TDS

LANGUAGES GUIDE

VERSION

2007.1

Table of Contents

Table of Contents (cont)

Table of Contents (cont)

Chapter 1
Language Overview

1.1 Waveform Generation

The Waveform Generation Language (WGL) is a data description language. It is used to convey an editable ASCII representation of the data contained in the Waveform DataBase (WDB), allowing you to use your system's text editor to fully customize the database.

A binary format for the ScanState and Pattern sections is supported, to be used (if desired) in place of ASCII pattern data. (Do *not* edit a WGL file that contains binary pattern data; null pattern bits may be deleted by the editor.)

WGL supports both scan hardware and test program generation that uses defined variables and embedded equation expressions.

NOTE

WGL constructs supporting scan hardware and equations are preserved in the WDB only if you have a TDS WaveBridge that includes scan support and equation support for your tester.

WGL programs are contained in an ASCII file called a WGL file. In this chapter, the term "WGL file" is used to denote an ASCII file that contains a WGL program. The term "WGL program" denotes the programming constructs contained within the WGL file.

For a complete description of the WGL, refer to Chapter 2 in this guide.

1.2 Test Control

The Test Control Language (TCL) provides control over TDS WaveBridge module processing. This section provides a brief overview of the roles that TCL plays in TDS operations.

Three basic types of TCL files can serve as input to TDS operations:

- Tester file, which describes tester-specific characteristics, such as the range of legal pin numbers, the maximum number of pattern rows allowed, and minimum cycle length. Typically, you will use the default Tester file that is provided for each tester, but you can create a customized version as well. For more information, refer to Section 4.10 of the *Getting Started Guide*.
- Override TCL file, which is an optional file that contains tester operating parameters that supercede the same parameters that were previously read from the Tester file. An override TCL file annot contain any Burst Blocks because the output files are specified using the Interactive Setup Form or DeskTop parameters.
- User TCL file, which is an optional file that defines the operations to be performed, the input and output databases, and the override information described in the previous bullet. A user TCL file provides more control over the operation of the TDS operation than is available from the TDS Shell Interactive Setup Form or the DeskTop Properties window.

For a complete description of TCL statements and syntax, refer to Chapter 3: *Test Control Language* in this guide.

Chapter 2
Waveform Generation Language

2.1 Introduction

The Waveform Generation Language (WGL) is a data description language. It is used to convey an editable ASCII representation of the data contained in a Waveform DataBase (WDB), allowing you to use your system's text editor to fully customize the database.

2.2 When to Use WGL

Since you can easily convert an existing TDS Standard Events Format (SEF) database to a WDB using the WaveMaker Browser, and edit a new or existing database using the WaveMaker editors, you may have little occasion to use WGL. However, WGL permits you to modify some parts of the WDB that are not accessible by WaveMaker's editors.

Use WGL to:

- Transfer a WDB from one host platform type to another type. WDBs are not otherwise portable.
- View and edit the ATE-specific portions of the WDB. Such portions of the WDB are not accessible by WaveMaker's editors.
- Create a WDB solely from WGL. This permits users who have a TDS WaveBridge module, but do not have WaveMaker, to run WaveBridge with a WDB.
- Use binary pattern data from the CAE simulation as input to TDS. (For more information, see *Binary WGL* on page 2-103.)
- Use your favorite text editor to perform sophisticated text manipulation operations, such as search and replace. (Do *not* edit a WGL file that contains binary pattern data; null pattern bits may be deleted by the editor.)

WGL is designed to be used in conjunction with the TDS WGL In Converter and WGL Out Converter modules. For details on how to use the WGL In Converter and the WGL

Out Converter, see Chapter 16 in the *In Converters Guide* and Chapter 17 in the *Out Converters Guide*.

2.3 WGL and Wavemaker

Since WGL describes a WDB, the language necessarily reflects the structure of the WDB. If you have used the WaveMaker editors to view a WDB, you will recognize this similarity. Many of the entities (such as ATE Pin and DUT Pin fields) that are visible in WaveMaker's editors are easily identifiable in WGL. Some WGL structures, however, are associated with ATE-specific descriptions, and are not visible in the WaveMaker Editors. The WGL Formats program block is an example of such a structure.

An example of the similarity of structure between the WaveMaker editors and WGL program structure is the WaveMaker Timing Editor. The WaveMaker Timing Editor allows you to edit a TDS timing template, or TimePlate. The TimePlate contains slots for one or more signals (identified by signal, group, or bus name), a signal direction indicator, and a waveform track. Slots are the area in which the signal name or names are entered.

Figure 2-1 shows the Timing Editor's view of a TimePlate named `Fetch`. Note the TimePlate name, the signal names, the signal directions, and the waveform tracks; all of

these entities can be described using WGL. Timing channels are arbitrary entities that contain signal, group, or bus names, direction information, and event and timing data.

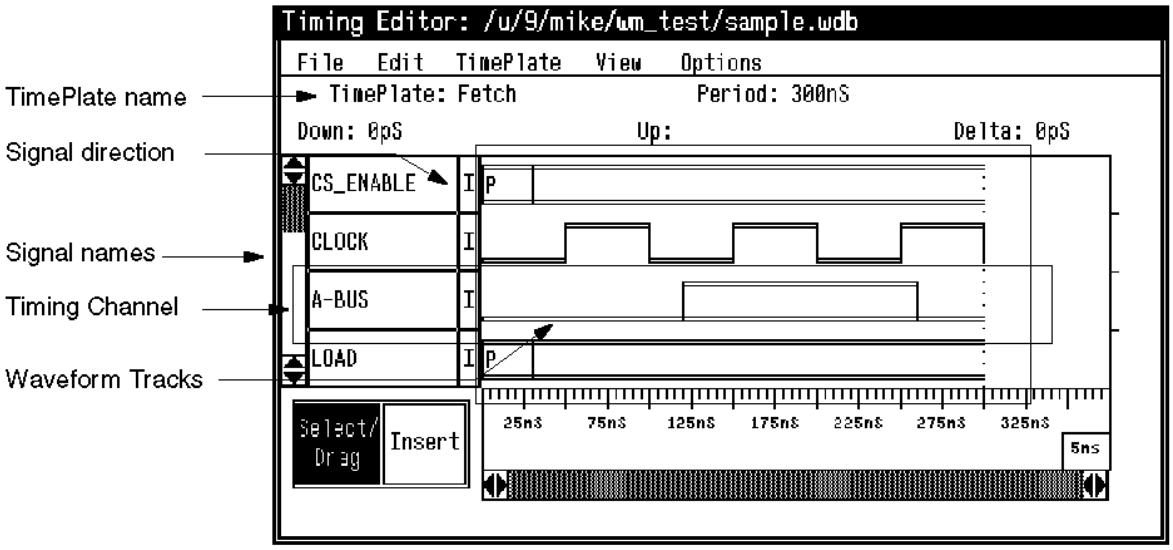

Figure 2-1. WaveMaker Timing Editor showing the TimePlate Fetch

The corresponding WGL description of the TimePlate `Fetch` is shown in the following example. Note how the TimePlate name, the signal name, direction, waveform track, and channel correspond to the same entities shown in the Timing Editor.

Start Example

```
timeplate Fetch period 300nS


  CS_ENABLE := input[0pS:P, 30nS:S];
 CLOCK    := input[0pS:D, 50nS:U, 100nS:D, 150nS:U, 200nS:D, 250nS:U,
300nS:D];
  A-BUS     := input[0pS:D, 120nS:S, 260nS:D];
  LOAD      := input[0pS:P, 30nS:S];
. . .
. . .
. . .
end
```

End Example

2.4 WGL Language Conventions

A WGL program is an ASCII text version of the information in the WDB.[1] The language is free-form (multiple white spaces are treated as a single white space and line returns are ignored) and limited to a line length of 512 characters. WGL reserved words are not case sensitive; keywords may be entered in any mix of upper and lower case letters. For user-defined names and pattern state characters, case is significant. The language uses the ASCII set of printable characters as legal input characters. WGL supports such features as macros, include files, in-line comments, post-compilation annotation, and many other operations normally available in programming languages.

2.4.1 WGL Syntax Notation Conventions

In describing the syntax of WGL, the following variation of the Backus-Naur Formalism (BNF) is used:

- Two colons followed by an equivalence sign (::=) denote a syntactic category to syntactic rules relationship.
- Double quotation marks (“ ”) or **Bold** typeface denote the literal use of a reserved word, typographical symbol, or parameter. If double quotation marks are to be used literally, they are enclosed within single quotation marks (‘ ’).

1. Binary pattern files use the WGL syntax notation plus have additional notations. *Binary File Format* on page 2-107.

- Angle brackets (< >) denote the use of a user-defined name, integer, or floating number.
- An equivalence sign (=) denotes the definition of a WGL reserved word or lexical primitive.
- Brackets ([ ]) denote optional syntax, appearing 0 or one time.
- Braces ({ }) denote an unspecified repetition (0 to *n* times) of the enclosed syntax. (This notation implies that the enclosed syntax is optional, since zero repetitions of a syntax is optional usage.)
- A vertical bar (|) denotes separate choices of syntax.
- Parentheses (( )) denote grouping of syntax options.

The use of *italics* in a text reference to a WGL syntactical element indicates higher-level BNF constructs. Such constructs are expanded to their full definition in the BNF accompanying the reference. For example, references to *FormatDecl* would appear in the appropriate BNF production as follows:

```
FormatDecl ::= <formatName> ":" "[" <TDSstate> { "," 
<TDSstate> } "]" ";"
```

User-defined identifiers, such as <TDSstate>, are defined in the *Glossary of WGL Terminology* on page 2-133.

NOTE

Do not confuse the BNF use of such typographical symbols as braces ({ }) with WGL's use of the same symbol. BNF uses braces to show a repetition of the action enclosed within the braces, while WGL uses braces to mark database annotations.

2.4.2 Comments

As in other programming languages, you can add explanatory comments to a WGL program to aid functional clarity. These comments are preceded by the pound sign (#), and are not included in the WDB when the WGL In Converter is run.

Comments can be inserted into any part of a WGL program except WGL annotations.[1] (See *Annotations* on page 2-63.) To insert a comment into a WGL program, enter a pound sign (#), followed by a text string. All characters on the line, starting with the pound sign and the terminating with the carriage return marking the end of the line, are included in the comment.

A complete BNF syntactical representation of the Comment feature follows.

```
Comment ::= "#" <any explanatory text> <end-of-line>
```

Example of WGL comments in a WGL program:

Start Example

```
# Signal block
signal
   clk: input;   # system clock
   dataReady: output;
   in: input;
   readWrite: bidir;
   data [0..31]: bidir;  # 32-bit data bus
   addr [0..15]: input;  # 16-bit address bus
end
```

End Example

2.4.3 Identifiers

An identifier is the alphanumeric name of a signal, bus, group, TimePlate, format, timing generator, pattern, subroutine, et cetera. Identifiers must begin with an alphabetic character, and may not contain white space (such as blanks, tabs,and newline characters) or any of the following delimiting characters:

(pound sign)
{ (left brace)
} (right brace)
“ (left double quotation marks)
” (right double quotation marks)
.. (double periods)
((left parenthesis)
) (right parenthesis)
+ (plus sign)
, (comma)
: (colon)
; (semi-colon)
[(left bracket)
] (right bracket)
. (period)

1. The binary pattern file cannot have comments, only annotations.

Identifiers must not conflict with any of the WGL reserved words. Any names that contain special characters or reserved words must be entered as a string surrounded by double quotation marks (“ ”).

In the WGL syntax descriptions in this chapter, identifiers are enclosed in angle brackets (< >).

2.4.4 Numbers

Unless noted otherwise, user-defined numeric values are integers that range from zero to the maximum integer that can be represented on your system’s architecture. Any exceptions are noted in the appropriate WGL syntax description section of this chapter.

In the WGL syntax descriptions in this chapter, user-defined numeric values are enclosed in angle brackets (< >).

2.4.5 Reserved Words

WGL reserves certain words as its linguistic set, from which data descriptions and procedural instructions can be synthesized. These reserved words can appear only in WGL statements in the correct syntax.

The following list shows the WGL reserved words:

atepin	event	last_force	ps	timegen
bidir	exprset	loop	radix	timeplate
binary	feedback	macro	reference	timeset
boolean	for	ms	registe	timing
call	force	mux	repeat	to
channel	force_or_z	ns	scan	us
compare	format	o	scancell	vector
decimal	hex	octal	scanchain	wavedata
direction	hexadecimal	out	scanstate	waveform
dont_care	i	output	signal	when
dutpin	in	pattern	skip	window
edge	initialp	period	subroutine	
end	input	pingroup	symbolic	
equationdefaults	integer	pmode	tg	
equationsheet	last_drive	procedure	time	

Unlike conventional programming languages, WGL cannot restrict or filter the use of reserved words. If a design has a signal name (or any other application-specific name) that

conflicts with any of the WGL reserved words, the signal name must be enclosed by double quotation marks (“ ”) to differentiate the signal name from the reserved word. This must be done throughout the program wherever the signal name occurs.

