be whatever is desired since the text macros are clearly identified and are separate from the rest of the text.

3. The user has explicit control over what is expanded and what is not.

To use this feature, the property cannot be given the PARAMETER attribute. That is, the PARAMETER attribute mechanism and this mechanism are mutually exclusive.

Text macros need to be identified within the property with the '%' character. This character serves to mark the presence of the text macro and to prevent confusion between text macros and normal text. For example:

PARMS = "W=%WIDTH,L=%LENGTH"

Note the presence of the two text macros (WIDTH and LENGTH) in the property value. They are flagged with the '%' character. By coincidence, the character 'L' is also a text macro, defined to be SCOPE="LOCAL". Without the use of the '%', the property value would be turned into garbage.

The text macro name must be an identifier: a string of letters, digits, and '_' starting with a letter and no more than 16 characters long. If the text macro is to be embedded in text so that the text macro name cannot be easily identified, the name must be quoted. For example,

PARMS = "This property value is %'TM'ed."

The text macro 'TM' is identified by the quoted name.

These text macros are processed on output only. That is, they are ignored until the Compiler outputs them to the output file.

4.7 ADVANCED PROPERTY TOPICS

This section describes some advanced topics that require some SCALD language knowledge.

\NAC PROPERTY

A signal of either assertion can be connected to a pin with the \NAC property. The Compiler assigns the assertion of the first signal connected to the pin as the pin's assertion. This forces any other signals connected to that pin to have the same assertion as the first signal. For example, assume the signals A, B, and C* are connected to the pin CLOCK \NAC. If the A signal is the first signal connected to the pin, the CLOCK pin is forced to be asserted high (since A is asserted high). This is the same as saying that pin CLOCK does not have a bubble. Since the signals B and C* are also connected to pin CLOCK, they are synonymed with the signal A. However, the signal C* is low asserted which will be flagged as an error. The \NAC property can only be used in pin names.

If, in the above example, the signal C* was the first signal connected to the pin, CLOCK would be forced to be low asserted (since C* is low asserted). This is the same as saying that the CLOCK pin has a bubble. The signals A and B will be flagged as errors since they have the wrong assertion.

The first signal connected to a pin is chosen at random by the compiler. There is no way to predict which signal will be chosen. This is not a problem since assertion errors will be caught regardless of which signal is chosen first.

The \NAC is used when the assertion level of signals is not important but all the signals must be compatible.

For an example of the use of the \NAC property, see the MERGER bodies in the standard Valid library. The \NAC property can be used on any body desired.

\NWC PROPERTY

The \NWC property is used when the width (in bits) of a pin is not known or when it is desired that signals of any width be connectable to the pin. If a pin has the \NWC property, the compiler determines the actual width from the context in which it is used. Once the width has been determined, the pin is assigned that width. This means that the pin inherits the width of the first signal connected to it. All other signals connected to that pin must have the same width as the first. In this manner, the \NWC behaves like the \NAC property (see above). The \NWC property can only be used in pin names.

The first signal connected to a pin is determined at random by the compiler. There is no way to predict which signal will be chosen. This is not a problem, since all width incompatibilities are detected regardless of which signal is chosen first.

For an example of the use of the \NWC property, see the MERGER bodies in the standard Valid library.

The presence of the \NWC property on any pin of a body defines that body as a "plumbing" body. This means that the body's definition contains only signal synonyming to "plumb" its signals around. A MERGER is an example of a plumbing body. The \NWC property cannot be used on hierarchical bodies that expand to primitives. The Compiler generates an error message when this is done.

4.8 PROPERTIES RECOGNIZED BY THE COMPILER

The SCALD Compiler recognizes several predefined properties that convey information about the design and control the compilation. These properties are described below, one to a page. The box at the top of each page shows the property name, the object(s) to which the property can be attached, and the inheritance of the property. The objects to which properties can be attached are:

BODY - attached to a body
PIN - attached to a pin
SIGNAL - attached to a signal (wire)
DRAWING- attached to a DRAWING body

The inherit field shows the objects that can inherit the property. These are body, pin, signal. When a property cannot be inherited, the field shows:

INHERIT ()

This is the syntax used in the property attributes file for no inheritance.

Here, as an example, is the box for the NEEDS_NO_SIZE property:

NEEDS_NO_SIZE	: BODY	: INHERIT()

This means that the NEEDS_NO_SIZE property is interpreted only if it is attached to a body. If it appears anywhere else, it is an error detected by the Compiler. The property is not inheritable.

ABBREV	: DRAWING	: INHERIT( )

The ABBREV property specifies an abbreviation for a drawing. It should be attached to the DRAWING body. (Each drawing must have a DRAWING body added to it; see STANDARD Library in the *Library Reference Manual* for more information on a DRAWING body.) The abbreviation is used by the Compiler to create path names (see the *Compiler Reference Manual* for a description of path names).

If the ABBREV property is not found, the Compiler makes an abbreviation, derived from the name of the drawing. The ABBREV property value can only include letters, digits, and the underscore '_'. The Compiler generates an error message if any other characters are used.

NOTE

If different abbreviations are placed on two different pages of the same drawing, the Compiler takes the abbreviation from the highest-numbered page.

ALLOW_CONNECT	: PIN : BODY : SIGNAL	: INHERIT( )

The ALLOW_CONNECT property allows different types of outputs to be connected by specifying which outputs are to be "ignored" when an OUTPUT_TYPE is checked by the Packager. The ALLOW_CONNECT property may appear on a library part (as in the case of a connector) or in a logical design. If the ALLOW_CONNECT property is attached to a net, it applies to all output pins on the net. When attached to a body, it applies to all output pins on the body. When attached to a pin, the ALLOW_CONNECT property applies only to the pin to which it is attached.

