6. Expressions

Extensions to IEEE Std. 1450-1999 Clause 6

STIL.0 defines a limited usage of expressions - see Timing Expressions using Spec Variables (STIL.0, clause 6.13), and Signal Expressions (STIL.0, clause 6.14). This standard extends these expression capabilities with additional variable types and corresponding expression constructs. The set of operators and the expression constructs as defined for STIL.0 are unchanged. The full set of STIL.0 and STIL.1 expression operations are defined in Table 2 on page 4. The detail of syntax and usage of the new expressions are defined in the following sub-clauses.

6.1 Variables and constants

Signal variables, integer variables, and constants are declared in a Variables block - see “Variables Block” on page 24. For allowed usage of these variables and constants, refer to the following sub-clauses on each of the new expression types.

Expressions may contain references to signals, groups, and spec variables. Therefore, the name space for variables, and constants includes the names in effect in these additional namespaces. Variables and constants follow the same name scoping rules as SignalGroup names, that is: names shall be unique across items in the current Variables block names; items defined in an unnamed Variables block are accessible without reference; items in a named variables block override a global definition; and an item of the same name shall not occur in more than one named Variables domain.

Constants are used to support the assignment of names to values used in expressions. There are two types of constants: integer constants and WFC constants. Integer constants are allowed wherever an integer value is allowed in an expression. WFC constants are allowed wherever a wfc string is allowed. Boolean expressions with WFCConstant and IntegerConstant support equals (==) and not-equals (!=) operators. Greater-than (>) and less-than (<), and the also-equals forms of these operations (<=, >=) are supported only for IntegerConstant. Constants are allowed in bracketed Signal or SignalGroup contexts. STIL rules for user-defined naming as defined in STIL.0 subclause 6.10 apply to the naming of variables and constants.

6.2 Expression constructs

Expressions define a sequence of arithmetic operations to be performed.

The left hand side of an expression defines the usage of the expression and determines the type of the expression to be allowed on the right hand side. The right hand side of the expression shall be either a single token, or an expression enclosed in single quotes.

Expressions can be constructed in the following ways:

a) A keyword identified expression:
   
   Loop 50 {
   
   ScanLength 100;
   
   ScanEnable ‘A&B’;
   
   b) A stand alone expression:
      
      Condition { VAR = 5; }
      
      Vector { VAR = ‘X+1’; }
      
   c) A conditional expression:
      
      If ‘VAR == 6’ { V { CLK = P; } }

Expressions may evaluate to integer, real number, boolean value, wfc-list, or logic. The result of the last operation executed in evaluating an expression determines the value - for example the result of assigning a value to an integer variable is the value of that assignment. The context in which the expression is used determines how it is to be interpreted - integer, logic, boolean.

Expressions consist of:
a) References to Variables and Constants that are defined under Variables blocks that are currently in scope according to the PatternBurst context.

b) References to Signals, SignalGroups, or Spec Variables that are currently in scope according to the Pattern-Burst and PatternExec context.

c) Integers, or real numbers.

d) Expressions as described in STIL.0 clause 6.13 but not including event labels nor time marks, and supporting the set of operators defined in table 2.

Consider the following examples of expressions:

```plaintext
1: STIL 1.0 { Design D15; }
2: Header {
3:   Source "P1450.1 Working-Draft 16, Sep 17, 2003";
4:   Ann (* clause 6.2 *)
5: }
6: Signals { S[1..4] In; }
7: SignalGroups { SIGGRP = 'S[1..4]'; }
8: Variables {
9:   IntegerConstant RUN=0;
10:  IntegerConstant STOP=1;
11:  IntegerConstant LOAD=2;
12:  IntegerConstant UNLOAD=3;
13:  Integer CMDS { Values RUN STOP LOAD UNLOAD; }
14:  Integer A;
15:  Integer B {InitialValue 1000;}
16:  Integer C;
17:  Integer E;
18:  SignalVariable W[1..4];
19:  SignalVariable VAR1;
20:  WFCConstant STOP=00;
21:  WFCConstant GO=01;
22:  WFCConstant RESET=10;
23:  SignalVariable VAR2[1..2];
24: }
25: Spec {
26:   Category CAT {
27:     R = '25ns';
28:   }
29: }
30: Pattern PAT {
31:   If 'A == 0' {} // False (in a boolean context)
32:   If 'A == 1' {} // True (in a boolean context)
33:   If 'A == 956' {} // True (if present in a boolean context)
34:   If A {} // True if a > 0
35:   If 'R >= 20ns' {} // a real number in engr units
36:   If 'A == 5' {} // test for a equal to 5; returns 1 if True and 0 if False
37:   If 'A <= B' {} // test for a less than or equal to b
38:   If 'A < min (B ,C)' {} // use of the min function
39:   C { A = 5; } // set variable a equal to 5
40:   C { E = \W RUN; } // set using constant definition
41:   C { W = LH01; } // set signal variable to the string LH01
42:   C { B = \h FF; } // set integer to 255
43: // As a boolean expression in a Pattern or PatternBurst:
```