2.4.6 Strings

Strings are any sequence of characters surrounded by double quotation marks (“ ”). Within a string, if you want to use double quotation marks, you must precede each occurrence with a back slash (\ ). If you want to use a back slash within a string, you must precede each occurrence with a back slash. For example, the string:

\design“1”\

The equivalent WGL syntax is:

```
“\\design\“1\”\\”
```

2.5 WGL Syntax

WGL is a block-structured language. The body of the WGL program comprises one large structure, bracketed by opening and closing statements. Within the overall structure are smaller, more specialized structures, or blocks, each bracketed by opening and closing statements.

A discussion of WGL’s syntactic elements follows.

2.5.1 General Syntax

In its simplest form, a WGL source file uses the following syntax:

```
waveform <waveFormName>
{ WaveformBlocks }
end
```

Valid syntax for the optional *WaveformBlocks* is any of sixteen program sections. These sections are referred to as WGL programming blocks or blocks. The block names are:

EquationDefaults	ScanChain
EquationSheet	ScanState
Formats	Signals
GlobalMode	Subroutines
Patterns	Symbolics
Pin Groups	TimeGens
Registers	TimePlates
ScanCells	TimingSets

The block names act as block identifiers that categorize the information in each of the program blocks used. The blocks are optional and can occur in any order, subject to the restriction that all items in a block must be defined before they are used, and a pattern block must be defined before a subroutine that uses it is defined. It is possible to create an empty WDB, a WDB with only signals defined, a WDB with signals and timing defined, a WDB with only signals and patterns defined, or a WDB with all components defined (as represented by inclusion of all program blocks describing WDB objects).

A high-level BNF syntactical representation of the WGL program follows:

```
WaveformProgram ::= “waveform” <waveFormName> [ “()” ]
{ WaveformBlocks } “end”

WaveformBlocks ::= ( EquationSheet  | EquationDefaults |
GlobalMode  |
Formats |TimeGens | PinGroups | Signals |
TimingSets | Registers | TimePlates | Symbolics | Patterns
|
Subroutines | ScanCells | ScanChain | ScanState )

EquationSheet ::=   “equationsheet” <equationSheetName>
{ ExpessionDecl } “end”

EquationDefaults ::= “equationdefaults” DefaultsDecl “end”

GlobalMode ::=  “pmode” “[” PmodeOption “]” “;”

Formats ::=  “format” { FormatDecl } “end”

TimeGens ::=  “timegen” { TgDecl } “end”
```

```
PinGroups :=  “pingroup” { PinGroupDecl } “end”

Signals ::=  “signal” { SignalDecl } “end”

TimingSets ::=  “timeset” <tsNumber> { TgAssign } end”

Registers ::=  “register” “(” PinList “)” { RegisterDecl }
“end”

TimePlates ::= “timeplate” <timeplateName> TimePlate “end”

Symbolics ::=  “symbolic” SignalReference [ SymDirection]
Radix
SymbolicAssignment “end”

Patterns ::=  “pattern” <patternName> “(”
PatternParameters “)”
PatternRows “end”

Subroutines ::=  “subroutine” <subroutineName> “( )”
PatternRows “end”

ScanCells ::=  “scanCell” { ScanCellDecl } “end”

ScanChain ::=  “scanChain” { ChainDecl } “end”

ScanState ::=  “scanState” { ScanStateDecl } “end”
```

An example of a typical WGL program is:

Start Example

```
waveform  generic
   signal
      CS_ENABLE : input
   dutpin[P1:1]
   atepin[CSENAB:1];
      A-BUS [15..0] : input
   radix hexadecimal
   dutpin[P2:2, P3:3, P4:4, P5:5, P6:6,
   P7:7, P8:8, P9:9, P10:10, P11:11,
   P12:12, P13:13, P14:14, P15:15, P16:16,
   P17:17]
   atepin[ABUS15:2, ABUS14:3, ABUS13:4, ABUS12:5,
   ABUS11:6, ABUS10:7, ABUS9:8, ABUS8:9, ABUS7:10,
   ABUS6:11, ABUS5:12, ABUS4:13, ABUS3:14, ABUS2:15,
```

```
   ABUS1:16, ABUS0:17];
      LOAD : input
   dutpin[P18:18]
   atepin[LOAD:18];
.  .  .
.   end

   timeplate Fetch period 300nS
      CS_ENABLE := input[0pS:P, 30nS:S];
      A-BUS := input[0pS:D, 120nS:S, 260nS:D];
      LOAD := input[0pS:P, 100nS:S];
      ENP := input[0pS:P, 50nS:S];
      DR := input[0pS:P, 100nS:S];
      RO := input[0pS:U, 70nS:S, 180nS:U];
      D-BUS := output[0pS:X, 100nS:Q, 250nS:X];
      DB-BUS := output[0pS:X, 100nS:Q, 250nS:X];
      AD-BUS := input[0pS:P, 100nS:S];
   end
   timeplate R_W period 200nS
      CS_ENABLE := input[0pS:P, 30nS:S];
      A-BUS := input[0pS:D, 60nS:S, 190nS:D];
      LOAD := input[0pS:S];
      ENP := input[0pS:S];
      DR := input[0pS:S];
      RO := input[0pS:U, 40nS:S, 180nS:U];
      D-BUS := output[0pS:X, 60nS:Q, 190nS:X];
      DB-BUS := output[0pS:X, 40nS:Q, 180nS:X];
      AD-BUS := input[0pS:P, 60nS:S];
   end
.  .  .

   symbolic DB-BUS input radix hexadecimal
      RESET := [1ED8];
      JMP := [BE43];
      LDA := [062D];
   end
   symbolic DB-BUS output radix binary
   end

   pattern  group_ALL (CS_ENABLE,A-BUS,LOAD,ENP,DR,RO,D-BUS,DB-BUS:I,DB-BUS:O,
            AD-B S:I,AD-BUS:O)
      repeat 5
   vector(0, 0pS, Startup) := [1 FFFF 0 0 0 1 3D66 RESET ---------------- AD -- ];
{ This is the COMMENT for the first row. This STARTUP TimePlate allows the tester
to start ALL stimulus at the LOW state, and initializes the device.}
```

```
    vector(5, 2.5uS, Fetch) := [1 ADBB 0 0 1 0 3CDA ---- 0011111000000100 BB -- ];

{ During the FETCH cycle, the address on the A-Bus is “fetched” and will be valid
(displayed) on the D-Bus until after the next FETCH cycle.}

     vector(6, 2.8uS, R_W) := [0 0C13 1 0 1 1 ADBB  ---- 0010100100101101 84 -- ];
     vector(7, 3uS, Write) := [0 8D18 0 1 0 0 ADBB JMP  ---------------- -- 99 ];
{    The WRITE cycle contains “mid-cycle I/O” on the DB-Bus.}
     vector(8, 3.4uS, Fetch) := [0 EF57 0 1 0 1 ADBB  ---- 1100001001000100 98 -- ];
     vector(9, 3.7uS, R_W) := [0 82DD 1 0 1 0 EF57  ---- 0110000001110101 7B -- ];
call sub1();
     vector(16, 5.7uS, Write) := [0 8D18 0 1 0 0 ADBB JMP  ---------------- -- 99 ];
     vector(17, 6.1uS, Fetch) := [0 EF57 0 1 0 1 ADBB  ---- 1100001001000100 98 -- ];
     vector(18, 6.4uS, R_W) := [0 82DD 1 0 1 0 EF57  ---- 0110000001110101 7B -- ];
     vector(19, 6.6uS, Write) := [0 2927 1 1 0 0 AA03 LDA  ---------------- -- 81 ];
     vector(20, 7uS, Fetch) := [0 84F5 0 1 1 1 AA03  ---- 0100000110110111 A4 -- ];
     vector(21, 7.3uS, R_W) := [1 8DB4 1 0 1 1 84F5  ---- 1100001100010001 97 -- ];
call sub1();
     vector(28, 9.3uS, Write) := [0 7306 1 1 0 0 84F5  00DF ---------------- -- 17 ];
.  .  .

   vector(107, 33.1uS, Fetch) := [0 9DF1 1 1 0 1 140F  ---- 0010100101000010 98 -- ];
{    This is the LAST vector row}
   end


subroutine sub1()
      vector(0, 0pS, Write) := [1 59E7 1 0 1 1 EF57  5FC9 ---------------- -- 65 ];
      vector(1, 400nS, Fetch) := [0 E327 0 0 0 0 EF57  ---- 0111100101000100 BF -- ];
      vector(2, 700nS, R_W)  := [0 28E7 1 0 1 1 E327  ---- 1101001110000110 CA -- ];
      vector(3, 900nS, Write) := [1 898B 1 1 0 1 E327  5F8B ---------------- -- A0 ];
      vector(4, 1.3uS, Fetch) := [1 AA03 0 0 0 1 E327  ---- 1001111010101101 83 -- ];
      vector(5, 1.6uS, R_W)  := [0 1ECD 1 0 1 0 AA03  ---- 0010001101010101 23 -- ];
      end

end
```

End Example

2.5.2 Program Block Syntax

All WGL program blocks begin with one of the WGL reserved word block names, and terminate with the reserved word end. Between these two delimiting reserved words are one or more WGL statements used to define data. These WGL statements themselves are subdivided into smaller structures that address more specific operations, such as setting timing for individual signal channels.

A colon (:) is used to assign an attribute (such as force or input) to an identifier. A colon-and-equivalence (:=) is used as an assignment operator, assigning a value (such as a numeric value) to an identifier. See the previous example of a typical WGL program for these usages.

In permitted instances commas and semi-colons are used as delimiters. When several parameters occupy the same line, each entry may be delimited by a comma (,). Each separate WGL statement must be delimited by a semicolon (;). Check the BNF notation for each WGL block for details of permissible usages. See the WGL program example on page 2-10.

Generally speaking, the WGL blocks are of three types: generic, tester-specific, and equation-specific.

The generic blocks let you address data that are related to the test waveforms.

The tester-specific blocks allow you to specify WDB data values that are directly related to the type of tester you are using.

The equation-specific blocks let you assign expressions and constant values to variables that can later be used in place of time values in timing sets and TimePlates. The results of these equations are then included in the test program you can generate using a TDS WaveBridge module.

While it is useful to consider the WGL blocks in these three general categories, it is important to remember that some blocks contain generic, tester-specific, and equation-specific components. For example, Signals blocks and TimePlates blocks contain both generic and tester-specific WGL statements. TimePlate blocks and TimingSet blocks contain generic, tester-specific, and equation-specific WGL statements.

Table 2-1 defines the block type of each of the sixteen WGL program blocks.[1]

Table 2-1. WGL program block types

WGL Program Block	*Type*
EquationDefaults	equation-specific
EquationSheet	equation-specific
Formats	tester-specific
GlobalMode	generic
Patterns	generic
Pin Groups	tester-specific
Registers	tester-specific
Scan Cells	generic
Scan Chain	generic
Scan State	generic
Signals	generic, tester-specific
Subroutines	generic
Symbolics	generic
TimeGens	tester-specific
TimePlates	generic, tester-specific, equation-specific
TimingSets	tester-specific, equation-specific

2.5.3 Generic Program Blocks

This section discusses the specific syntax for each of the generic program blocks. The following list shows the WGL generic program blocks:

Signals
Scan Cells
Scan State
Scan Chain
TimePlates
Patterns
Subroutines
Symbolics

1. WGL constructs supporting equations are preserved in the WDB only if you have a TDS WaveBridge that includes equation support for your tester.

Use the generic program blocks to define WDB objects that are not specific to any tester. The generic program blocks are presented in the likely order of use when creating a WDB.

2.5.3.1 Signals

The Signals block is used to declare four types of signal definitions: single-bit signals, multi-bit buses, groups, and multiplexed signals or buses. Groups may include signals, buses, or other groups.