BODY_TYPE	: BODY	: INHERIT()

The BODY_TYPE property specifies certain special bodies. It can be given the following values:

COMMENT
: The body is a comment and to be totally ignored. This property replaces the previous COMMENT_BODY property.

FLAG_BODY
: The body is used to indicate an I/O signal. Used by the Packager and DIAL to process module interface signals. It should be noted that parts identified as FLAG_BODYs are only output by the Compiler if they appear in the root level drawing.

REL_REF
: The origin of the body is used as the reference point for the XY properties attached to all other bodies within the drawing. This property usually appears on the B SIZE PAGE or similar drawing. The system also interprets it to be a comment (same as BODY_TYPE=COMMENT above).

ABS_REF
: Causes the Graphics Editor to use absolute coordinates for the XY property if found on any body within the drawing.

PLUMBING

The body is a plumbing body. Standard plumbing bodies are MERGERs, NOTs, SYNONYMs, etc. They are used to "plumb" signals in the drawing. Normally, the presence of the NWC (NOWIDTH) property on any pin of the body is used to determine whether a body is a plumbing body. If the body is a plumbing body but does not have any NWC pin (as is the case with the SLASH body), this property is attached.

BUBBLED	: PIN	: INHERIT( )

The BUBBLED property is used by the Graphics Editor to indicate when a pin is bubbled (has a bubble). The property only makes sense in this context and is an error everywhere else. Indeed, this property should NEVER be entered, assigned, or attached by the user; it is mentioned here only so that the user may know how the bubble information is passed to the Compiler.

The presence of the BUBBLED property on a pin means that only low-asserted signals may be connected to it. The assertion of the pin name has no bearing on assertion checking. The reason for this is the BUBBLE command capability of the editor which changes the graphical representation of the body (adds or deletes a bubble) without changing the pin name. The Compiler needs to know whether a pin has a graphical bubble on it since the pin name does not (and cannot) convey this information.

EXPR	: DRAWING	: INHERIT( )

The EXPR property specifies the selection expression for a drawing. Drawings can be conditionally compiled. The specific condition for which a particular drawing is intended is determined by the selection expression. For example, given the drawing PART1, there may be three versions with the following selection expressions:

PART1.LOGIC.1.1	(SIZE<4)	
PART1.LOGIC.2.1	(SIZE>=4)	and (SIZE<8)
PART1.LOGIC.3.1	(SIZE>=8)	

The PART1 drawing has three versions. Each version, presumably, is different. The Compiler selects one of the versions based on the value of its selection expression. If the selection expression has a non-zero value, that version of the drawing is used. The Compiler checks to make sure that one and only one of the selection expressions evaluates true. If no selection expression evaluates true, the Compiler outputs an error message (for additional information on selection expressions, see the *Compiler Reference Manual*).

The value of the EXPR property can be any integer expression involving constants, text macros, arithmetic operators (*, /, -, +, MOD), logical operators (OR, AND, NOT), relational operators (<, <=, =, >=, >, <>), or functions (ORD, ABS). See the Sections 6 and 7 for additional information.

NOTE

When selecting a multipage drawing with the EXPR property, the property must be attached to the drawing body on the first page.

HAS_FIXED_SIZE	: BODY	: INHERIT()

The HAS_FIXED_SIZE property identifies those bodies which have a fixed size. These bodies should not be given a SIZE property; indeed, it is an error to do so. The HAS_FIXED_SIZE property has two functions. First, it informs the Compiler that the body it attaches to has a fixed known size (specified in the property value); the Compiler will not produce a warning (#196). Second, it causes an error message to be produced if a SIZE property is found.

Many of the bodies in the standard Valid libraries have this property attached to them. It is used for versions of physical parts that display all sections. The vectored version of the body typically represents a one bit section of the part. The second body version represents all sections of the part. If the part has, for example, four sections, then the second body version will be given a HAS_FIXED_SIZE="4" property to specify that it represents four bits. This is important since the models (for the Timing Verifier and Logic Simulator) are modeled as one-bit sections with the SIZE property specifying the actual number of bits for each instance. The HAS_FIXED_SIZE property causes a "default" SIZE property to be attached to support the models. The presence of a user assigned SIZE property on these bodies is always an error since none of the pins of the body or the definition of the body use the SIZE property.

The HAS_FIXED_SIZE property can be attached as a default body property (attached to the ORIGIN body in the .BODY drawing) or it can also be attached to the body when used in a drawing (with the Graphics Editor's PROPERTY command).

NEEDS_NO_SIZE	: BODY	: INHERIT()

The NEEDS_NO_SIZE property is used to identify those bodies which need no SIZE property; indeed, it is an error to attach a SIZE property to one of these bodies. The NEEDS_NO_SIZE property has two functions. First, it informs the Compiler that the body it attaches to does not need a SIZE property; the Compiler will not produce a warning (#196). Second, it causes an error message to be produced if a SIZE property is found.

Many of the bodies in the standard Valid libraries have this property attached to them. Most notable are the NOT and MERGER bodies. These bodies automatically conform to the widths of the signals they are attached to. The presence of the SIZE property on these bodies is always an error since none of the pins of the body or the definition of the body use the SIZE property.

The NEEDS_NO_SIZE property can be attached as a default body property (attached to the ORIGIN body in the .BODY drawing) or it can also be attached to the body when used in a drawing (with the Graphics Editor's PROPERTY command).