Copyright © 2001 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.
6.3 Expressions and operator precedence

The following table describes the usage of each type of variable and expression and where each variable type is defined. This table does not have a corresponding table in STIL.0.

<table>
<thead>
<tr>
<th>expr-type</th>
<th>variable types</th>
<th>where defined</th>
<th>where used</th>
<th>syntax examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>engr_expr</td>
<td>time</td>
<td>Spec</td>
<td>Timing</td>
<td>'23.0ns/2+16.5e-9-t2'</td>
</tr>
<tr>
<td></td>
<td>integer</td>
<td>Variables</td>
<td></td>
<td>'t1/i + r - t2'</td>
</tr>
<tr>
<td></td>
<td>real</td>
<td>Spec</td>
<td>Timing</td>
<td>Same as an engr_expr except that it may be used in a non-engr unit context. e.g. volts/ns.</td>
</tr>
<tr>
<td>real_expr</td>
<td>integer</td>
<td>Variables</td>
<td>Pattern</td>
<td>Loop 'x * 25' {}</td>
</tr>
<tr>
<td></td>
<td>integer constant</td>
<td>Variables</td>
<td></td>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Variables</td>
<td></td>
<td>3 6 4 5 1 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Spec</td>
<td>Timing</td>
<td>ScanStructures</td>
</tr>
<tr>
<td></td>
<td>integer constant</td>
<td>Signals</td>
<td></td>
<td>ScanGroups { a='b+c'; }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SignalGroups</td>
<td></td>
<td>V { 'x+y' = 11; }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Variables</td>
<td>Pattern</td>
<td>V { grp = \Wsv1; }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ScanStructures</td>
<td></td>
<td>V { grp = \Wsv2[1..5]; }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Variables</td>
<td>Pattern</td>
<td>V { grp = \Wsv3[5 4 2 3 1]; }</td>
</tr>
<tr>
<td>sigref_expr</td>
<td>signals</td>
<td>Signals</td>
<td>most blocks</td>
<td>SignalGroups { a='b+c'; }</td>
</tr>
<tr>
<td></td>
<td>signal groups</td>
<td>Variables</td>
<td></td>
<td>V { 'x+y' = 11; }</td>
</tr>
<tr>
<td></td>
<td>integer constant</td>
<td>on-the-fly</td>
<td></td>
<td>V { grp = \Wsv1; }</td>
</tr>
<tr>
<td>sigvar_expr</td>
<td>signal variable</td>
<td>Variables</td>
<td>Pattern</td>
<td>V { grp = \Wsv1; }</td>
</tr>
<tr>
<td></td>
<td>wfc constant</td>
<td>Variables</td>
<td></td>
<td>V { grp = \Wsv2[1..5]; }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ScanStructures</td>
<td>ScanEnable</td>
<td>'A &amp; B';</td>
</tr>
<tr>
<td>logic_expr</td>
<td>signal</td>
<td>Signals</td>
<td>ScanStructures</td>
<td>ScanEnable 'A &amp; B';</td>
</tr>
<tr>
<td></td>
<td>signal group</td>
<td>SignalGroups</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>integer constant</td>
<td>Variables</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 1—Variable and Expression Usage