Signal attributes must be defined in the same entry. For example,

```
signal
reset: input;

...

reset: radix symbolic;
end
```

is incorrect. The correct way to state these signal attributes is:

```
signal
reset: input radix symbolic;
end
```

The syntax of the WGL Signals block is:

```
signal
SignalDecl
end
```

A complete BNF syntactical representation of the Signals block follows:

```
Signals ::=  “signal” { SignalDecl } “end”

SignalDecl ::= <signalName> [ BusOrGroup ] [ “:”
SignalAttributes ]
[ Pstate ] “;”

BusOrGroup ::= ( BusRange | GroupMembers | MuxMembers )

BusRange ::= “[” <bitNumber> “..” <bitNumber> “]”

GroupMembers ::= “[” [ SignalReference { “,”
SignalReference } ] “]”
```

```
SignalReference ::= <signalName> [ Range ]

Range ::= “[” <bitNumber> [ “..” <bitNumber> ] “]”

MuxMembers ::= [ MuxPartList ] [ Range ]

MuxPartList ::= “[” <muxPartName> “,” <muxPartName> [ {
“,” <muxPartName> } ] “]”

SignalAttributes ::= ([“mux” ] [ Direction ] ) { Strobe }
[ Radix ] [ DutPins ]
[ AtePins ]

Direction ::= ( “input” | “output” | “bidir” ) [ (
“reference” | “timing” ) ]

Strobe ::= ( “in” | “out” ) “when” “[” <validityClause>
“]”

Radix ::= “radix” ( “binary” | “octal” | “decimal” | “hex”
| “hexadecimal” | “symbolic” )

DutPins ::= “dutpin” “[” DutPinGroup { “,” DutPinGroup }
“]”

DutPinGroup ::= ( PinInfo | “(” PinInfo { “,” PinInfo }
“)” )

PinInfo ::= PinName| PinNumber

PinName ::= <pinName> [PinNumber]

PinNumber ::= “:” <pinNumber>

AtePins ::= “atepin” “[” AtePinGroup { “,” AtePinGroup }
“]”

AtePinGroup ::= ( AtePinInfo | “(” AtePinInfo { “,”
AtePinInfo } “)” )

AtePinInfo ::= PinInfo [ “tg” “[” <timeGenName>  { “,”
<timeGenName> } “]” ]

Pstate ::= “initialp” “[” <TDSstate> “]”
```

The *SignalDecl* begins with a user-defined identifier or string. The *SignalDecl* can be any of four types:

- Single-bit signals
- Multi-bit buses
- Groups of signals, buses, or other groups
- Multiplexed signals or buses

Single-Bit Signals

Single-bit signals are defined by an identifier followed by a list of attributes. The following is an example of a WGL Signals block with only single-bit signals defined.

Start Example

```
signal
   clk: input;
   dataReady: output;
   in_1: input;
   readWrite: bidir;
end
```

End Example

Buses

Buses are defined by an identifier followed by the range of the bus, enclosed in brackets ([ ]). The total, combined number of single-bit signals and buses that can be defined is limited to 16384.

The following is an example of a WGL Signals block with single-bit signals and buses defined.

Start Example

```
signal
   clk: input;  # system clock
   dataReady: output;
   in_1: input;
   readWrite: bidir;
   data [0..31]: bidir; # 32-bit data bus
   addr [0..15]: input; # 16-bit address bus
end
```

End Example

Groups

Groups are defined by a list of previously defined single-bit signals, buses, bus members, or other groups. Groups can name single-bit signals, buses, bus members, or groups only once in the list. The number of groups used does not contribute to the combined total of 16384.

The following is an example of a WGL Signals block with single-bit signals, buses, and groups defined:

Start Example

```
signal
   clk: input;  # system clock
   dataReady: output;
   in_1: input;
   readWrite: bidir;
   data [0..31]: bidir; # 32-bit data bus
   addr [0..15]: input; # 16-bit address bus
   busses [data, addr]; # both busses together
   data0_8 [data[0..8]];
   oddAddr [addr[1], addr[3], addr[5], addr[7]];
   inputs [clk, in];
end
```

End Example

There are predefined groups available that you can use in any correct syntax for groups. The predefined group names must be entered as upper-case characters, as shown. They are:

ALL

This predefined group contains all signals, buses, and multiplexed signals and buses (but not multiplexed parts). Groups are not included.

ALLINPUT

This predefined group contains all signals, buses, and multiplexed signals and buses (but not multiplexed parts) with the `input` signal direction attribute.

ALLOUTPUT

This predefined group contains all signals, buses, and multiplexed signals and buses (but not multiplexed parts) with the `output` signal direction attribute.

ALLBIDIR

This predefined group contains all signals, buses, and multiplexed signals and buses (but not multiplexed parts) with the `bidir` (bidirectional) signal direction attribute.

ALLMUX

This predefined group contains all multiplexed signals and multiplexed buses (but not multiplexed parts) with the `mux` (multiplexed) signal attribute.

There is no limit to the number of groups that can be defined.

Multiplexed Signals or Buses

Multiplexed signals are defined by an identifier followed by a list of multiplexed parts, enclosed in brackets ([ ]); multiplexed buses are defined by an identifier followed by a list of multiplexed parts, enclosed in brackets ([ ]), and followed by the *Range*, which is also enclosed within brackets ([ ]).

Do not confuse multiplexed parts (<muxPartName>s) with signals; multiplexed parts describe the ATE resources used to supply pattern data to a multiplexed signal or bus. Multiplexed parts function in much the same manner as signals in the TimePlates, carrying timing parameters and pattern data that is eventually associated with a multiplexed signal defined in the Signals block.

An example of a WGL Signals block with definitions of a multiplexed signal, a single-bit signal, and a multiplexed bus follows. Note the use of the mux attribute:

Start Example

```
signal
   fastClock [edge1, edge2]:     mux input;     # Multiplexed parts edge1,
                                                # edge2 on multiplexed
                                                # signal fastClock
   rd/_wr:output;
   Databus [bus1, bus2] [0..31]: mux bidir;     # Multiplexed parts bus1,
                                                # bus2 on multiplexed
                                                # bus Databus
end
```

End Example

When waveforms are more complicated than those supported by the target tester's formatting set, multiplexed signals and buses are typically used to generate test programs that contain pin multiplexing for these complicated waveforms. By using this ability, you can multiply the effective frequency of the tester. If multiple pattern bits are needed to define a waveform (for example, multiple pulses in a single tester cycle), you should define such signals or buses as multiplexed signals or buses.

Following the optional *BusOrGroup* syntax are other attributes that are associated with the current signal declaration. If you are defining a group, only the radix attribute is applicable.

atepin

ATE pin information is defined in the Signals block using the reserved word atepin. The *AtePinInfo* syntax is used to describe the mapping of the current signal declaration to tester pins and the binding between a tester pin and its timing generators. The atepin value is an alphanumeric string. When multiple ATE pins are specified for a multi-bit bus, the mapping is one-to-one unless parentheses are used to group two or more pin declarations with a single signal.

ATE timing generator information is also defined in the signals block. The timing generator binding is introduced with the reserved word tg. The *tgName* is the name of the tester-specific timing generator that is generating the timing for all the edges of the signals in the current signal declaration. Multiple *tgNames* indicate that the timing generators are being multiplexed or the existing timing generators (defined in a TimeGens block) are responsible for multiple edges.

NOTE

Pin information and timing generator information are both tester-specific

The following is an example of a WGL Signals block with dutpin and atepin attributes defined:

Start Example

```
signal
   clk   : input  dutpin [c:p1] atepin [fclock:123 tg [ACLK1] ];
   dr    : input  dutpin [r:p2] atepin [p124:124 tg [BCLK1,CCLK1] ];
   data  : output dutpin [d:p3] atepin [p2:2 tg [STRB1]];
end
```

End Example

direction

The direction attribute describes the direction of a signal and controls how the signal is used in test program generation.

A signal may be forcing **(**input), sensing (output), or both forcing and sensing at different times (bidir); the default is input. A direction may not be specified for groups. If a bus has a direction of input or output, all the bits of the bus must have the same direction; otherwise, only bidir is legal.

To control how the signal is used in test program generation, you can choose either reference or timing. If neither of these is specified, the signal is considered in TimePlate matching and tester program generation. If the clause is used with timing specified, the signal is considered in TimePlate binding but not in test program generation. If reference is specified, the signal is not considered in either TimePlate binding or test program generation. When this clause is used, complete WGL syntax is still required for the signal (signal, TimePlate track, and data).

The following is an example of a WGL Signals block with signals `I1` and `I3` use restricted:

Start Example

```
signal
   I1 : input reference;
   I2 : input;
   I3 : input timing;
   .  .  .
end
```

End Example

Strobe Clause

Signals and buses may include strobe clauses after their direction attributes. Use strobe clauses to specify:

1. that an input or output signal is valid only when another signal takes a certain value, or
2. the conditions under which a bidirectional signal is an input, and those under which it is an output.

Strobe clauses take the form

in|**out when** [*signal_name state_character*]

See Chapter 4 in the *Getting Started Guide* for detailed information about Signal Definition file syntax.

In the following example, the bus named `data` only contains valid output when signal `cntrl` has the value ‘D’. In addition, the bidirectional signal `dr` is an input when `cntrl` has the value ‘D’, and an output when `cntrl` has the value ‘U’:

Start Example

```
signal
   cntrl       : input;
   dr          : bidir in when [cntrl D] out when [cntrl U];
   data[7..0]  : output out when [cntrl D];
end
```

End Example

dutpin

The dutpin attribute specifies the names (and optional numbers) of the pins on the device-under-test associated with the signal. The dutpin value is an alphanumeric string. If a device has multiple pins dedicated to the same signal, or different pins in use when a bidirectional signal is input or output, more than one pin may be specified. dutpin may not be specified for groups.

If multiple pins are specified in a multi-bit bus declaration, the mapping is assumed to be one-to-one between the bus elements and the pins, in a left-to-right, most-significant-pin to least-significant-pin order. Other distributions of pins to signals (such as that required for multiplexed pins) can be accomplished by grouping the pin declarations within parentheses. This indicates that multiple pins are bound to single-bit bus member.

The following is an example of a WGL Signals block with dutpin attribute defined:

Start Example

```
signal
   clk  : input dutpin [c:1];
   data[0..7]: bidir
      dutpin [(d0i, d0o), (d1i, d1o), (d2i, d2o), (d3i, d3o),
            (d4i, d4o), (d5i, d5o), (d6i, d6o), (d7i, d70)];
end
```

End Example

mux

The mux attribute defines a signal or bus as a multiplexed signal or bus. The signal or bus receives pattern data from a list of multiplexed parts. If the multiplexed parts are themselves buses, these buses must be followed by the range of the bus enclosed in brackets ([ ]).

The names of the multiplexed parts must be identified for the first time in the current signal definition; it is illegal to use the names of other signals, groups, or buses that have been previously defined in the Signals block of the WGL file.

See page 2-19 for an example of the use of the mux attribute.

initialp

Each signal definition may have an optional initialp state specified. P states are resolved to this state the first cycle of the waveform. Any legal TDS state may be specified. If the

initialp clause is omitted, the default is D (FORCE_LO). initialp may not be specified for groups.

The following is an example of a WGL Signals block with initialp specified for signals `clk` and `bus`:

Start Example

```
signal
   clk      : input initialp[U];
   bus[0..7]: output initialp[X];
end
```

End Example

Radix

The radix attribute describes the base in which the pattern data for the bus is described in the Patterns block. The radix attribute can be `binary`, `hexadecimal`, `octal`, `decimal`, or `symbolic`. Only binary and symbolic are legal for single-bit signals. The default radix is binary when the radix attribute is unspecified.

The `symbolic` radix indicates that identifiers defined in subsequent symbolic blocks may be used in pattern vectors. Decimal radix may only be specified for buses and groups with 32 or fewer scalar member signals.

NOTE

Legal binary pattern characters are `1`, `0`, `Z`, `X`, and `-`; if you specify a non-binary radix (hexadecimal, decimal, octal, symbolic) in the WGL file, and edit the WDB using the WaveMaker Pattern Editor, do not use the 1 or 0 binary pattern characters in conjunction with the Z, X, or - characters. Since the X, Z, or - characters represent an ambiguous data bit, the pattern data for the entire digit (four bits for hexadecimal, three bits for octal, one or two bits for decimal, or n bits for symbolic) is discarded and replaced with a question mark (?). If all the bits are Z, the hexadecimal or octal digit is replaced with Z. If all the bits are X, the hexadecimal or octal digit is replaced with X.

2.5.3.2 Scan Cells

The Scan Cells block is used to represent internal storage registers of a device that may be loaded or observed using serial shift scan circuitry. The total number of scan cells allowed in a single WGL In file is limited to 32767.

It is important to distinguish scan cells from signals. WDB stores continuous waveform information for signals. Scan cells, however, can represent only logic states at particular instants. Scan cells do not have direction and there is no direct association with ATE or DUT pins. Scan cells cannot be referenced in TimePlates or pattern parameter lists.