NOASSERT	: PIN : SIGNAL	: INHERIT( )

The NOASSERT property can only be used as a PIN property or as a property of the pin name (signal name for a pin) It is not permitted on other signals. The presence of the property causes the Compiler to interpret the pin's assertion specially. The NOASSERT property is not permitted as a property of a pin or pin name that also has an explicit assertion specification (such as '*').

A pin with the NOASSERT property is interpreted as having no particular assertion. The pin assumes the assertion of the first signal connected to it. For example, if the signal CLOCK is connected to a pin with the NOASSERT property, the pin would become high asserted. If the signal ENABLE* were attached, the pin would become low asserted. If more than one signal is attached to the pin, all the signals would have to have the same assertion (the assertion of the first signal encountered since it is this assertion that the pin inherits).

A pin with the NOASSERT property may assume a different assertion for each instance of the body on which it appears. The best example of a body with pins with the NOASSERT property is the MERGE body. The MERGE body can be used on signals of any assertion; it assumes the assertion of whatever signal it is connected to.

To make the NOASSERT property easier to use, the standard text macro "NAC" (No Assertion Check) is provided with the following definition:

NAC = NOASSERT=""

The property value for the NOASSERT property is unimportant; just the presence of the property is significant. The signal

PIN NAME \NAC

is equivalent to the signal:

PIN NAME \NOASSERT=""

NO_IO_CHECK	: PIN : BODY : SIGNAL	: INHERIT( )

The NO_IO_CHECK property is used to suppress the Packager's input and output checks on a pin-by-pin, body-by-body, or net-by-net basis. The NO_IO_CHECK property can be given one of the following three values:

LOW	Suppresses the "0 state" I/O check; the "1 state" check is performed.
HIGH	Suppresses the "1 state" I/O check; the "0 state" check is performed.
BOTH or TRUE	Suppresses both the "0 state" and the "1 state" I/O check.

The NO_IO_CHECK property may appear on a library part (as in the case of a standard connector) or can be attached to various pins, bodies, and nets on a drawing. If the NO_IO_CHECK property is attached to a net, it applies to all pins on the net. When attached to a body, it applies to all pins on the body. When attached to a pin, the NO_IO_CHECK property applies only to the pin to which it is attached.

NOWIDTH	: PIN : SIGNAL	: INHERIT()

The NOWIDTH property can only be used as a PIN property or as a property of the pin name (signal name for a pin) It is not permitted on other signals. The presence of the property causes the Compiler to interpret the pin's width specially. The NOWIDTH property is not permitted as a property of a pin or pin name that also has an explicit bit width specification (bit subscript).

A pin with the NOWIDTH property is interpreted as having no particular width. The pin assumes the width of the first signal connected to it. For example, if the signal CLOCK is connected to a pin with the NOWIDTH property, the pin would be assigned the width 1. If the signal DATA<3..0> were attached, the pin would assume the width 4. If more than one signal is attached to the pin, all the signals would have to be the same width (the width of the first signal encountered since it is this width that the pin inherits).

A pin with the NOWIDTH property may assume a different width for each instance of the body on which it appears. The best example of a body with pins with the NOWIDTH property is the NOT body. The NOT body can be used on signals of any width; it assumes the width of whatever signal it is connected to.

The presence of the NOWIDTH property can cause signals that have indeterminate widths. If an UNNAMED signal connects to a pin with the NOWIDTH property, the Compiler must look further to determine the width of both the signal and the pin (since the width of an unnamed signal is determined from how it is used.) In some cases, there may be insufficient information available to determine the width. Such cases are detected in Pass 2 of the Compiler and reported.

To make the NOWIDTH property easier to use, the standard text macro "NWC" (No Width Check) is provided with the following definition:

NWC = NOWIDTH=""

The property value for the NOWIDTH property is unimportant; just the presence of the property is significant. The signal:

PIN NAME \NWC

is equivalent to the signal:

PIN NAME \NOWIDTH=""

PART_NAME	: DRAWING	: INHERIT( )

The PART_NAME property is used to specify the name of a primitive component. When the Compiler is ready to output a primitive component into the expansion output file, it has to determine the name of the component. Normally, this name is just the name of the primitive component. There are times, however, when it is desired to have the primitive component name be different from the logical component name.

For example, the LSTTL library components are called LS00, LS01, LS02, etc. Each part, however, is known in the Compiler expansion file as 74LS00, 74LS01, etc. since this is a more explicit name. The '74' that is left off the logical component name makes the name easier to type. Of course, giving the LS00 the PART_NAME 74LS74 is counter-productive. The PART_NAME properties are found attached to the DRAWING body within the .PART drawing for the component.

If the Compiler finds a PART_NAME property, it uses it as the name of the primitive component otherwise, it uses the logical component's name.

PATH	: BODY	: INHERIT( )

The PATH property is used by the Compiler in the formation of path name elements. It is attached to each body in a drawing (either manually with the PROPERTY command or automatically in the Graphics Editor). The PATH property value can be any string of letters or digits. The Graphics Editor creates a PATH property of the form *n*P where *n* is unique for each body on the drawing.

The PATH property can be added as a default body property (by attaching the property to the origin body in the .BODY drawing), but is normally not a good idea. If more than one of the bodies (with a default PATH property) is used within the same drawing, the Compiler will output an error message since the path element created for the two bodies is the same. If default PATH properties are used, the property value will have to be changed if there is more than one of these bodies in the drawing.