<table>
<thead>
<tr>
<th>expr-type</th>
<th>variable types</th>
<th>where defined</th>
<th>where used</th>
<th>syntax examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean_expr</td>
<td>integer</td>
<td>Variables</td>
<td>Pattern</td>
<td>If 'i == 0' {}</td>
</tr>
<tr>
<td></td>
<td>integer constant</td>
<td>Variables</td>
<td>ScanStructures</td>
<td>If 'period &gt; 23.0ns' {}</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>Spec</td>
<td></td>
<td>If 'value &lt; 3e-6' {}</td>
</tr>
<tr>
<td></td>
<td>real</td>
<td>Spec</td>
<td></td>
<td>If 'sv2[1..5] == 11000' {}</td>
</tr>
<tr>
<td></td>
<td>SignalVariable</td>
<td>Variables</td>
<td></td>
<td>If '(s1==H)</td>
</tr>
<tr>
<td></td>
<td>Signal</td>
<td>Signals</td>
<td></td>
<td>If 'grp != HHHH' {}</td>
</tr>
<tr>
<td></td>
<td>SignalGroup</td>
<td>SignalGroups</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following table is expanded from Table 5 of STIL.0, to identify operator usage in additional expression types, and to define the following operators not supported in STIL.0: % (modulus), ! (negation), && (boolean and), || (boolean or), ~ (tilde, bit-wise negation), & (bit-wise and), | (bit-wise inclusive or), ^ (bit-wise exclusive or), and ^~, ~^ (bit-wise equivalence). Logical behavior of these operators is the same as defined in IEEE Std. 1364. Table 2 defines the allowed operators on each type of variable. The table is ordered by precedence. Operators within the bold line are of equal precedence and are processed left to right. Operators in each bold line separated group have higher precedence that the groups below them and shall be processed first.

Table 2—Operators and functions allowed in expressions

<table>
<thead>
<tr>
<th>Op</th>
<th>Definition</th>
<th>time</th>
<th>real</th>
<th>integer</th>
<th>logic</th>
<th>boolean</th>
<th>sigvar</th>
<th>sigref</th>
<th>cellref</th>
<th>user func</th>
<th>stmt token</th>
</tr>
</thead>
<tbody>
<tr>
<td>min ()</td>
<td>minimum value</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>max ()</td>
<td>maximum value</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Merged-Scan ()</td>
<td>boolean function that returns true to select BreakPoint or other option during scan operation. See clause 6.10.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>()</td>
<td>parenthesis</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>,</td>
<td>comma - used as a separator in user functions</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>dot used as a domain reference operator: xxx.yyy means item yyy within object xxx (used only in Timing context of STIL.0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>::</td>
<td>double colon used as a domain reference operator: xxx::yyy mean item yyy within object xxx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>negation (boolean value)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td>bit-wise negation</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2—Operators and functions allowed in expressions

<table>
<thead>
<tr>
<th>Op</th>
<th>Definition</th>
<th>time</th>
<th>real</th>
<th>integer</th>
<th>logic</th>
<th>boolean</th>
<th>sigvar</th>
<th>sigref</th>
<th>cellref</th>
<th>user func</th>
<th>stmt token</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>timing event reference</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>dot operator used as string concatenation (for long strings): xxx.yyy yields string “xxxxxy”</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>multiply</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%</td>
<td>modulus</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>add</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>subtract</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>less than (boolean value)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than (boolean value)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;=</td>
<td>less or equal (boolean value)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater or equal (boolean value)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>and (boolean value)</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>or (boolean value)</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>==</td>
<td>equal (boolean value)</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!=</td>
<td>not equal (boolean value)</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>bit-wise and</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit-wise inclusive or</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>bit-wise exclusive or</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>^=</td>
<td>bit-wise equivalence</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>conditional expression</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>assignment</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>..</td>
<td>range operator (ellipsis)</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 6.4 Boolean expressions (boolean_expr)

A boolean expression is an expression that evaluates to a true or false. The result of a boolean operation is FALSE if the result is 0, and TRUE if the result is 1. Contexts that require boolean expression values interpret the value 0 and negative values as FALSE and non-zero positive values as TRUE.

The following are examples of boolean expressions:

59: STIL 1.0 { Design D15; }
6.5 Integer expressions (integer_expr)

An integer expression is an expression that evaluates to integer. See the list of allowed operators in Table 2 on page 4. The following rules of interpretation apply to integers:

a) A bare integer may be declared either with or without single quotes.

b) The underscore character may be used as a separator within an integer declaration.

c) If an integer operation results in a number with a fractional part, the fraction is truncated to produce an integer value.

d) The following formats are not allowed for integers: 56E3, 56K.