The syntax of the WGL Scan Cells block is:

scancell
ScanCellDecl
end

A complete BNF syntactical representation of the Scan Cells block follows:

```
ScanCells ::=  “scanCell” { ScanCellDecl } “end”

ScanCellDecl ::= <cellName> [ ScanGroup ] [ “:” Radix ]
“;”

ScanGroup ::= “[” [ ScanRange | ScanGroupMembers ] “]”

ScanRange ::= <bitNumber> “..” <bitNumber>

ScanGroupMembers ::= CellReference { “,” CellReference }

CellReference ::= ( <cellName> [ Range ] )

Range ::= “[” <bitNumber> [ “..” <bitNumber> ] “]”

Radix ::= “radix” ( “binary” | “octal” | “decimal” | “hex”
| “hexadecimal” | “symbolic” )
```

The ScanGroup statement allows you to specify a logical grouping of scan cells. The scan cells in a group may be from multiple scan chains. Each ScanGroupMember must be previously defined, unless it is the name of another scan group.

The optional Radix specification for scan groups and registers is used in scan state vectors. The supported radices are implemented by using the WGL reserved words: binary, hex, octal, decimal, and symbolic.

An example of a ScanCells block is:

Start Example

```
scancell
   latchA;
   latchB;
   datareg[0..7]: radix hexadecimal;
   group_1[latchA, latchB, datareg[7]]: radix octal;
end
```

End Example

The Scan Cells block example names scan-able cells within the device. Cells may be single-bit latches, such as `latchA`, or multi-bit registers, such as `datareg`. Logical groups of scan cells, such as `group_1`, also may be specified.

A complete example of WGL scan structures is provided on page 2-96 of this chapter.

NOTE

Regardless of the order in which signals are defined in a scangroup declaration, it is the order in which they are defined in a scanchain declaration that determines how pattern bits are assigned to a cell.

2.5.3.3 Scan State

Each state declaration in a Scan State block defines the entire state of the set of all scan cells at some instant in time. The goal of input scanning is to achieve that state; the goal of output scanning is to observe that state. A scan state vector may be referenced from zero or more scan pattern rows. It may take multiple scan chains to load or observe all the cells in a state.

A binary format of the scan vectors is supported. For more information, see *Binary WGL* on page 2-103. This capability allows you to use binary data from a CAE simulation as input to TDS.

The syntax of the WGL Scan State block is:

`scanstate`
`ScanStateDecl`
`end`

A complete BNF syntactical representation of the Scan State block follows:

```
ScanState ::=  “scanState” { ScanStateDecl } “end”

ScanStateDecl ::= <stateName> “:=” { StateVectorElement }
“;”

StateVectorElement ::= <chainName> “(“ { <stateString> }
“)”
```

ScanStateDecl specifies a name for the scan state and the values of all the scan cells for that state. The <stateName> is an identifier; some special characters may be used if the <stateName> is enclosed within double quotation marks (“ ”). <stateNames> occupy their own name space but must be unique among all other states. The StateVectorElements are assigned by naming the cell, register, cell group, or chain and appending a <stateString> value in parentheses. The <stateString> is interpreted in the radix of the associated cell reference similar to the technique used for pattern states. The WGL Out Converter always generates state vectors using `ALLSCAN` as the only cell reference. The <chainName> is an identifier and must be unique among all other scan chain names.

The value of any cell not specified in the scan state declaration is implicitly X, the TDS state character representing a compare unknown state. The actual value used by a tester to drive X is technology-dependent and programmed in TDS Test Control Language (TCL). If that portion of the state is scanned out, the comparison is masked. For more information on how to use TCL, see Chapter 3: *Test Control Language* in this guide.

Legal characters in the *stateString* are `0`, `1`, `Z`, and `X` for binary radix, `0-9`, `A-F`, `Z`, and `X` for hexadecimal radix, `0-7`, `Z`, and `X` for octal radix, and `0-9` for decimal radix.

The following is an example of a Scan State block. The bit order of the scan group `ALLSCAN` is the order that the scan cells (and scan registers) are defined in the Scan Cell block of the WGL file. If you choose to specify scan state vectors using the ALLSCAN group, you must specify a bit for every scan cell element that is listed in the Scan Cell block.

Start Example

```
scanState
   state1 := latchA(1) latchB(0) datareg(3F);
   state2 := latchA(0) latchB(1) datareg(01);
   state3 := ALLSCAN(XX00000000);
   stateX := ;
end
```

End Example

The `stateX` state declaration in this example sets up a state of all X (compare unknown) values.

A complete example of WGL scan structures is provided on page 2-96 of this chapter.

2.5.3.4 Scan Chain

The Scan Chain block defines the configuration of a circuit path connecting edge signals to scan cells and inverters. Each chain is named with an identifier or quoted string that must be unique among signals, scan cells, buses, scan registers, groups, and other scan chains.

The syntax of the WGL Scan Chain block is:

scanchain
ChainDecl
end

A complete BNF syntactical representation of the Scan Chain block follows:

```
ScanChain ::=  “scanChain” { ChainDecl } “end”

ChainDecl ::= <chainName> “[” ChainMembers “]” [ “:” Radix
] “;”

ChainMembers ::= ( InEdgeSignal | ChainMemList |
OutEdgeSignal )

InEdgeSignal ::= SignalReference ”,”

OutEdgeSignal ::= “,” SignalReference

ChainMemList ::= ChainMemReference { “ , ”
ChainMemReference }
```

```
SignalReference ::= <signalName>

ChainMemReference ::= ( CellReference | “!” )

CellReference ::= ( <cellName> [ Range ] )

Range ::= “[” <bitNumber> [ “..” <bitNumber> ] “]”

Radix ::= “radix” ( “binary” | “octal” | “decimal” | “hex”
| “hexadecimal” | “symbolic” )
```

The <chainName> is an identifier and must be unique among all other scan chain names.

The ChainMembers list represents the ordered sequence of scan chain elements where the implied shift direction is left-to-right.

Either the InEdgeSignal or the OutEdgeSignal can be omitted, but if the chain is directly referenced by a scan pattern row, at least one must be present.

The SignalReference for an InEdgeSignal must have been previously declared as a 1-bit wide input or bidirectional signal. The SignalReference for an OutEdgeSignal must have been previously defined as a 1-bit wide output or bidirectionalsignal. The reserved symbol ! indicates state inversion. Scan chains may be members of other chains as long as the declaration is not recursive.

If the Radix is omitted, binary radix is supplied by default.

An example of a Scan Chain block is:

Start Example

```
scanchain
   chain1 [ SC1_IN, datareg[0], latchA, datareg[2], SC1_OUT] : radix octal;
   chain2 [ SC2_IN, datareg[1], !, datareg[7], datareg[5], latchB,
   datareg[4], !, datareg[6]];
end
```

End Example

The Scan Chain block example shows the order of scan cells on two physical chains. The first and last elements of the `chain1` cell list are the names of edge signals `SC1_IN` and `SC1_OUT`, which must have been previously defined in a Signals block. `chain2` has an input signal `SC2_IN` but no corresponding output signal. Therefore, `chain2` may be used to control the state of the listed scan cells but there is no way to observe their state. The reserved symbol ! appears twice in the `chain2` cell list. This indicates that states are

inverted when they shift between `datareg[1]` and `datareg[7]`, and between `datareg[4]` and `datareg[6]`.

Parallel scan chains are supported, but the scan chains can not be identical. The following is an example of the legal use of parallel scan chains.

Start Example

```
waveform t1
scancell
   latch1; latch2; latch3; latch4;
   latch5; latch6; latch7; latch8;
end
scanstate
   state1 := latch1(0) latch2(0) latch3(0) latch4(0);
   state2 := latch1(0) latch2(0) latch3(0) latch4(1);
   state3 := latch1(0) latch2(0) latch3(1) latch4(1);
   state4 := latch1(0) latch2(1) latch3(0) latch4(0);
   state5 := latch1(0) latch2(1) latch3(0) latch4(1);

   estate1 := latch5(1) latch6(1) latch7(1) latch8(0);
   estate2 := latch5(1) latch6(1) latch7(0) latch8(1);
   estate3 := latch5(1) latch6(1) latch7(0) latch8(0);
   estate4 := latch5(1) latch6(0) latch7(1) latch8(1);
   estate5 := latch5(1) latch6(0) latch7(1) latch8(0);
   estateX := ;
end
signal
   clock : input;
   scanIO : bidir;
   scanOut : output;
   enable : input;
end
scanChain
   chain1 [scanIO, latch1, latch2, latch3, latch4];
   chain3 [latch1, latch2, latch3, latch4, scanIO];
   chain2 [latch5, latch6, latch7, latch8, scanOut];
end

timeplate scanTiming period 200ns
   clock := input [0ps:D, 50ns:S, 100ns:D];
   enable := input [0ps:S];
   scanIO := input [0ps:S];
   scanIO := output [0ps:X, 50ns:Q];
   scanOut := output [0ps:X, 50ns:Q, 90ns:X];
```

```
end
pattern pat1 (clock, enable, scanIO:I, scanIO:O, scanOut)
  vector(+, scanTiming) :=    [1 1 1 - X];
  scan(+,scanTiming)    :=    [1 1 - - -], input[chain1:state1],
   output[chain3:estate1];
  vector(+, scanTiming) :=    [1 1 1 - X];
  scan(+,scanTiming)    :=    [1 1 - - -], input[chain1:state2],
   output[chain2:estate2];
  vector(+, scanTiming) :=    [1 1 1 - X];
  scan(+,scanTiming)    :=    [1 1 - - -], input[chain1:state3],
   output[chain2:estate3];
  vector(+, scanTiming) :=    [1 1 1 - X];
  scan(+,scanTiming)    :=    [1 1 - - -], input[chain1:state4],
   output[chain2:estate4];
  vector(+, scanTiming) :=    [1 1 1 - X];
  scan(+,scanTiming)    :=    [1 1 - - -], input[chain1:state5],
   output[chain2:estate5];
end
end
```

End Example

A complete example of WGL scan structures is provided on page 2-96 of this chapter.

Note that the order in which signals are defined in a scanchain declaration determines how pattern bits are assigned to a cell, not the order of those in a scangroup. For example, the resulting scan chain bit order for the example below would be `110`, instead of the intended `011`.

```
signal
sin : input;
sout : output;
end

scancell
    c1;c2;c3;
end;

scangroup
   grp1 [c3, c2, c1]; (note cell order)
end

scanstate
```

```
   st1 := grp1(011);
end

scanchain
   chn1 [sin, c1, c2, c3, sout]; (note cell order)
end
```

2.5.3.5 TimePlates

The TimePlates block is used to define the timing component of the waveforms. The TimePlates convey the unique kinds of timing that are present in the overall waveforms.

The syntax of the WGL TimePlate block is:

timeplate <*timeplateName*>
TimePlate
end

A complete BNF syntactical representation of the TimePlates block follows:

```
Timeplates ::= “timeplate” <timeplateName> TimePlate “end”

TimePlate ::= “period” TimeReference [ “timeset”
<tsNumber> ] Channels

TimeReference ::= (Time | <variableName> )

Time ::= <timeValue> Unit

Unit ::= ( “ps” | “ns” | “us” | “ms” | “sec” )

Channels ::= { SignalReference { “,” SignalReference }
“:=” Track }

SignalReference ::= <signalName> [ Range ]

Range ::= “[” <bitNumber> [ “..” <bitNumber> ] “]”

Track ::= [ Direction ] [ “[” FirstEvent { “,” Event } “]”
] “;”

Direction ::= ( “input” | “output” | “bidir” ) [ (
“reference” | “timing” ) ]

FirstEvent ::= “0” Unit “:” <TDSstate> [ “ ’ ” ( “edge”  |
```

```
“window” ) ]

Event ::= TimeReference “:” <TDSstate> [ “ ’ ” ( “edge”  |
“window” ) ]
```

<timeplateName> is an identifier used to reference the TimePlate throughout later portions of the WGL program. An overall timing period is assigned to each TimePlate by the reserved word period. The TimePlate declaration is a definition of the constituent parts of the TimePlate.

<variableName> is the name of a variable that has been previously defined in the ExprSet sub-block of an EquationSheet block. (For more information, see *ExprSet* on page 2-55.)

Each TimePlate is given an overall time period applying to the length of the cycle following the reserved word period. The period can be a numeric value greater than zero, or a variable having been previously defined in the ExprSet sub-block of an EquationSheet block. (For more information, see *ExprSet* on page 2-55.)

NOTE

A variable used in the TimePlates block must have a value that is meaningful when expressed in units of time.

A TimePlate contains a list of signal *channels*. Conceptually, a channel is a container for one or more signal names, each of which is followed by a *track*. Each channel can contain one or more signals, buses, groups, or multiplexed parts. These entities must have been previously declared in the Signals block. Each channel associates the signals with a track. The track itself contains the actual information about the direction, shape, and timing of the waveform. The TDS states that are used to represent the waveform must be consistent with those available for the direction (input or output). (For a list of TDS state characters, see Table 2-7 on page 2-72.) All the signals that share the channel must have a compatible direction.

