

1. Summary of changes to 1450.1 - 2/13/2003

6.7 SignalVariable Expression (*sigvar_expr*)

SignalVariable expressions are like sigref_expr, except that they are not assigned to actual signal names. They are either a single token , or an expression enclosed in single quotes. SignalVariable are assigned WaveformCharacter list values, which may then be assigned to actual signals or groups. A WaveformCharacter list value results by direct assignment of a WaveformCharacter list, variable of type SignalVariable, or variable of type SignalVariableEnum in a sigvar_expr. It is an error to manipulate a WaveformCharacter 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 of 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 (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:

```

1: STIL 1.0 { Design D15; }
2: Header {
3:   Source "P1450.1 Working-Draft 15, Feb 13, 2003";
4:   Ann {* clause 6.7 *}
5: }
6:
7: Signals {
8:   X[1..5] In;
9: }
10:
11: Variables {
12:   SignalVariable SIG_VAR[1..5];
13: }
14:
15: SignalGroups {
16:   GRP_X = 'X[1..5]';
17: }
18:
19: MacroDefs {
20:   APPLY_VAR {
21:     C { SIG_VAR[1..5] = #; }
22:     V { GRP_X = 'SIG_VAR[1..5]'; }
23:     //error if above two lines were changed to:
24: //  V { SIG_VAR[1..5] = #; GRP_X = 'SIG_VAR[1..5]'; }
25:   }
26:
27: Pattern PAT {
28:   //following macro call use signal variables as parameters
29:   Macro APPLY_VAR { SIG_VAR[1..5] = 11100; }
30:   Macro APPLY_VAR { SIG_VAR[5 4 2 3 1] = 00011; }
31:   Macro APPLY_VAR { SIG_VAR[5..1] = ABBAB; }
32:   //following use signal variables in line
33:   C { SIG_VAR[1..5] = 11001; }
34:   V { GRP_X = 'SIG_VAR[1..5]'; }
```

35: }

10.1 Variables Block Syntax

```

Variables (VARS_BLOCK_NAME) { (1)
  (IntegerConstant CONST_NAME = DECIMAL_INTEGER ; )* (2)
  (Integer VAR_NAME; )* (3)
  (Integer VAR_NAME {
    (Usage Test; )
    (InitialValue logical_expr; )
    (IntegerEnum INTEGER_ENUM_NAME; )
    (Values integer_list; )
  } )* // end Integer
  (IntegerEnum INTEGER_ENUM_NAME { (4)
    Values {
      (ENUM (DECIMAL_INTEGER); )+
    } // end Values
  } )* // end IntegerEnum
  (SignalVariable VAR_NAME; )*
  (SignalVariable VAR_NAME {
    (Base <Hex | Dec> WFC_LIST ; )
    (Alignment <MSB | LSB> ; )
  }
  Question: I think this statement should be removed. The way to define a signal variable of length 5 is SV[1..5].
  See example 6.7 and 10.2.
  (Length DECIMAL_INTEGER; )
  (InitialValue vec_data ; )
  (SignalVariableEnum SIGNAL_ENUM_NAME)
} )* // end SignalVariable
(SignalVariableEnum SIGNAL_ENUM_NAME { (6)
  (Base <Hex | Dec> WFC_LIST ; )
  (Alignment <MSB | LSB> ; )
  (Length DECIMAL_INTEGER; )
  Values {
    (ENUM vec_data ; )+
  } // end Values
} )* // end SignalVariableEnum
} // end Variables

```

10.2 Variables Example

```

36: STIL 1.0 { Design D15; }
37:   Header {
38:     Source "P1450.1 Working-Draft 15, Feb 13, 2003";
39:     Ann {* clause 11.2 *}
40:   }
41:   Variables {
42:     IntegerConstant BUSTOP = 15;
43:     Integer H1;
44:     Integer H2 { Usage Test; InitialValue 13; }
45:     Integer H3 { Values 2 4 6 8; }
46:     Integer H4 { Values 1..100; }
47:     IntegerEnum COLORS {
48:       Values { RED; GREEN; BLUE; }
49:     }