Integer expressions may contain integers, operators as previously indicated, integer variables, and integer constants. The following are examples of integer expressions and their usage:

81: STIL 1.0 { Design D15; }
82: Header { 
83: Source "P1450.1 Working-Draft 16, Sep 17, 2003"; 
84: Ann {* clause 6.5 *}
85: }
86: 
87: Signals { SIGA In; SIGB Out; SIGC InOut; } 
88: Variables { 
89: Integer I; 
90: Integer J { InitialValue '56*1024'; } 
91: IntegerConstant K = 2; 
92: } 
93: SignalGroups { 
94: SIGREF = 'SIGA+SIGB+SIGC'; 
95: } // end SignalGroups 
96: 
97: Timing { WaveformTable { 
98: Waveforms { 
99: SIGREF { 


6.6 Logic expressions (logic_expr)

A logic expression is an expression comprised of signal and group names and bit-wise logic operators. A positive (i.e., TRUE) result of the expression means that the function is enabled.

The following is an example of a logic expression:

```
117: STIL 1.0 { Design D15; }
118: Header { 
119:   Source "P1450.1 Working-Draft 16, Sep 17, 2003";
120:   Ann {* clause 6.6 *}
121: }
122:
123: Signals { A In; B In; SI1 In; SO1 Out; }
124: SignalGroups { G = G[1..4]; }
125:
126: ScanStructures S1 {
127:   ScanChain CHAIN1 {
128:     ScanLength 100;
129:     ScanCells CC[1..100];
130:     ScanIn SI1;
131:     ScanOut SO1;
132:     ScanEnable 'A & B';
133:   }
134: }
```

6.7 Real expressions (real_expr)

A real expression is an expression that evaluates to a real number as determined by the context (i.e., key word) or the variable type on the left hand side of an expression. Real variables and expressions are defined in a Spec block in STIL.0. The term real_expr is used in the syntax definition to indicate where a real number is to be used. A real number can be expressed in one of two formats:

a) A real number can be of the form: <number>e<+-><number>. A real number can be used to represent values that are not standard SI units, for example a slew rate in volts/ns.

b) A real number can also be of the form <number><prefix><SI unit>. For example, '23ns' is a time expression, '10uf' is a capacitance expression. This generic reference can be used whenever one of the standard
Although there are commonly accepted rules with regard for the algebraic combination of values with engineering units, these rules are not enforced in any way by this standard. Therefore, although '1.5V+2ns' may not make logical sense, it is nevertheless allowed. A practical application of why this might be desirable is if a measured voltage, representing a timing calibration value, is stored in the spec variable 'V'. A resultant timing expression might be something like '2ns + V/1000'.

The following are examples of real number expressions:

```plaintext
135: STIL 1.0 { Design D15; }
136: Header {
137:   Source "P1450.1 Working-Draft 16, Sep 17, 2003";
138:   Ann (* clause 6.7 *)
139: }
140:
141: Signals { SIG_NAME InOut; }
142:
143: Spec {
144:   Category CAT {
145:     TIME = '25ns';
146:     REAL = '25e-9';
147:     VOLTS = '25nV';
148:     WATTS = '25mw';
149:     SLEWRATE = '1v/1ns';
150:   } // end Category
151: } // end Spec
152:
153: Pattern PAT {
154:   If 'TIME >= 23.5ns + 1.5ns/2' {} // time expression
155:   If 'WATTS >= 5v * 2ua' {}  // where wattage is of type 'Watts'
156:   If 'SLEWRATE >= 5v/1ns' {} // where slewrate is of type 'Real'
157:   If 'REAL >= 5v' {}       // where realvar is of type 'Real'
158:   If 'REAL >= 5ma' {}     // where realvar is of type 'Real'
159:   If 'REAL >= 5e-3' {}    // where realvar is of type 'Real'
160: } // end Pattern
161:
162: Timing {
163:   WaveformTable WFT {
164:     Waveforms { SIG_NAME {
165:       01 { '25ns' D/U; }  // constant time
166:       01 { 'REAL' D/U; }  // spec variable
167:       01 { 'VOLTS' D/U; } // illogical, but allowed
168:     }
169:   }
170: } // end Timing
```

### 6.8 SignalVariable expression (sigvar_expr)

Signal variables are like signal groups except that they are not associated with actual signals. A `sigvar_expr` is a mechanism for extracting the data from a signal variable in order to create a `wfc_list`. A SignalVariable is assigned a WFC_LIST value, which may then be assigned to actual signals or groups. A `wfc_list` value shall be comprised of either a list of WFC’s, variable of type SignalVariable, or WFCConstant. It is an error to manipulate a Waveform-Character list with operators or functions.