NOTE

It is important to note that while multiplexed parts are permitted, multiplexed signals or buses (those signals or buses tagged with the mux attribute in the Signals block that receive their timing parameters from multiplexed parts) are not permitted. In effect, timing is defined for the multiplexed parts, which then supply data for the multiplexed signal or bus with which they are associated in the Signals block.

The first event in a track must have a literal time value of 0. Timing supplied by a variable is not legal for the first event. Subsequent events can use either a literal time value or a variable to specify the timing of the event. A variable, if used, must have been previously defined in the ExprSet sub-block of an EquationSheet block. (For more information, see *ExprSet* on page 2-55.)

The reserved word `timeset` lets you define a tester-specific timing set name that is associated with the timing in the TimePlate. The timing set is defined in the WDB that is produced by a WaveBridge run.

The following is an example of a simple TimePlates block:

Start Example

```
timeplate read period 250ns timeset 1
   clock:= input  [0ps:D, 50ns:U, 100ns:D, 150ns:U, 200ns:D,
                  250ns:U];
      in    := input [0ps:D,170ns:U];
      out   := output [0ps:X,180ns:Q'edge, 220ns:X];
   end
```

End Example

A bidirectional signal can occupy one channel if the direction is specified using the reserved word bidir, or two channels if the direction is defined using both of the reserved words input and output. In the first instance, the channel is doing intra-cycle input/output switching; in the second instance, the channel is doing inter-cycle input/output switching. These two can be combined to make a maximum of three channels per signal.

Contained within each track is a comma-separated list of events. Each event consists of a time value defined by time and a TDS state. For input channels, the TDS force logic state characters must be used; for output channels, TDS expect logic state characters must be used; for bidirectional channels, both force and expect TDS state characters may be used. The TDS state character S indicates that the actual state character is to be "substituted" into the waveform at that point. The actual state character comes from the data bit in the corresponding column in a pattern block. In other words, when *Track* contains an S state character, the actual state is derived from the pattern data. The TDS state character P indicates that the state is to be provided from the previous state (from the previously juxtaposed template). The TDS state character C indicates that the state is the complement of the substituted state. See Table 2-7 on page 2-72 for a list of TDS logic state characters.

For output channels, the compare logic states must be used. The TDS state character Q indicates that the state is to be substituted from the data bit from the corresponding column

in a pattern block. The TDS state character R indicates that the state is the complement of the substituted state. The optional reserved words edge or window (default) can follow an output state to indicate edge or window strobing. During a WaveBridge run, the WaveBridge resource allocation attempts to allocate the type of strobe specified by the reserved word. (The example above uses the reserved word edge.)

An example of a typical TimePlates block, including the corresponding signal definitions in the Signals block and the pattern data defined in the Patterns block, follows. (Note the use of multiplexed buses.)

Start Example

```
signal

#=============================================================================
   # FastClock is generated using eight multiplexed components.
   # Databus bus is made up of two separate busses, bus1 and bus2.

#=============================================================================
   FastClock[edge0, edge1, edge2, edge3, edge4, edge5, edge6, edge7]: mux input;
   rd/_wr                      : output;
   Databus[bus1, bus2][0..3]  : mux bidir; # Multiplexed the two four bit
                                          # busses to get a byte-wide bus.
end

timeplate writeTP period 80ns
   edge0: input[0ps:D, 2ns:U, 8ns:D, 10ns:?]; # Clock for data bit bus1[0]
   edge1: input[0ps:?, 10ns:D, 12ns:U, 18ns:D, 20ns:?]; # Clock for data bit
bus1[1]
   edge2: input[0ps:?, 20ns:D, 22ns:U, 28ns:D, 30ns:?]; # Clock for data bit
bus1[2]
   edge3: input[0ps:?, 30ns:D, 32ns:U, 38ns:D, 40ns:?]; # Clock for data bit
bus1[3]
   edge4: input[0ps:?, 40ns:D, 42ns:U, 48ns:D, 50ns:?]; # Clock for data bit
bus2[0]
   edge5: input[0ps:?, 50ns:D, 52ns:U, 58ns:D, 60ns:?]; # Clock for data bit
bus2[1]
   edge6: input[0ps:?, 60ns:D, 62ns:U, 68ns:D, 70ns:?]; # Clock for data bit
bus2[2]
   edge7: input[0ps:?, 70ns:D, 72ns:U, 78ns:D, 80ns:?]; # Clock for data bit
bus2[3]
   rd/_wr: input[0ps:?, 20ns:D, 80ns:?];      # Indicate write cycle

   bus1[0]: input[0ps:D, 5ns:S, 10ns:?];       # Data bit 0
   bus1[1]: input[0ps:?, 10ns:D, 15ns:S, 20ns:?];# Data bit 1
```

```
   bus1[2]: input[0ps:?, 20ns:D, 25ns:S, 30ns:?];# Data bit 2
   bus1[3]: input[0ps:?, 30ns:D, 35ns:S, 40ns:?];# Data bit 3
   bus2[0]: input[0ps:?, 40ns:D, 45ns:S, 50ns:?];    # Data bit 4
   bus2[1]: input[0ps:?, 50ns:D, 55ns:S, 60ns:?];    # Data bit 5
   bus2[2]: input[0ps:?, 60ns:D, 65ns:S, 70ns:?];    # Data bit 6
   bus2[3]: input[0ps:?, 70ns:D, 75ns:S, 80ns:?];    # Data bit 7
end

pattern load1( FastClock, rd/_wr, Databus)
    vector(+, writeTP) := (11111111 1 10101010XXXXXXXX);
end
```

End Example

You can see in the example that the multiplexed parts do not need be defined as contiguous sections of the timing track; gaps in the defined timing for the multiplexed parts are allowed to support the requirements of your particular tester.

The multiplexed parts can occur in any order in the TimePlate block, as can the timing defined in the timing track. For example, the timing for *edge7* and *edge2* could legally be defined as:

```
edge2: input[0ps:?, 70ns:D, 72ns:U, 78ns:D, 80ns:?];
.   .   .
edge7: input[0ps:?, 20ns:D, 22ns:U, 28ns:D, 30ns:?];
```

As you can see, the timing values are in the reverse order of those shown in the example.

The pattern data `(11111111 1 10101010XXXXXXXX)` is mapped to the buses and signals, as described in *Patterns* on page 2-38.

An edge strobe is an instruction to the tester comparator hardware to take an instantaneous sample of the DUT output, and compare it with the expect data. A window strobe tells the tester comparator hardware to verify that the expect data is appearing at the DUT throughout a window of time. If neither reserved word is specified, the event is assumed by the WaveBridge you are using to be a window strobe.

When defining a track, make sure that you assign increasing time values for each event subsequently defined, whether using a constant time value or a variable; the first event of the waveform must always begin at 0pS, and it is unacceptable to define a second event at 20nS and a third event at 15nS. Remember that all event times are relative to the beginning of the cycle.

TimePlates used with scan pattern rows must satisfy certain requirements. Those signals that terminate scan chains referenced from the same pattern row must have sample states; that is, signals that appear at the start of a scan chain must have an S state character, and signals that appear at the end of a scan chain must have a Q state character in their respective waveform shapes. Any other state characters violate these restrictions, generating a TDS WDB Checker Utility error message when you run the TDS WGL In Converter, or a TDS Tester Rules Checker error message when you use the WDB containing the TimePlate as input to a TDS ScanBridge module. Pattern values are available, but not required, for other signals. For more information, see *Patterns* on page 2-38.

The following is an example of a TimePlates block that can be used with scan pattern rows:

Start Example

```
timeplate runSC period 500ns
   SC1_IN := input[0pS:S, 250nS:D];
   SC2_IN := input[0pS:S, 250nS:D];
   SC1_OUT := output[0pS:X, 250nS:Q];
   SC_CLOCK := input[0pS:U, 250nS:D];
   SC_EN := input[0pS:U];
   BUS_D := output[0pS:X];
   ADDR_IN := input[0pS:P];
end
```

End Example

NOTE

In the above example, only signals containing TDS state characters for unresolved states (such as S or Q) are scan signals (signals that terminate scan chains).

The TimePlates block example shows how to encode a protocol that exercises both `chain1` and `chain2` in parallel. (Scan chains were previously defined in the Scan Chain block example, on page 2-29.) A common scan clock `SC_CLOCK` and enable pin `SC_EN` are shared by both chains. Inputs `SC1_IN` and `SC2_IN` are driven during the first half of the cycle, and the output `SC1_OUT` is sampled during the second half. Other input signals not associated with the scan chain, such as `ADDR_IN`, are held at the “previous” value (that is, at the value they held before the scan operation began). Non-scan outputs, such as `BUS_D`, are masked. For more information, see the following *Patterns* section.

You can use variables in the place of literal time values in the TimePlates block. The variables must be previously defined in a default ExprSet sub-block of an EquationSheet block. (For more information, see *ExprSet* on page 2-55.)

Variables can be substituted for the TimePlate period value and any event time. You can intermix literal time values and variables, although the initial event in a time track must occur at 0pS, and it must be expressed as a literal time value.

The following example shows how variables that were defined in an EquationSheet block can be used in a TimePlate block. The use of variables is highlighted in **Bold** typeface:

Start Example

```
timeplate ts1 period write_cycle
   clk := input[0pS:D, 20nS:U, tclk1:D, 90nS:U];
   ale := input[0pS:D, t1:S, t2:D];
   RE := input[0pS:D, 20nS:S, 50nS:D];
   OE := input[0pS:P, 30nS:S];
   strobe := output[0pS:X, t3:Q, 90nS:X];
end
```

End Example

2.5.3.6 Patterns

The Patterns block is used to define rows of data bits. These rows are also called vectors. The vectors defined in the Patterns block are to be modulated through the TimePlate that is associated with each vector. The result of this modulation creates the waveform.

A binary format of the pattern vectors, to be used in place of ASCII pattern data, is supported. This capability allows you to use binary pattern data from a CAE simulation as input to TDS. You cannot mix ASCII pattern vectors and binary pattern data within a Pattern block. However, you can have an ASCII Pattern block and a binary Pattern block within a WGL file. For more information about binary vectors, see *Binary WGL* on page 2-103.

The syntax of the WGL Patterns block is:

```
pattern <patternName> PatternParameters
PatternRows
end
```

A complete BNF syntactical representation of the Patterns block follows:

```
Patterns ::=  “pattern” PattName “(” PatternParameters “)”
PatternRows “end”

PattName ::= ( <patternName> | <patternNameStr> )

PatternParameters ::= PatternParam { “,” PatternParam }

PatternParam ::= SignalReference [ “:” ( “I” | “O” ) ]

SignalReference ::= <signalName> [ Range ]

Range ::= “[” <bitNumber> [ “..” <bitNumber> ] “]”

PatternRows ::= { [ <vectorLabel> “:” ] ( Loop | Repeat |
ScanRow ) }

Loop ::= “loop” [ <loopName> ] <loopCount>
PatternRows “end” [ <loopName> ]

Repeat ::= [ “repeat” <repeatCount> ] ( Vector | Call |
Offset )

Vector ::= “vector” Address “:=” PatternExpression [
TimeComment ] “;”

Address ::= “(” AddressElement { “,” AddressElement } “)”

AddressElement ::= ( “+” | <cycleNumber> | [ Time] |
<timeplateName> )

Time ::= <timeValue> Unit

Unit ::= ( “ps” | “ns” | “us” | “ms” | “sec” )

PatternExpression ::= “[” {  ( <stateString> |
<patternIdentifier> ) } “]”

TimeComment ::= “(” Time “)”

Call ::= “call” <subroutineName> “( )” “;”

Offset ::= “skip” Time “;”

ScanRow ::= “scan” Address “:=” ScanRowElement { “,”
ScanRowElement } “;”
```

```
ScanRowElement ::= (PatternExpression | ScanRun)

ScanRun ::= ScanDir “[” <chainName> “:” <stateName> “]”

ScanDir ::= ( “input” | “output” | “feedback” )
```

Multiple Pattern blocks are allowed in WGL and can be used to partition a test program into pattern bursts when the WDB is processed by a WaveBridge.

<patternName> is a user-defined name such as `Group_ALL`. <patternNameStr> is a user-defined name such as “`Group+two`”. (String notation allows the use of characters not otherwise permitted.) The <patternName> and <patternNameStr> user-defined names are stored in the WDB.

The PatternExpression defined for each identifier must contain legal pattern <stateString>s. The number of bits in the PatternExpression must be the same as the number of bits in the corresponding signal, bus, group, or multiplexed signal or bus that is associated with it.