REP	: SIGNAL	: INHERIT( )

The REP property is used to replicate a signal in much the same way that SIZE is used to replicate a body. The REP property causes the Compiler to create multiple copies of the signal and append them to the original to form one signal. For example, the signal:

BUS SIGNAL<4..1> \REP="2"

is equivalent to:

BUS SIGNAL<4..1> : BUS SIGNAL<4..1>

To make the REP property easier to use, the standard text macro 'R' is available which has the definition:

R = REP="%1"

where %1 is the parameter of the macro. The signal given above would appear as follows when this text macro is used:

BUS SIGNAL<4..1> \R 2

SCOPE	: SIGNAL	: INHERIT( )

The SCOPE property is used to define the scope of a signal. There are three possible values for this property, defined as follows:

LOCAL
: Signals on different pages of the same drawing are equated.

GLOBAL
: Signals at all levels of a hierarchical design are equated.

INTERFACE
: Used in hierarchical design and library development to indicate an interface signal from a higher level drawing.

By default, the scope of all signals is LOCAL. The SCOPE property is usually included in the signal name by using one of three standard text macros:

\I = SCOPE="INTERFACE"
\L = SCOPE="LOCAL"
\G = SCOPE="GLOBAL"

as, for example, in the signal DATA <15..0> \L. For the correct syntax, see under Signal Name Syntax/General Properties, earlier in this manual.

A signal cannot be given more than one scope, and cannot inherit its scope from some other signal.

SIZE	: BODY	: INHERIT( )

The SIZE property is one of the most powerful and basic of SCALD properties. Only bodies can be given the SIZE property. For more information, see the *Graphics Editor Reference Manual* and the following sections in this manual:

Signal Name Syntax/Parameter Attributes
Text Macro Facility/Use in Body Parameters

TERMINAL	: BODY	: INHERIT( )

The TERMINAL property is used to tell the Compiler that an empty drawing is not an error. Normally, the Compiler will generate an error message if a drawing is found that is empty; that is, it contains no bodies. It is expected that the drawing was not written, accidentally deleted, or otherwise ruined.

There are some drawings that are intentionally empty. The SYNONYM.LOGIC drawing is one of these. The entire purpose of the SYNONYM body is to synonym signals together. This it does by virtue of the fact that all of its pins have the same name. There is nothing else to be done and, therefore, the SYNONYM.LOGIC drawing is empty. The Compiler will generate an error. To inform the Compiler that the drawing is intentionally empty, the TERMINAL=TRUE property is attached to the DRAWING body within the SYNONYM drawing.

TIMES	: BODY : DRAWING	: INHERIT(BODY)

The TIMES property is one of the most powerful and basic of SCALD properties. Its use is complex and proper justice cannot be given it here. See the *Packager* and *Graphics Editor Reference Manuals* for a complete discussion.

Here it is appropriate to mention that the TIMES property is inherited. If a high level body is given the TIMES property (either by attaching it to the BODY or to the DRAWING body with the corresponding LOGIC drawing), it is inherited by all bodies within the corresponding LOGIC drawing. This makes it possible to assign a TIMES property to a large group of components without having to assign it individually.

TITLE	: DRAWING	: INHERIT( )

The TITLE property is used to specify the name of the drawing. The property can only be attached to the DRAWING body found within the drawing. If the drawing's name and the TITLE property's value are different, the Compiler will output an error message. The names must be identical (character for character) if the Compiler is not to produce this error.

The TITLE property is used to document, within the drawing, the name of the drawing. It it not required; no error, oversight, or warning message is output if it does not appear.

WIRE_DELAY	: PIN : SIGNAL	: INHERIT(PIN)

The WIRE_DELAY property is used to specify wire delays on signals. The text macro \WD has been defined to take a single parameter, as follows:

WD = 'WIRE_DELAY="%1"

Wire delay can, therefore, be succinctly added as a property in a signal name. Here are some examples:

\WD 2.0-5.6
\WD 1.0-2.0

Wire delay can also be attached as a signal property directly to a signal, or as a pin property directly to an input pin. To do so, use the property command and enter:

WIRE_DELAY 2.0-5.6

Remember the underscore in WIRE_DELAY (to make the entire string the property name), and the space after WIRE_DELAY (to separate property name from property value).

When wire delay is added to a signal, it is inherited by the pin attached to that signal.

The value of the \WD property can be a single range, two ranges, or a fixed-point number. When the value includes two ranges, the first range is the minimum and maximum rising delay, and the second range is the minimum minimum and maximum falling delay, as in:

WIRE_DELAY 2.0-5.6,2.5-6.2

When only one range is given, the rising and falling delays are assumed to be the same. See Delay Properties in the *Timing Verifier Reference Manual* for additional information.

SECTION 5
TEXT MACRO FACILITY

This section describes the text macro facility of the SCALD Compiler. Text macros are used to globally replace one string of characters with another. The first section explains what text macros are and how they are used. The following sections describe the specific syntax used to define text macros and their use in signal names and properties. Examples are given to demonstrate various features. Knowledge of the SCALD signal name syntax is helpful as the examples refer frequently to signal names.

5.1 WHAT IS A TEXT MACRO?

A text macro is a string of characters (usually short) that represents another string of characters. The Compiler replaces each occurrence of each text macro with the string it represents. For example, the text macro "VLS" can represent the string "Valid Logic Systems," and the text macro "CURRENT_ADDRESS" can represent "2820 Orchard Parkway, San Jose, CA 95134." Then, the sentence:

> Company headquarters of VLS are at CURRENT_ADDRESS.

appears as follows after the text macros have been replaced:

> Company headquarters of Valid Logic Systems are at
> 2820 Orchard Parkway, San Jose, CA 95134.