The application of signal variables is to hold wfc list data and to pass signal data from pattern vectors, macro calls or procedure calls to macros, procedures or other vectors.
When a signal variable is set in a vector or a condition statement (\( V \{ SV=1111; \} \)) the data is immediately available for use. The contents of the signal variable are maintained upon a call to a procedure and the return; and the content is maintained on the invocation and exit from a Macro.

When a signal variable is set as a parameter to a macro or procedure call (\( \text{Macro} \{ SV=1111; \} \Call \{ SV \{1111; \ 0101; \ \} \} \)), then the data is not available until the the \# or \% operator is used (\( C \{ SV=\#; \} \)). This is the same behavior as for any signal or signal group parameter.

The following are examples of signal variable usage:

171: \text{STIL 1.0} \{ \text{Design DI5;} \}
172: \text{Header} \{}
173: \hspace{1em} \text{Source } "\text{P1450.1 Working-Draft 16, Sep 17, 2003}";
174: \hspace{1em} \text{Ann } (* \text{clause 6.8} *)
175: \}
176:
177: \text{Signals} \{}
178: \hspace{1em} X[1..5] \text{ In;}
179: \}
180:
181: \text{Variables} \{}
182: \hspace{1em} \text{SignalVariable SIG_VAR[1..5];}
183: \}
184:
185: \text{SignalGroups} \{}
186: \hspace{1em} \text{GRP_X = } 'X[1..5]'\;
187: \}
188:
189: \text{MacroDefs} \{}
190: \hspace{1em} \text{APPLY_VAR } \{}
191: \hspace{3em} C \{ \text{SIG_VAR[1..5] = \#; } \}
192: \hspace{3em} V \{ \text{GRP_X = } \text{WSIG_VAR[1..5]; } \}
193: \hspace{1em} \text{// error if above two lines were changed to:}
194: \hspace{3em} \text{// \ V \{ SIG_VAR[1..5] = \#; GRP_X = } \text{WSIG_VAR[1..5]; } \}
195: \hspace{1em} \}
196: \hspace{1em} \text{APPLY_TWO_VARS } \{}
197: \hspace{3em} C \{ \text{SIG_VAR[1..5] = \#; } \}
198: \hspace{3em} V \{ \text{GRP_X = } \text{WSIG_VAR[1..5]; } \} \text{ // apply 11111}
199: \hspace{3em} C \{ \text{SIG_VAR[1..5] = \#; } \}
200: \hspace{3em} V \{ \text{GRP_X = } \text{WSIG_VAR[1..5]; } \} \text{ // apply 00000}
201: \}
202: \}
203:
204: \text{Pattern PAT } \{}
205: \hspace{1em} \text{// following macro call use signal variables as parameters}
206: \hspace{1em} \text{Macro APPLY_VAR } \{ \text{SIG_VAR[1..5] = 11100; } \}
207: \hspace{1em} \text{Macro APPLY_VAR } \{ \text{SIG_VAR[5 4 2 3 1] = 00011; } \}
208: \hspace{1em} \text{Macro APPLY_VAR } \{ \text{SIG_VAR[5..1] = ABBAB; } \}
209: \hspace{1em} \text{Macro APPLY_TWO_VARS } \{ \text{SIG_VAR[5..1] \{11111; 00000; \}} \}
210: \hspace{1em} \text{// following use signal variables in line}
211: \hspace{1em} C \{ \text{SIG_VAR[1..5] = 11001; } \}
212: \hspace{1em} V \{ \text{GRP_X = } \text{WSIG_VAR[1..5]; } \}
213: \}

Copyright © 2001 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.
6.9 Integer lists (integer_list)

An integer_list is used to specify an ordered list of integer values. The only allowed operator in an integer list is the ellipsis ".." which specifies a range of integers. An integer_list may contain either single integers, space-separated lists of integers or integer ranges. Integer constants may appear in place of integers. An integer range is represented by 2 integers (or constants) separated by "..". For example, '3..6' is equivalent to '3 4 5 6' and '4 .. 2' is equivalent to '4 3 2'. Spaces are allowed between the integer or Constant, and the ".." operator. The following code shows examples of integer lists:

```
214: STIL 1.0 { Design D15; }
215: Header {
216:   Source "P1450.1 Working-Draft 16, Sep 17, 2003";
217:   Ann {* clause 6.9 *}
218: }
219: Signal {
220:   SIG[1..5] In;
221:   SI1 In { ScanIn 100; }
222: }
223: ScanStructures S1 {
224:   ScanChain CHAIN1 {
225:     ScanLength 100;
226:     ScanCells CC[1..100];
227:     Pattern PAT {
228:       V { SIG[5 4 3 2 1] = 11001; }
229:       V { SIG[5 .. 1] = 00110; }
230:     }
```