PatternParameters is a parentheses-enclosed list of signal names that have already been defined in the Signals block. The PatternParameters are used to map signals, buses, groups, and multiplexed signals or buses (defined in the Signals block) to columns in the PatternExpressions. If multiplexing is used for signals or buses, the pattern bits are combined under the control of the associated radix, in exactly the same manner that the pattern bits are controlled for non-multiplexed buses. For multiplexed parts, the binding order of the pattern bits is left-to-right as specified in the multiplexed signal definition in the Signals block. Each PatternParam in the parameter list corresponds in order of occurrence to columns of data in each vector statement. See the TimePlate example on page 2-34.

PatternRows are definitions of rows of data bits used to supply data to waveforms when modulated through a TimePlate, as defined in the TimePlate block.

The optional TimeComment provides a mechanism for binding a time to a Vector. It is stored in the database as a comment only. (TDS Output Converters may construct these from simulation output times.)

A Vector consists of an Address and an associated pattern expression. The simplest form of an Address is an integer cycle number. A plus sign (+) can be used as an address to automatically increment the cycle number from the previous row. The starting time of the cycle may also appear in the address. If a <timeplateName> is mentioned in an Address, it must reference an existing TimePlate.

All fields of an Address except the TimePlate designation (+, <cycleNumber>, and Time) are ignored by the WGL In Converter. These fields are provided for compatibility with the WGL Out Converter, which generates the fields for documentation purposes.

The <patternIdentifier> can be used in subroutines, pattern blocks, or scan state vectors as a shorthand for PatternExpression when the radix of the associated signal, bus, group, or scan element is set using the reserved word symbolic. See the Symbolics section in this chapter for more information on how to use the reserved word symbolic.

The following vector declaration uses an integer address (`0`), starting time of the cycle (`0pS`), the TimePlate name with which the vector is associated (`t1`), and the pattern data (`[ 1 ZZZZZZZZ ]`).

```
vector(0, 0pS, t1) := [ 1 ZZZZZZZZ ];
```

The vector declaration below uses only automatic increment address (+) and the pattern data (`[1- 1111111100000000 1 -]`).

```
vector(+) := [1- 1111111100000000 1 -];
```

Vectors and subroutine calls may have optional repeat counts. To cause the vector to be used more than once, the reserved word repeat and a repeat count are used.

The following is an example of a simple WGL Patterns block:

Start Example

```
pattern group_ALL (C0,C1,C2,C3,C4,C5,C6,C7,C8)

   vector(0, TimeSet0_0) := [0 0 0 1 1 0 1 1 0 ];
   vector(1, TimeSet1_0) := [1 1 1 0 0 1 1 1 1 ];
   vector(2, TimeSet1_1) := [0 1 1 0 1 1 0 1 0 ];
   vector(3, TimeSet2_0) := [1 1 1 1 1 1 0 1 1 ];
   vector(4, TimeSet3_0) := [0 0 0 0 0 0 1 1 1 ];
   vector(5, TimeSet3_1) := [0 0 0 0 1 0 1 0 0 ];

end
```

End Example

The example below is a WGL Patterns block with a repeat statement that describes a waveform which has a periodic clock for two cycles and an 8-bit data bus that has a value of all Hi-Z for the first cycle, and a value of 0001 1010 for the second cycle. The repeat statement causes third through sixth cycles of the waveform to all have the same value on the data bus.

Start Example

```
signal
   clock      : input;
   data[0..31] : input radix binary;
end

timeplate t1 period 200ns
   clock := input[0ps:D, 100ns:S, 150ns:D];
   data  := input[0ps:Z, 120ns:S] radix binary;
end

pattern load1 (clock, data[8..15])
   vector(0, 0pS, t1) := [ 1 ZZZZZZZZ ] (100ns);
   vector(1, 200nS, t1) := [ 1 00011010 ] (300ns);
   repeat 4 vector(3, 200nS, t1) := [ 1 00011010 ];
end
```

End Example

Bidirectional *patternParameters* always require twice the number of pattern columns to account for input and output directions. If a bidirectional single-bit signal is mentioned as a pattern parameter, two adjacent bits are required (no space between them is allowed). If a bidirectional signal is mentioned with an `:I` or `:O`, this counts as one parameter per occurrence. A space is required between them if both directions are used. Bidirectional buses have all of their input pattern bits mentioned first, followed by the output pattern bits. If an `:I` or `:O` is used on a bidirectional bus, this counts as one pattern parameter, and at least one space is required as a separator.

The number of bits for each pattern parameter must be the same as the width of the signal, bus, group, or multiplexed signal or bus. The number of bits for a bus is the difference between its upper and lower bounds, plus one. The number of bits in a group is the sum of the number of bits of all the group members. The number of bits for a single direction multiplexed bus is the width of the bus times the number of multiplexed parts. The number of bits for a bidirectional multiplexed bus is the width of the bus times the number of the multiplexed parts times two.

The following is an example of a WGL Patterns block with bidirectional bus pattern spacing:

Start Example

```
signal
   foo[0..7] : bidir radix binary;
   fee[0..7] : bidir radix hexadecimal;
   fum[0..7] : bidir radix hexadecimal;
end

pattern load1 (foo,fee,fum:I,fum:O)
   vector(+) := [10101010--------  FF-- F- --];
```

End Example

The `:I` and `:O` can only be used with bidirectional signals, buses, groups, multiplexed signals or buses, or parts of multiplexed signals or buses.

If the number of the pattern bits in the vector statement does not equal the sum of the bits assigned to the buses defined in the Signals block (that is, the bus range, see *Buses* on page 2-17), an error is reported.

The reserved word call invokes a pattern subroutine, as indicated by the <subroutineName>. The rows of the subroutine are treated exactly as if they had been included in-line at the point of the call. Like vectors, calls may have optional repeat counts specified.

The following is an example of a WGL Patterns block with subroutine call `foo`:

Start Example

```
pattern load1 (clock, data[8..15])
   vector(0, 0pS, t1) := [ 1 ZZZZZZZZ ];
   call foo();
   vector(+, t1) := [ 1 00011010 ];
end

subroutine foo()
   vector(t1) := [ 1 00011111 ];
end
```

End Example

The reserved word loop allows a sequence of other vectors, calls, and loops to be repeated a specified number of times. Loops can be nested to any depth. Loops have optional names that have no significance other than as a commentary tag.

The following is an example of a WGL Patterns block with loop `loopName`:

Start Example

```
pattern load1 (clock, data[8..15])
   vector(0, 0pS, t1) := [ 1 ZZZZZZZZ ];
   loop loopName 3
      call foo();
      vector(+, t1) := [ 1 00011010 ];
  end loopName
end
```

End Example

The reserved word skip provides for the declaration of a time period when the waveform state is unspecified. Signal states and event timing are suppressed during the skipped period.

The following is an example of a WGL Patterns block with a skip of 400nS:

Start Example

```
pattern load1 (clock, data[8..15])
   vector(0, 0pS, t1) := [ 1 ZZZZZZZZ ];
   vector(+, t1)      := [ 1 00011010 ];
   skip 400nS;
   vector(+, 0pS, t1) := [ 1 ZZZZZZZZ ];
   vector(+, t1)      := [ 1 00011010 ];
end
```

End Example

Scan pattern rows may appear in pattern blocks freely intermixed with the other row types. Each row represents an arbitrary number of cycles dependent on the lengths of the scan chains that it references.

Note that the scan state defines the values of all scan cells in the device. Only those scan cells on the indicated scan chain(s) are loaded or observed by a particular scan row. Other scan cells not referenced by a chain in the pattern row are not affected by the row. Multiple combinations of chain, state, and direction may appear in each scan row. This provides for parallel scan chains or simultaneous loading and observing of a single chain. It is illegal, however, for a scan row to specify the same chain more than once if the direction of the chain is the same but state values associated with the chain are different.

The following is an example of parallel scan chains:

Start Example

```
pattern pat1 (clock, enable, scanIn, scanOut, scanIn1, scanOut1)
  vector(+, scanTiming) := [1 1 1 1 1 1];
  scan(+,scanTiming)    := [1 1 - - - -],
      input[chain1:state1],
      output[chain2:estate1],
      input[chain11:state3],
      output[chain12:estate3] ;
  vector(+, scanTiming) := [1 1 1 1 1 1];
  scan(+,scanTiming)    := [1 1 - - - -],
      input[chain11:state4],
      output[chain12:estate4],
      input[chain1:state2],
      output[chain2:estate2];
  vector(+, scanTiming) := [1 1 1 1 1 1];
end
```

End Example

It is illegal for a scan chain with no input edge signal to follow the reserved word input. It is illegal for a scan chain with no output edge signal to follow the reserved word output.

The reserved word feedback indicates that the signals appearing on the chain output should be directed back into the chain input while simultaneously comparing against the specified scan state vector. Chains referenced in a feedback clause must have both an input and an output signal. For more information, see *Scan Chain* on page 2-28.

It is important to make certain that signals that terminate scan chains have the proper state character supplied to them, as described on page 2-37 , either from parallel pattern data or from the scan chain associated with the scan run. The following example illustrates a common error made in using scan chains.

Start Example

```
waveform t1
   scancell
      latch1; latch2; latch3; latch4;
      latch5; latch6; latch7; latch8;
   end
   scanstate
      state1 := latch1(0) latch2(0) latch3(0) latch4(0);
      state2 := latch1(0) latch2(0) latch3(0) latch4(1);
      . . .
      estate1 := latch5(1) latch6(1) latch7(1) latch8(0);
```

```
      estate2 := latch5(1) latch6(1) latch7(0) latch8(1);
      estate3 := latch5(1) latch6(1) latch7(0) latch8(0);
   . . .
   end
   signal
   clock : input;
      scanIO : bidir;
      scanOut : output;
      enable : input;
   end
   scanChain
      chain1 [scanIO, latch1, latch2, latch3, latch4];
      chain3 [latch1, latch2, latch3, latch4, scanIO];
      chain2 [latch5, latch6, latch7, latch8, scanOut];
   end
   timeplate scanTiming period 200ns
      clock := input [0ps:D, 50ns:S, 100ns:D];
      enable := input [0ps:S];
      scanIO := input [0ps:S];
      scanIO := output [0ps:X, 50ns:Q];
      scanOut := output [0ps:X, 50ns:Q, 90ns:X];
   end
   pattern pat1 (clock, enable, scanIO:I, scanIO:O, scanOut)
   vector(+, scanTiming) :=[1 1 1 - X];
   scan(+,scanTiming)    :=[1 1 - - -], input[chain1:state1],
                  output[chain3:estate1];
   . . .
   end
end
```

End Example

Edge signals terminating scan chains that are used in the scan runs of a scan pattern row must contain a sample state of the appropriate directionality in the TimePlate referred to by the scan pattern row. Signals that appear at the start of a scan chain (input) must include an S state character, and signals that appear at the end of a scan chain (output) must include a Q state character in their respective waveform shapes. A given scan chain may appear in some, but not all, scan pattern rows in a WDB. A single TimePlate may be used in all scan pattern rows, as long as the state of the edge signal in the scan chain is supplied by the parallel pattern data of the pattern rows that do not use the scan chain in a scan run.

In the parallel scans chain example on page 2-44, the edge signal `scanOut`, which is a part of the scan chain `chain2`, contains a sample state (Q) in the TimePlate `scanTiming`. Problems arise because the associated pattern column contains the

placeholder character (-). In this case, because the edge signal contains the sample state Q, and the Q state requires that a state exists to be sampled, the associated parallel pattern data must supply that state. The example does not, and hence is erroneous.

To repair the error you must either supply a state value in the parallel pattern data, or use `chain2` instead of `chain3` as the terminal chain in the scan run. The remedial sections of the examples below are highlighted in **Bold** type face.

An example of state character supplied in the parallel pattern data is:

Start Example

```
   .  .  .
   pattern pat1 (clock, enable, scanIO:I, scanIO:O, scanOut)
   vector(+, scanTiming) :=   [1 1 1 - X];
   scan(+,scanTiming)    :=   [1 1 - - X], input[chain1:state1],
                              output[chain3:estate1];
   . . .
   end
end
```

End Example

An example of state characters supplied by a scan chain is:

Start Example

```
   .  .  .
   pattern pat1 (clock, enable, scanIO:I, scanIO:O, scanOut)
   vector(+, scanTiming) :=[1 1 1 - X];
   scan(+,scanTiming)    :=[1 1 - - -], input[chain1:state1],
            output[chain3:estate1], output[chain2:estate1];
   . . .
   end
end
```

End Example

A complete example of WGL scan structures is provided on page 2-96 of this chapter.

2.5.3.7 Subroutines

The Subroutines block is used to define pattern sequences that are called repeatedly from a Patterns block.