The process of replacing the text macros with the strings of characters they represent is called text macro expansion.

Text macros serve two basic functions, both of which are demonstrated by the above example. VLS is a useful abbreviation for the longer and more cumbersome Valid Logic Systems. CURRENT_ADDRESS represents a parameter that could change. The text macro lets you concentrate the variable information in one place. When the

parameter changes value, you need only change the definition of the macro. Text macros are useful for defining global information that is needed in many places and is likely to change.

In the SCALD language, text macros are most commonly used in signal names. For example, if the text macro ADDRESS_BUS is defined as:

ADDRESS_BUS = "23..0"

then signals that reference the address bus can be named as follows:

INTERFACE <ADDRESS_BUS>

This expands to:

INTERFACE <23..0>

If the size of the address bus is changed to, for instance, 31..0, the text macro definition is all that need be changed.

A more typical use might be the assignment of bit fields that represent register fields within an instruction or portions of some interface bus. The bit assignments need be determined only once and can be used throughout the design with little chance of error. Again, should the bit assignments be changed, only the text macro definitions have to be altered. For example:

```
interface_bus  = "0..63"
address_bus    = "0..23"
data_bus       = "24..39"
interrupts     = "40..42"
flags          = "43..50"
control        = "51..61"
reset          = "62"
power_fail     = "63"
```

5.2 WHERE TO DEFINE TEXT MACROS

There are two places within the SCALD system to define text macros: on individual drawings, and in a text file used by the compiler. A text macro that is defined on a particular drawing is operative (in full compilation) within that drawing and all other drawings under it in the hierarchy. A text macro that is defined in a text file is globally operative each time the compiler is used. (For separate compilation, text macros should either be globally defined, or explicitly defined on each drawing.) When you define a global text macro (in a text file) that macro cannot be overridden. Certain global text macros have been predefined for use in signal names. See "Globally Defined Text Macros" below.

A text macro defined on a particular drawing can be overridden by a macro on a drawing lower in the hierarchy. For example, take the drawing ALU.LOGIC that contains the drawings PART1.LOGIC, PART2.LOGIC, AND PART3.LOGIC, and also contains the text macro definition CTR = counter. If the drawing PART3.LOGIC contains the text macro definition CTR = counter1, the compiler will expand all occurrences of CTR in ALU.LOGIC, PART1.LOGIC, and PART2.LOGIC into "counter," but it will expand the occurrences of CTR in PART3.LOGIC into "counter1." This expansion lets you use text macros in higher level drawings to designate general cases, and override these macros for a specific case on a lower level drawing.

When you define a text macro on a page of a drawing (for example, ALU.LOGIC.1), the text macro is operative on all pages of that drawing (ALU.LOGIC.1, ALU.LOGIC.2,...). A given text macro cannot be defined more than once in the same drawing. The Compiler generates an error message when this happens.

5.3 DEFINING A TEXT MACRO ON A DRAWING

A text macro is an identifier, that is,

> a string of not more than 16 characters that includes letters, digits, and '_'(underscores) and starts with a letter.

The text macro definition can be any character string (with a maximum length of 255 characters). Text macros are defined in a DEFINE body placed in a drawing. To define a text macro for a drawing, add a DEFINE body and use the PROPERTY command to attach properties to the DEFINE body. The PROPERTY command expects a name/value pair separated by a space. If, at the PROPERTY command, you enter:

```
xxxx yy zz
```

The Compiler will interpret xxxx to be the text macro and yy zz to be the macro definition. Each time it finds xxxx on the drawing, it will expand that macro to yy zz. Any number of properties may be attached to the DEFINE body and any number of DEFINE bodies may appear in a drawing. Text macros defined on one page of a drawing are operative on all pages of that drawing.

See also Globally Defined Text Macros below.

5.4 HOW TO USE TEXT MACROS

Text macros may be used in several places: in other text macros, in signal names, in properties, and in body parameters. Each of these is described below.

USE IN OTHER TEXT MACROS

Text macros can be nested one inside the other. The macro COPYRIGHT can be defined as:

Copyright CURRENT_YEAR, VLS Inc.

where CURRENT_YEAR and VLS are also macros, that represent, respectively, "1985" and "Valid Logic Systems." To make a current copyright page, you need only type COPYRIGHT. This expands to:

Copyright 1985, Valid Logic Systems Inc.

The text macros CURRENT_YEAR and VLS are nested inside of the text macro COPYRIGHT. The text macro "COPYRIGHT" has a nesting depth of 2. The Compiler permits a maximum text macro nesting depth of 10. The Compiler rescans strings for macros until no more are found.

A text macro that is defined in terms of itself, either directly or indirectly (through another text macro that references the first), is a recursive text macro. A recursive text macro causes an error condition in the compiler, but it is difficult for the Compiler to detect the cause of the error condition. Usually, a recursive text macro results in one of the following error messages:

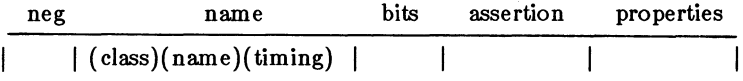
TEXT MACRO NESTING DEPTH EXCEEDED

EXPANDED TEXT MACRO EXCEEDS MAX LENGTH

USE IN SIGNAL NAMES

The use of text macros within signals requires some care. The Compiler recognizes a text macro by searching for an identifier and checking to see if the identifier is a text macro. Since SCALD signal names can contain almost any character sequence, it is conceivable that a signal name may inadvertently contain a sequence of characters that formed an identifier that just happened to be a text macro; it would be an error if the Compiler were to expand it.