6.10 MergedScan function

The MergedScan() function returns a boolean value which is used to determine if scan operations are to be applied in a merged or unmerged fashion. The behavior of merged and unmerged scan is described in clause 5.5 of STIL.0. The evaluation of this function shall be in the context of the STIL consumer environment. If the STIL consumer environment has no specific needs for altering the merged scan operation then this function shall return a default value of True.

This construct in conjunction with the If and Else constructs allow the STIL producer to generate STIL patterns/procedures/macros which represent both merged and unmerged scan operations. See annex N for an example of syntax and usage.

6.11 Scan cell names (scan_cell_name)

A scan_cell_name is used in a ScanStructures block (see “ScanStructures Syntax” on page 38) to define the name of the scan cell. The name can be expressed in several ways:

a) a scan cell name (following the naming rules of STIL.0 clause 6.8)

b) an indexed sequence of names defined by enclosing the indices in square brackets.

c) the inversion symbol "!"

d) a scan segment reference - A scan segment is a reference to scan cells that are defined as a ScanChain and are to be inserted in sequence in the currently being defined scan cell list.

The scan segment is defined as follows:

**(STRUCTNAME) ::CHAINNAME (::INSTANCENAME)**

where STRUCTNAME is the name of a ScanStructure block, CHAINNAME is the name of the a ScanChain in the ScanStructure block, and INSTANCENAME is a user-name that is assigned to the instance of the chain being
referenced. The STRUCTNAME is optional and if omitted, the referenced chain shall be in the same block as the reference. The INSTANCENAME is optional and is needed when there is more than one instance of a given chain.

When segment instances are created, then the cell names are referenced as CELLNAME::INSTANCENAME.

The following diagram shows how segments may be referenced:

**Case 1: Multiple Segment Instances**

```plaintext
ScanStructures STR {
    ScanChain SEG {
        ScanCells C1 C2 C3 C4;
    } 
    ScanChain CH1 {
        ScanCells A B C D ::SEG::INST1;
    } 
    ScanChain CH2 {
        ScanCells E F G H ::SEG::INST2;
    }
}
```

**Case 2: Shared Segments**

```plaintext
ScanStructures STR {
    ScanChain SEG {
        ScanCells C1 C2 C3 C4;
    } 
    ScanChain CH1 {
        ScanCells A B C D ::SEG;
    } 
    ScanChain CH2 {
        ScanCells E F G H ::SEG;
    }
}
```

The following code shows an example of each form of syntax:

```plaintext
231: STIL 1.0 { Design D15; } 
232: Header { 
233: Source "P1450.1 Working-Draft 16, Sep 17, 2003"; 
234: Ann {* clause 6.11 *} 
235: } 
236: ScanStructures STRUCT1 { 
237: ScanChain CHAIN1 { 
238: ScanCells { CXX[1..99]; } 
239: } 
240: } 
241: ScanStructures STRUCT2 { 
242: ScanChain CHAIN2 { 
243: ScanCells { CYY[1..99]; } 
244: } 
245: ScanChain CHAIN3 { 
246: ScanCells { 
247: CELLI1; 
248: CELL2 { CellIn MASTER ! SLAVE; } 
```
6.12 Scan cell reference expressions (cellref_expr)

Scan cell reference expressions (cellref_expr) define an ordered list of scan cells. The rules for cellref_expr are the same as defined for sigref_expr in clause 6.14 of STIL.0. The elements of a cellref_expr are scan_cell_names as defined in “Scan cell names (scan_cell_name)” on page 10, with the following additional rules:

a) The ! inversion symbol is not valid in a cellref_expr.

b) Scan segments cannot be referenced, only the cells that make up the segments (note: cells and groups are in separate name spaces). The cells of a segment are referenced as either CELLNAME or CELLNAME::INSTANCENAME.

Please see “ScanCellGroups example” on page 44 for code examples.