The syntax of the WGL Subroutines block is:

subroutine <*subroutineName*>
PatternRows
end

A complete BNF syntactical representation of the Subroutines block follows:

```
Subroutines ::=  “subroutine” <subroutineName> “( )”
PatternRows “end”

PatternRows ::= { [ <vectorLabel> “:” ] ( Loop | Repeat |
ScanRow ) }

Loop ::= “loop” [ <loopName> ] <loopCount>
PatternRows “end” [ <loopName> ]

Repeat ::= [ “repeat” <repeatCount> ] ( Vector | Call |
Offset )

Vector ::= “vector” Address “:=” PatternExpression [
TimeComment ] “;”

Address ::= “(” AddressElement { “,” AddressElement } “)”

AddressElement ::= ( “+” | <cycleNumber> [ Unit ] |
<timeplateName> )

Unit ::= ( “ps” | “ns” | “us” | “ms” | “sec” )

PatternExpression ::= “[” {  ( <stateString> |
<patternIdentifier> ) } “]”

TimeComment ::= “(” Time “)”

Time ::= <timeValue> Unit

Call ::= “call” <subroutineName> “( )” “;”

Offset ::= “skip” Time “;”

ScanRow ::= “scan” Address “:=” ScanRowElement { “,”
ScanRowElement } “;”

ScanRowElement ::= (PatternExpression | ScanRun)
```

```
ScanRun ::= ScanDir “[” <chainName> “:” <stateName> “]”

ScanDir ::= ( “input” | “output” | “feedback” )
```

<subroutineName> is a user-defined name, such as `patterns_1`, that is used to define a specific subroutine. PatternRows are definitions of rows of data bits used to supply data to waveforms when modulated through a TimePlate, as defined in the TimePlate block. The interpretation of pattern state information is the same as in the most recently preceding Patterns block; the pattern parameter from the preceding Patterns block also defines the column interpretation in the subroutines that follow.

You define the contents of a subroutine in the Subroutines block, and access the subroutine using the reserved word call. When you call the subroutine you defined in the Subroutines block, WGL jumps to the beginning of the corresponding Subroutines block. On completion of the subroutine, WGL returns to the part of the WGL code immediately after the call statement.

An example of a WGL Subroutines block is:

Start Example

```
subroutine foo()
   vector(t1) := [ 1 00011111 ];
end
```

End Example

The following is an example of a WGL call statement for the subroutine defined in the example above:

Start Example

```
pattern load1 (clock, data[8..15])
   vector(0, 0pS, t1) := [ 1 ZZZZZZZZ ];
   loop loopName 3
      call foo();
      vector(+, t1) := [ 1 00011010 ];
  end loopName
end
```

End Example

2.5.3.8 Symbolics

The Symbolics block is used to associate an identifier with a bit pattern for a specific signal, bus, group, scan cell, scan register, or scan group, making it easier to specify hardware operation codes. Also, if a single-bit signal, bus, or group was defined with a symbolic radix, a Symbolics block must be created that corresponds to the definition.

The syntax of the WGL Symbolics block is:

```
symbolic SigReference [ SymDirection ] Radix
SymbolicAssignment
end
```

A complete BNF syntactical representation of the Symbolics block follows:

```
Symbolics ::=  “symbolic” SignalReference [ SymDirection ]
Radix
SymbolicAssignment “end”

SignalReference ::= <signalName> [ Range ]

Range ::= “[” <bitNumber> [ “..” <bitNumber> ] “]”

SymDirection ::= ( “input” | “output” ) [ ( “reference” |
“timing” ) ]

Radix ::= “radix” ( “binary” | “octal” | “decimal” | “hex”
| “hexadecimal”)

SymbolicAssignment ::= [ <patternIdentifier> ] “:=”
PatternExpression “;”

PatternExpression ::= “[” {  ( <stateString> |
<patternIdentifier> ) } “]”
```

Symbols defined in the Symbolics block can be used in place of the corresponding pattern states in the vectors in the Patterns block.

Each Symbolics block refers to the name of a previously defined signal, bus, group, scan cell, scan register, or scan group. The reserved word input or output must be omitted for scan elements. Signals defined using the reserved word bidir may be associated with two Symbolics blocks. Radix, the radix of the Symbolics block, must also be specified. PatternExpressions within the block are interpreted in the specified radix.

The <patternIdentifier> can be used in subroutines, pattern blocks, or scan state vectors as a shorthand for PatternExpression when the radix of the associated signal, bus, group, or scan element is set using the reserved word symbolic. If a bit pattern is to be entered for which there is no defined identifier, the pattern may be entered in the radix defined in the Symbolics block.

The PatternExpression defined for each identifier must contain legal pattern stateStrings. The number of bits in the PatternExpression must be the same as the number of bits in the corresponding signal, bus, or group that is associated with it. *Scan State* on page 2-26 for more information about stateStrings.

The following is an example of a WGL Symbolics block, and a symbolic radix assignment in pattern block `group_in`:

Start Example

```
signal
   inst [0..7] : input radix symbolic;
   foo : input;
   bar : output;
end
symbolic inst input radix binary
   add   := [00000001];
   sub   := [00000010];
   mul   := [00000011];
   div   := [00000100];
   xor   := [10000000];
   lsl   := [11000000];
   asl   := [11100000];
end
pattern group_in (foo, inst, bar)
   vector(+) := [1 add 0];
   vector(+) := [0 div 1];
   vector(+) := [1 add 1];
end
```

End Example

All the pattern expressions that make up a Symbolics block must be unique. All the identifiers must also be unique. Note that WGL supports partially specified symbolic blocks. It is possible to have identifiers without pattern expressions or pattern expressions without identifiers.

Pattern data that does not match one of the defined symbols may be entered directly in the pattern block in the table radix. If an identifier could also be a legal pattern expression, it is recognized as an identifier. Decimal radix may only be used with buses and groups with 32 or fewer scalar member signals.

The following is an example of Symbolics block with unspecified pattern expressions and identifiers:

Start Example

```
   signal
   data[0..7]: input radix symbolic;
end

symbolic data input radix hex

   GO       := [ 00 ];
   STOP     := [ FF ];
   IDLE     := [ A2 ];
   missing  := [];
            := [ 22 ];
end

pattern sample (data)
   vector(+)   := [ GO   ];
   vector(+)   := [ IDLE ];
   vector(+)   := [   01 ];
   vector(+)   := [   3B ];
   vector(+)   := [ STOP ];
end
```

End Example

2.5.4 Equation-Specific Program Blocks

This section discusses the specific syntax for each of the equation-specific program blocks that have not been discussed previously. The WGL equation-specific program blocks are:

EquationSheet
EquationDefaults

Use the equation-specific program blocks to assign variable timing values for edge placement and current, voltage, and frequency level values for signal strength. You enable equation support by programmatically declaring an EquationSheet block containing at

least one ExprSet sub-block. The ExprSet sub-block contains a list of variables that you create, paired with their assigned constant values, or expressions used to determine the variable value.

You can add more control over which variables are used when you create a test program by declaring the optional EquationDefaults block. The EquationDefaults block specifies which sets of expressions or constant values assigned to variables in the ExprSet sub-blocks are used during subsequent transactions with TDS products that interact with a WDB.

The following example shows the structure of the equation-specific program blocks in a WGL file, and the order in which they are declared. While some of the programming blocks used in the example are optional, the example portrays all possible equation-specific blocks and sub-blocks.

Start Example

```
equationsheet <sheet name>
   exprset <expression set name>
      expression information goes here
   end
   exprset <expression set name>
      expression information goes here
   end
      .
end
equationsheet <sheet name>
   exprset <expression set name>
      expression information goes here
   end
      .
end
equationdefaults
   default information goes here
end
```

End Example

The ExprSet sub-block must be contained within an EquationSheet block and cannot be used as a stand-alone block.

NOTE

The right side of the equation, delimited by the equal sign (=) on one side and the terminating newline character, cannot exceed 247 characters. The total includes white spaces.

In the following manual sections, the equation-specific program blocks are presented in the order that you would be most likely to use them when creating a WDB that includes equations.

2.5.4.1 EquationSheet

EquationSheet blocks allow for the overall organization of variable declarations. An EquationSheet block contains one or more ExprSet sub-blocks.

The ExprSet sub-blocks contain variable declarations, that is, expressions or constant values assigned to variable names. To support equations in your WGL file, the WGL file must contain at least one EquationSheet block with at least one ExprSet sub-block. The number of EquationSheet blocks in a WGL file cannot exceed 100.

EquationSheet blocks and ExprSet sub-blocks must be declared before they are referenced in an EquationDefaults block. For this reason, it is a good idea to declare all EquationSheet blocks before you declare any EquationDefaults blocks. Additionally, the EquationSheets blocks must be declared before the TimePlate block.

The syntax of the WGL EquationSheet block is:

```
equationsheet <equationSheetName>
ExpressionDecl
end
```

A complete BNF syntactical representation of the EquationSheet block follows:

```
EquationSheet ::=   “equationsheet” <equationSheetName>
{ ExpessionDecl } “end”

ExpressionDecl ::= “exprset” <exprSetName> { VariableDecl
} “end”
```

The identifier <equationSheetName> is used to name the specific instance of an Equation Sheet block of the WGL program; it is the unique name of that block.

An <equationSheetName> must be unique within a WGL file and must conform to the naming conventions for identifiers, as described in *Identifiers* on page 2-6. An <equationSheetName> has the same length limitations as signal name for your tester and automatic truncation is performed when EquationSheet names are too long. Any <equationSheetName> that is identical to a WGL reserved word (see the WGL reserved word list on page 2-7) is flagged by the WGL parser as illegal. You can still use an <equationSheetName> that is the same as a WGL reserved word by enclosing the name in double quotation marks (“ ”).

The identifier <exprSetName> refers to an ExprSet sub-block declared within the EquationSheet block of the WGL program. (For details of the WGL constructs contained in the ExprSet sub-block, see *ExprSet* on page 2-55.) The <exprSetName> identifier must conform to the naming conventions for identifiers, as described in *Identifiers* on page 2-6.

The following is an example of two EquationSheet declarations:

Start Example

```
equationsheet AC
   exprset SET1
      tclk1 := tclk + 10nS;
      write_cycle := tclk1*3;
      tclk := 35nS;
      Vcc := 4.5V;
   end
   exprset SET2
      tclk1 := tclk + 20nS;
      write_cycle := tclk1*2;
      tclk := 40nS;
      Vcc := 5.0V;
   end
equationsheet AC_control
   exprset Control_set
      Vih := Vcc-0.5V;
      Vil := Vih-3.0V;
   end
end
```

End Example

2.5.4.2 ExprSet

ExprSet sub-blocks are contained within EquationSheet blocks. They contain precise assignments of expressions and constant values to variables.

The syntax of the WGL ExprSet sub-block is:

```
exprset <exprSetName>
{ VariableDecl }
end
```

A complete BNF syntactical representation of an ExprSet sub-block follows:

```
VariableDecl ::= <variableName> ":="  [ Expression ] [ "["
MinMax "]" ] ";"

Expression ::= Constant | <variableName>
| Expression Operator Expression
| "(" Expression ")" | ("+" | "-") Expression | BuiltInVar
| BuiltInFunc (Expression [, Expression ] )
| ("++" | "--") Expression | Expression ("++" | "--")

BuiltInVar ::= "PI" | "E" | "DEG"

BuiltInFunc ::= "ACOS" | "ASIN" | "ATAN" | "CEIL" | "COS"
| "COSH"
| "EXP" | "FABS" | "FLOOR" | "LOG' | "LOG10"
" SIN" | "SINH" | "SQRT" | "TAN" | "TANH" | "ATAN2"
| "POW"

Operator ::= ( "+" | "-" | "*" | "/" | "^" )

Constant ::= ( <integerValue> | <floatingPointValue> ) [
Scale ] [ EqUnit ]

Scale ::= ( "p" | "n" | "u" | "m" )

EqUnit ::= ( "A" | "V" | "S" | "H" )

MinMax ::=  Constant  | "," Constant  |  Constant ","
Constant
```

An ExprSet sub-block is contained within an EquationSheet block and must have a unique name, the <exprSetName>, within the context of the EquationSheet block that contains it. Multiple ExprSet sub-blocks can be declared within an EquationSheet. Multiple ExprSet sub-blocks allow for the assignment of more than one value or expression to a variable.

The ExprSet sub-block begins with the reserved word exprset followed by the <exprSetName>, which must conform to the naming conventions for identifiers, as

described in *Identifiers* on page 2-6. The body of the ExprSet sub-block contains a list of <variableName>s and the values assigned to them. The sub-block ends with the block terminator, end.

The the number of ExprSet sub-blocks within a EquationSheet block in a WGL file cannot exceed 100. An <exprSetName> must conform to the same length limitations as signal names for your tester; automatic truncation is performed when ExprSet sub-block names are too long.