For this reason, text macros are not permitted within the name portion of signal names. Remember, a signal name has these parts:



Since text macros would not be appropriate in the negation and assertion fields, macros within signal names are used only to define bit subscripts and properties.

USE IN PROPERTIES

The Compiler allows text macros to be placed within property values and have them expanded. Text macros are not allowed within property names. This capability is different than that supported for body parameters (see below).

Text macros need to be identified within the property value with the '%' character. This character serves to mark the presence of the text macro and to prevent confusion between text macros and normal text. For example:

PARMS = 'W=%WIDTH,L=%LENGTH'

Note the presence of the two text macros (WIDTH and LENGTH) in the property value. They are flagged with the '%' character. The Compiler only expands the identifier following the '%' character. The text macro name must, as always, be an identifier. The comma marks the end of the identifier. The character L is also a text macro, defined to be SCOPE=LOCAL within the SCALD III Language. But because it is not preceded by '%' it is not interpreted as a text macro. Because it is preceded by a comma, it is not interpreted as part of the text macro WIDTH.

If WIDTH = 2, and LENGTH = 3, then the above property expands to:

PARMS = 'W=2,L=3'

If the text macro is to be immediately followed by text (that is, by any character acceptable in an identifier), enclose it in quotes. For example,

PARMS = "This property value is %'TM'ed."

The text macro TM is identified by the quote marks. Text macros within property values cannot include parameters nor can they have embedded text macros. If such appear, they are ignored.

USE IN PARAMETERS

A parameter is a special body property that is evaluated by the Compiler and made available to the logic, time, or sim drawings associated with the body as a text macro.

The best example of such a property is SIZE which is attached to a body and used to specify the width of pin names, signal names, and to control size expansion.

The value of the parameter may be changed on an instance by instance basis thereby providing a means of passing information from the using drawing to the used drawing (hence the name body parameter). The body parameter may refer to text macros. The text macros must be defined on the drawing in which the body appears or in drawings above the containing drawing. A body parameter cannot refer to a text macro defined within the drawing to which the body corresponds.

All strings of characters that are identifiers (a string of letters, digits, and '_' starting with a letter and no more than 16 characters long) that happen to be text macro names are expanded. This means that body parameters must have very restricted formats. Typically, they are defined as simple integer expressions such as 'X+1' or 'SIZE-1'.

The most common text macro used within the SCALD language is SIZE which is defined by the user to specify the number of bits a component represents or the width of a bus. Pin names are often defined as "SIZE" bits wide as follows:

PINA <SIZE-1..0>

When the SIZE text macro is expanded, the width of the pin can be determined.

5.5 WHERE TEXT MACROS MAY NOT BE USED

There are several places where text macros are not permitted. They are described below.

DRAWING NAMES

Drawing names are assigned in the Graphics Editor and are stored in the TITLE property of the drawing and the drawing directory. Text macros are not permitted here because the Graphics Editor would have to know about them and how to expand them.

PROPERTY NAMES

The name of a property is exactly as entered. It cannot contain, reference, or otherwise depend upon text macros.

5.6 TEXT MACROS WITH PARAMETERS

The text macro capabilities described above are useful, but inadequate for some applications where simple text substitution does not provide enough flexibility. The SCALD Compiler text macro processor also supports text macros with parameters.

At times, it is advantageous to allow the use of a text macro to be customized on an instance by instance basis. For example, suppose a text macro is to be defined that specifies a bus by specifying the width desired. This could be supported as follows:

```
BUS8 = "0..7"
BUS16 = "0..15"
BUS24 = "0..23"
etc.
```

Since the relationship between the left and right sides of these text macros is the same (8 = 7+ 1, 16 = 15+ 1, 24 = 23+ 1), all three macros can be replaced by a single text macro that includes a parameter describing the size of the bus. For example:

```
BUS = "0..%1-1"
```

where '%1' is replaced by the parameter that the user gives to the text macro "BUS."

```
BUS 8
```

is expanded into

```
0..7
```

and

```
SIGNAL <BUS 8 >
```

is expanded into

```
SIGNAL <0..8-1>
```

A text macro parameter may have up to 16 characters, and may include any character except a space.

Note that the text macro parameter (in this case, 8) is preceded and followed by at least one space. If the trailing space is left out, as in:

SIGNAL <BUS 8>

the Compiler interprets 8> to be the parameter. This results in

SIGNAL <0..8>-1

and will produce an error message from the Compiler. The Compiler uses spaces to delimit parameters in text macros. If there are several parameters in a text macro, they must be separated by spaces.

When the Compiler detects an error in the syntax of the line it is reading, it prints out the line and all text macros that are being expanded so that is it clear how the expansion has been done. This makes it easier to find errors such as this one.

5.7 MULTIPLE PARAMETERS IN TEXT MACROS

Text macros can include up to nine parameters (1 through 9). The parameters are numbered from left to right (following the text macro) starting with one. For example, given the text macro "DBUS" with five parameters and a use as follows:

DBUS 23-4 9..4+1 w , BDAC

the parameters are:

parameter 1 = "23-4"
parameter 2 = "9..4+1"
parameter 3 = "w"
parameter 4 = ","
parameter 5 = "BDAC"

A text macro parameter may itself be a text macro. For example, given the following text macro definitions:

```
ORDER = "Beginning %1 the %2"
LAST = "ENDING"
MIDDLE = "PRECEDES"
```

The use of "ORDER" as follows:

```
ORDER MIDDLE LAST
```

expands as "Beginning PRECEDES the ENDING."