```

```

50:    Integer COLOR { IntegerEnum COLORS; }
51:    SignalVariable VX[1..5]{ initialValue 11000; }
52:    SignalVariable VY[7..1]{ Base Hex AB; initialValue FE; }
53:    SignalVariableEnum CODE {
54:        Values { RESET 00; RUN 01; EXTEST 10; INTEST 11; }
55:    }
56:    SignalVariable VZ[0..1] { SignalVariableEnum CODE; }
57: }
58: Signals {
59:     bus1[ BUSTOP .. 0 ] Inout;
60: }

```

10.3 Variable Scoping

Care should be taken when defining variables as to whether the variable is to be shared or unique to a pattern. Shared variables are useful for passing data, whereas unique (local) variables provide independent operation. The scope of a variable is determined by 1) whether the variables block is named or unnamed, and 2) how a named variables block is referenced. The following example shows the various usage models. This example uses integer variables, but the same scoping rules apply to signal variables and integer constants.

```

61: STIL 1.0 { Design D15; }
62:   Header {
63:     Source "P1450.1 Working-Draft 15, Feb 13, 2003";
64:     Ann { * clause 11.3 * }
65:   }
66: Variables { Integer GLOB_VAR; }
67: Variables SHARED { Integer SHARED_VAR {InitialValue 0;} }
68: Variables LOCAL { Integer LOCAL_VAR; }
69:
70: Procedures {
71:   PROC1 {
72:
73:   PatternBurst {
74:     Variables SHARED;
75:     PatList { PAT1; }
76:     ParallelPatList {
77:       PAT2 { Variables LOCAL; }
78:       PAT3 { Variables LOCAL; }
79:     }
80:   }
81:
82:   Pattern PAT1 {
83:     C { GLOBAL_VAR=1; }
84:     If 'GLOBAL_VAR==1' {} //true
85:     If 'SHARED_VAR==0' {} //true
86:     If 'LOCAL_VAR==3' {} //error - not defined in this context
87:     C { SHARED_VAR=2; }
88:   }
89:
90:   Pattern PAT2 {
91:     C { LOCAL_VAR=3; }
92:     If 'GLOBAL_VAR==1' {} //true
93:     If 'SHARED_VAR==2' {} //true
94:     If 'LOCAL_VAR==3' {} //true
95:   }

```

```

96:
97: Pattern PAT3 {
98:   C { LOCAL_VAR=4; }
99:   If 'GLOBAL_VAR==1' {} //true
100:  If 'SHARED_VAR==2' {} //true
101:  If 'LOCAL_VAR==4' {} //true
102: }
103:

```

21. Pragma Block

A Pragma is a block of code that is implementation dependant. A Pragma block is a means of embedding non-STIL code within a STIL file. As with standard annotations, the syntax within the pragma is not defined or limited in any way, except by the opening and closing brace/asterisk convention. The use of a Pragma block is dependent on the application understanding the specific named Pragma; a Pragma with a name not understood by the application is ignored.

The Pragma block is used to support application-specific constructs outside the definitions of this standard. For example, to embed external functionality such as a fragment of C-code or Perl-code as part of the information present for a STIL test, or to embed test-specific code understood by an application as part of STIL test data.

Be aware that the use of the Pragma block is extremely application-specific. This data is primarily intended to instruct specific applications on how to apply STIL constructs, for example to identify test resource allocations for a specific test environment. Functionality necessary for the correct operation of a STIL test program should not appear in a Pragma block as the interpretation of that functionality is non-standard and the application of Pragma constructs are inherently non-portable.

21.1 Syntax

Pragma NAME { * ... APPLICATION DEPENDANT SYNTAX ... * } (1)

(1) **Pragma**: keyword identifying an implementation dependant block of code. If the block is used at the top level of a STIL file, then it has global scope. As with UserKeywords, a pragma block is locally scoped to a containing STIL block.

NAME: A user defined name for the block.

{ * ... APPLICATION DEPENDANT SYNTAX ... * }: The content of the block is enclosed within brace/asterisk delimiters. The only restriction on this block of data is that it not contain the asterisk/brace character pair which shall signal the end of the pragma data to a STIL parser.