The text macro "ORDER" is given two parameters; "MIDDLE" and "LAST." The two parameters are themselves text macros which are expanded to "PRECEDES" and "ENDING" respectively. Text macros that require parameters (such as ORDER above) should not be used as parameters of other text macros. They will be expanded, but the parameter order and binding is very obscure. Parameterized text macros should NOT be used as parameters of text macros.

5.8 GLOBALLY DEFINED TEXT MACROS

There are many predefined text macros created by the Compiler. These text macros are globally known (that is, they are accessable by every drawing within a design) and are reserved; the designer is prevented from creating a text macro with the same name. These predeclared text macros are:

L	→ gives local scope to a signal
G	→ gives global scope to a signal
I	→ identifies a signal as an interface signal
R	→ used to specify signal replication
WD	→ wire delay property
B	→ indicates that a pin has a bubble
NWC	→ no width check property
NAC	→ no assertion check property
TRUE	→ constant 1
FALSE	→ constant 0
X	→ current value of the X variable

Their definitions are:

```
R       REP="%1"
G       SCOPE="GLOBAL"
L       SCOPE="LOCAL"
I       SCOPE="INTERFACE"
NWC     NOWIDTH=""
NAC     NOASSERT=""
B       BUBBLED=""
TRUE    1
FALSE   0
```

The designer may define additional globally known reserved text macros. These are given to the Compiler in a text macro file (see the TEXT_MACRO_FILE directive in the Compiler directives section for a description of the method). The form of the file is

```
FILE_TYPE = TEXT_MACROS;
text macro name = text macro definition ;
    .
    .
    .
    END.
```

where *text macro name* is the name of the text macro being defined and *text macro definition* is the text macro value enclosed with quotes. There are a few text macros that should be defined for the Timing Verifier. They appear below as an example of what the text macro file should look like.

```
FILE_TYPE=TEXT_MACROS;
S  = 'S_ASSERT="%1"';
P  = 'P_ASSERT="%1"';
C  = 'CLOCK="%1"';
WD = 'WIRE_DELAY="%1"';
CD = 'CHIP_DELAY="%1"';
E  = 'EVAL="%1"';
END.
```

These text macros are used to support the timing assertion properties used by the Timing Verifier. Note that all of them require one parameter. For example, the "WD" text macro could be used in a signal as follows:

SIGNAL NAME <0..31> \WD 3.2-4.5

The parameter for the "WD" text macro is "3.2-4.5." Note also that these text macros are used as shorthand for property specifications in signal names. A property in a signal name is specified by the property name followed by '=' followed by the property value in quotes. Since the text macro definition contains quotes, two kinds of quotes are used to reduce confusion. An alternate, and equivalent, method of defining the "WD" text macro appears below:

WD = 'WIRE_DELAY="%1"';

Two quotes in a row are taken to mean a single quote when found within a character string.

SECTION 6
SELECTION EXPRESSIONS

Parameters attached to a body can be used to "customize" the body's drawing. The most common parameter is the SIZE property which is used in structured and hierarchical designs to specify the number of bits represented by the body. Another commonly used parameter is the DELAY property. The use of parameters allows the designer to have each instance of a body represent a slightly different implementation without having many different bodies and drawings.

In some cases, there is a need for radical differences between implementations of a single circuit. These differences are not simply parametric, which can be handled easily with body parameters, but involve changes in the circuit topology. For example, a gate may be designed with several different versions, one version having input protection diodes, one with internal pullups, one with high capacitance load/drive capability, and others with combinations of these. Since each version represents the same gate, the user would prefer to define a single body to represent the gate and then use some parameter to control which of the gate representations is used. This is supported in the SCALD system by drawing versions and selection expressions.

6.1 DRAWING VERSIONS

Each different implementation of a single circuit is called a drawing version. There is no limit to the number of versions that can be defined for a single drawing. Each of the drawing versions corresponds to the same body.

In the gate example described above (a NAND gate), the following drawings might be created:

```
NAND.BODY.1.1
NAND.LOGIC.1.1 <- ''generic'' NAND representation
NAND.LOGIC.2.1 <- NAND with input diodes
NAND.LOGIC.3.1 <- NAND with internal pullups
NAND.LOGIC.4.1 <- NAND with high-C drive
        .
        .
        .
```

Four versions are shown above (though more can be defined) and each version shown has only one page although a drawing version can have any number of pages.

6.2 SELECTION EXPRESSIONS

If a drawing has more than one version, there must be a method of selecting which version is to be used for any particular instance. This is done with the use of parameters. When the NAND body is placed in a drawing, a parameter must be attached to the body to specify which of the NAND drawing versions is to be used to allow each instance of the body to refer to a different implementation.

Once the parameters have been attached to the bodies, there must be a method of selecting the appropriate drawing version. This form of selecting is done with a selection expression. The selection expression defines the "context" in which the drawing is valid. In the case of the NAND, the context is used to select which of the drawing versions (or NAND gate implementations) is to be used.

A selection expression can be an arbitrary integer or Boolean expression. In the NAND example, assume that the parameter TYPE is used to select among the drawing versions. Four selection expressions are needed; one for each drawing version:

```
NAND.LOGIC.1.1   (TYPE=0)
NAND.LOGIC.2.1   (TYPE=1)
NAND.LOGIC.3.1   (TYPE=2)
NAND.LOGIC.4.1   (TYPE=3)
      .
      .
      .
```

The selection expressions (TYPE=0, TYPE=1,...) define which value of the parameter TYPE is to be used for a particular drawing version. The user sets the value of the TYPE parameter to select the desired version of the NAND.LOGIC drawing.

Selection expressions are defined in the drawing as an EXPR property attached to the DRAWING body of the drawing. If a drawing has more than one version, each version must be given a selection expression to specify under which condition that version is to be used.

6.3 HOW SELECTION EXPRESSIONS ARE EVALUATED

Whenever a component is found that has more than one implementation (more than one drawing version), the Compiler must determine which version is to be used. This is done by evaluating the selection expressions for each version and picking the version with the selection expression that evaluates TRUE for that instance.

If the selection expression is a Boolean expression (e.g., SIZE>2), the selection expression is TRUE if the expression evaluates TRUE. If the selection expression is an integer expression (e.g., SIZE+ 1), the selection expression is TRUE if its value is not 0. If the selection expression is empty (or absent), it evaluates TRUE.

Only one version's selection expression may evaluate TRUE for any given instance. An error is generated if the Compiler finds that the selection expressions for more than one version evaluate TRUE. For example, the following selection expressions are in error because the first two expressions evaluate TRUE for SIZE=2:

```
(SIZE>1)
(SIZE=2)
(SIZE<=1)
```

In the above examples, the selection expressions have been shown using the SIZE parameter although any parameter or text macro may be used in a selection expression.

If the Compiler discovers an error when evaluating selection expressions, it outputs the selection expressions for all of the versions to make it easier for the user to see what has happened and to provide a guide to solving the problem.

6.4 SELECTION EXPRESSIONS IN DRAWINGS

As mentioned above, the selection expression is defined by an EXPR property attached to the DRAWING body of the drawing. It is important to note that the EXPR property must appear in the first page of the drawing (the first page is the lowest numbered page and not necessarily page 1).

The EXPR property in the first page of the drawing defines the context for the entire drawing. Once the drawing has been selected, further selection can be performed. If EXPR properties are used in other pages of the drawing, the Compiler evaluates them to decide if that page is to be used. This gives the user the ability to define a selection expression for the entire drawing, and to specify a second selection expression for each page to determine whether it is used.

6.5 EXPRESSION EVALUATION

Selection expressions follow the standard SCALD system expression evaluation rules described in section 7. Only expressions that evaluate to integer quantities are supported for selection expressions.

SECTION 7
EXPRESSIONS

7.1 USE OF EXPRESSIONS

In general, anywhere a number is expected, an expression can be used. This capability allows the designer to use expressions that clearly represent the source or structure of a number. For example, if a half of some other quantity is needed, it is much better to enter Y/2 than 4 (if Y is 8). This has two advantages. First, if the value of Y changes, all other quantities that depend on Y also change. Second, it shows that the second quantity depends on Y.

The expression syntax in the SCALD language supports expressions that evaluate to integer quantities only. The following operators are supported:

OR	- inclusive OR value is 0 (false) or 1 (true)
XOR	- exclusive OR value is 0 (false) or 1 (true)
AND	- AND value is 0 (false) or 1 (true)
<	- signed less than value is 0 (false) or 1 (true)
>	- signed greater than value is 0 (false) or 1 (true)
<=	- signed less than or equal value is 0 (false) or 1 (true)
>=	- signed greater than or equal value is 0 (false) or 1 (true)
=	- equal value is 0 (false) or 1 (true)
<>	- not equal value is 0 (false) or 1 (true)
+	- signed addition value is integer
-	- signed subtraction value is integer

*	- signed multiplication value is integer
/	- signed division value is integer
MOD	- remainder value is integer
NOT	- logical complement value is either 0 or 1
ORD	- ordinal value value is either 0 or 1
ABS	- absolute value value is positive integer
MAX	- maximum of *n* values value is integer
MIN	- minimum of *n* values value is integer

Operator precedence refers to the order in which operations are performed when evaluating an expression. Operators with highest precedence are evaluated first. The operator precedence can be demonstrated by considering the following expression:

X>1 OR Y<2 AND X=4

This is obviously confusing. This expression is evaluated as though there were parentheses as follows:

((X>1) OR ((Y<2) AND (X=4)))

Operator precedence is as follows:

1. NOT {highest precedence}
2. MAX MIN ABS ORD
3. * / MOD
4. + -
5. < > = <= >= <>
6. AND
7. OR XOR {lowest precedence}

Parentheses can be used to force expression evaluation order if desired. For operators that have the same precedence, evaluation is performed left to right.

7.2 BNF FOR EXPRESSIONS

The following is the BNF for expressions within the SCALD language.

expression list ::= *expression* |
expression list, *expression*

expression ::= *boolean expression* |
expression bool OP *boolean expression*

bool OP ::= OR | XOR

boolean expression ::= *relational expression* |
boolean expression AND *relational expression*

relational expression ::= *simple expression* |
simple expression rel OP *simple expression*

rel OP ::= < | > | <> | = | >= | <=

simple expression ::= *term* |
sign term |
simple expression add OP *term*

sign ::= + | -

add OP ::= + | -

term ::= *factor* |
term mul OP *factor*

mul OP ::= * | / | MOD

factor ::= *unsigned constant* |
identifier |
(*expression*) |
NOT *factor* |
ABS (*expression*) |
ORD (*expression*) |
MIN (*expression list*, *expression*) |
MAX (*expression list*, *expression*)

expression list ::=*expression* |
expression expression list

unsigned constant::=*unsigned number* |
string

INDEX