An <exprSetName> is case sensitive and must begin with an alphabetic character. <exprSetName>s that are identical to WGL reserved words (see the WGL reserved word list on page 2-7) are flagged by the WGL parser as illegal. You can still use a name that is the same as a WGL reserved word by enclosing the name in double quotation marks (“ ”).

While no two <equationSheetName>s can be identical, there can be multiple identical <exprSetName>s and <variableName>s, provided that identical <exprSetName>s are not contained in the same EquationSheet block. Multiple identical <variableName>s are also legal, provided that they are not contained in the same ExprSet sub-block.

The following example shows an illegal usage of <exprSetName>s and <variableName>s.

Start Example

```
# THE FOLLOWING USE OF IDENTICAL EXPRSET NAMES IS ILLEGAL

equationsheet Sheet_1
   exprset worst
      Vcc1:= 4.5V;
      TempDegC1 := 70;
      Textern1 := 10nS;
   end
   exprset best
      Vcc1 := 5.75V;
      TempDegC1 := 0;
      Textern1 := 0nS;
   end
   exprset worst{THIS EXPRSET NAME IS ILLEGAL BECAUSE IT HAS ALREADY BEEN USED
IN THIS EQUATIONSHEET BLOCK}
      Vcc1:= 3.0V;
      TempDegC1 := 90;
      Textern1 := 50nS;
      Vcc1:= 5.0V { THIS VARIABLE NAME IS ILLEGAL BECAUSE IT OCCURS IN THE SAME
EXPRSET SUB-BLOCK AS AN IDENTICALLY NAMED VARIABLE.}
   end
```

```
equationsheet Sheet_2
   exprset worst
      Vcc2:= 4.5V;
      TempDegC2 := 70;
      Textern2 := 10nS;
   end
   exprset best
      Vcc2 := 5.75V;
      TempDegC2 := 0;
      Textern2 := 0nS;
   end
```

End Example

Variables

The <variableName> identifier gives a unique name to a variable that can then be referenced in other parts of the WGL file. The identifier, <variableName>, must conform to the naming conventions for identifiers, as described in *Identifiers* on page 2-6. See *Tester-Specific Program Blocks* on page 2-71 and *TimingSets* on page 2-78 for more information.

Once you assign a value to a <variableName> (or declare the variable) in an ExprSet sub-block, you can reference the <variableName> in the TimePlates block to specify the cycle period or to specify times at which events within TimePlates occur. You can also reference <variableName>s in the TimingSets block to specify a time assignment to a timing generator. Additionally, a <variableName> can be referenced by expressions within ExprSet sub-blocks in EquationSheet blocks other than the one in which the variable was declared.

All variable declarations within an EquationSheet block are unique to that EquationSheet block. A variable of the same name cannot be declared in another EquationSheet block, but it can be declared again in another ExprSet sub-block contained in the same EquationSheet block. In fact, that is the main purpose of multiple ExprSet sub-blocks: to provide a way for you to reassign the value of a variable by naming it in another ExprSet sub-block and giving it a different value.

Any <variableName> declared in any ExprSet sub-block in the WGL file can be referenced in other expressions in the same EquationSheet block or in other EquationSheet blocks.

Forward referencing of variables is allowed. This means that you can reference variables even though those variables are not declared until later in the WGL file.

When you declare a variable in an ExprSet sub-block, the variable name is added to a conceptual list of all the variable names that are declared in all of the ExprSet sub-blocks contained in an EquationSheet block. The set of variable names on the list is actually associated with the EquationSheet block containing the ExprSet sub-block in which the variable was declared. The value assigned to the variable, however, is associated with the ExprSet sub-block.

A conceptual model of the arrangement of equation sheet/expression set data contained within the WDB, follows:

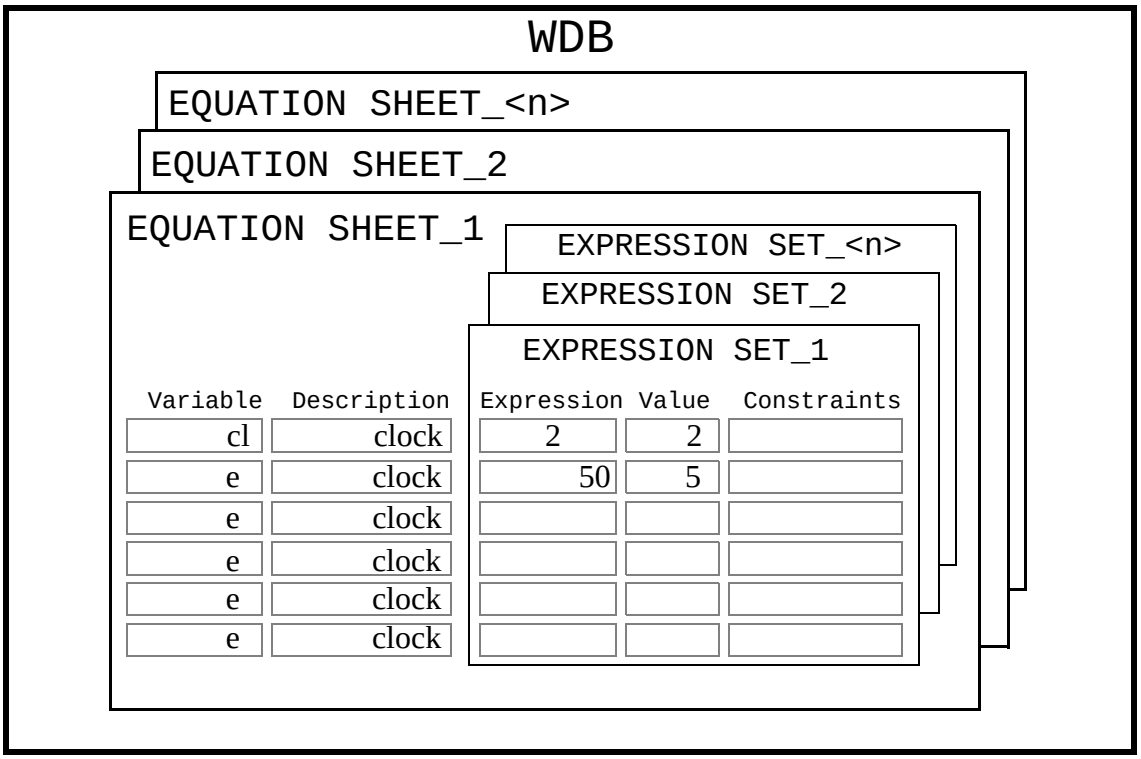

Figure 2-2. Conceptual model of equation sheet data organization.

For example, if you have an EquationSheet block that contains three ExprSet sub-blocks, and in each sub-block you assign values or expressions to two of the variables, the EquationSheet block will have a list of six unique variable names associated with it. On any given ExprSet sub-block, the two variables to which you assigned values have valid,

assigned values; the other four variables associated with the EquationSheet block are unassigned, having no value associated with them.

This becomes important when you use the EquationDefaults block to specify which ExprSet sub-block from an EquationSheet you want to use to assign values to variables. Since all the variables from all of the ExprSet sub-blocks are on the EquationSheet variable name list, you must make certain to explicitly re-declare all variables from all of the ExprSet sub-blocks contained in the EquationSheet block mutually in every other block. Any variable name that is on the list but has no explicit value assigned to it in the active ExprSet sub-block is given an “unassigned” value. While it is syntactically permissible to have unassigned variables in your WGL file, it is a bad practice to do so; if you use any variable that is not explicitly assigned a value in an ExprSet sub-block, and that sub-block is named in the EquationDefaults block, the variable will generate an error message when you use the TDS WGL In Converter to convert your WGL file to a WDB. For more information on how to use the EquationDefaults block, see *EquationDefaults* on page 2-66.

There is no limit to the number of variables within an ExprSet sub-block. A <variableName> must conform to the same length limitations as signal names for your tester; automatic truncation is performed when a <variableName> is too long.

<variableName>s are case sensitive and must begin with an alphabetic character. <variableName>s that are identical to WGL reserved words (see the WGL reserved word list on page 2-7) are flagged by the WGL parser as illegal. You can still use a name that is the same as a WGL reserved word by enclosing the name in double quotation marks (“ ”).

An example of a valid ExprSet sub-block variable is:

```
volt := 5.5V
```

where `volt` is the variable to which a value is assigned.

Constants

A constant can be either an integer (<integerValue>*)* or a floating-point number (<floatingPointValue>*)*.

An example of a valid ExprSet sub-block constant is:

```
t := 3
```

where `3` is the constant value assigned to the variable `t`.

Expressions

An expression is a formula for combining variables, constants, or other expressions in a mathematical way. An expression can be something as simple as a constant value, a reference to a variable, or a combination of constants and variables related to each other with mathematical operators (such as +, -, *, and /).

An example of a valid ExprSet sub-block expression is:

```
clock := 10nS*t
```

where `10nS*t` is the expression whose calculated value is assigned to the variable `clock`.

Operators and Incrementors

The ExprSet sub-block supports a list of standard mathematical operators that you can use when writing an expression.

Table 2-2 is a list of operators, listed in order of decreasing precedence. Operators with the same level of precedence are grouped and separated from operators of differing precedence by bold lines:

Table 2-2. Equation Operators

Operator	*Operation*
*	multiplication
/	division
+	addition
-	subtraction
^	exponent

Built-ins

You can use any of a number of predefined variables or functions in the ExprSet sub-block. The predefined variables (*BuiltInVar*) are listed in the following table:

Table 2-3. Built-in Variables

WGL BuiltInVar	*Value*
E	2.718281828459045523536
DEG	57.2957795130823208768
PI	3.14159265358979323846

The following example shows the use of a built-in variable, **PI**:

```
Start Example
hi_volt := low * PI
End Example
```

where the variable `hi_volt` will receive the value of another variable, `low`, multiple by 3.14159265358979323846.

The following table lists the built-in functions (*BuiltInFunc*) supported in the ExprSet sub-block:

Table 2-4. Built-in Functions

WGL BuiltInFunc	*Performs Operation*
ACOS	arc cosine
ASIN	arc sine
ATAN	arc tangent
CEIL	ceiling (round up to integer)
COS	cosine
COSH	hyperbolic cosine
EXP	exponential e^x
FABS	absolute value
FLOOR	floor (round down to integer)

Table 2-4. Built-in Functions (continued)

WGL BuiltInFunc	*Performs Operation*
LOG	natural logarithm
LOG10	base 10 logarithm
SIN	sine
SINH	hyperbolic sine
SQRT	square root
TAN	tangent
TANH	hyperbolic tangent
ATAN2	arc tangent y/x
POW	x^y

The following example shows the use of a built-in function, **LOG**:

```
Start Example
sim_time := LOG (clock)
End Example
```

where the variable `sim_time` will receive the value of the natural logarithm of another variable, `clock`.

Annotations

Annotations are supported and may be attached to variables in the ExprSet sub-block through the use of curly braces ({ }). Only one annotation is allowed per variable. If a variable is encountered in multiple ExprSet sub-blocks with different annotations, the contents of the annotations are concatenated in the resultant WDB. For identical annotations, only the first instance of the annotation is stored in the WDB, the remaining instances being discarded as redundant.

For further information on how to use WGL annotations, see *Annotations* on page 2-87.

Scaling

You can scale constant values assigned to variables by specifying a value for Scale.

Scale works in concert with EqUnit (see *Units of Measurement* on page 2-64) to permit you to adjust the unit of measurement to suit your needs. The scale prefix must follow the constant to which it applies with no intervening white space and must precede the EqUnit value that it modifies.

The following scale factors represent the available scaling multipliers for constants:

Table 2-5. Scaling prefixes

Suffix	*Multiplier*
p (pico-)	10^{-12}
n (nano-)	10^{-9}
u (micro-)	10^{-6}
m (milli-)	10^{-3}

You can add the scaling prefix to modify the basic units of measurement, as described in *Units of Measurement* on page 2-64.

An ExprSet sub-block using a scaled constant is shown in the following example. In the example, the scaled constant is identified by a WGL annotation:

Start Example

```
exprset AC
   Vol := 2mV; {THIS CONSTANT IS SCALED TO 10-3 }
end
```

End Example

Units of Measurement

Use EqUnit to specify a unit of measurement to be associated with a constant value. You can specify the following units of measurement in the ExprSet sub-block:

Table 2-6. Units of Measurement

WGL Notation	*Unit*
A	ampere
H	hertz

Table 2-6. Units of Measurement

WGL Notation	*Unit*
S	Second
V	volt

You can add a scaling factor to modify the basic units of measurement, as described in *Scaling* on page 2-63.

A WGL fragment showing a EqUnit setting affixed to a constant value assigned to a variable follows:

Start Example

```
exprset timing
   clock := 200nS; { Note the use of the “S” unit value.}
end
```

End Example

Minimum and Maximum Ranges

MinMax lets you specify minimum and maximum values when setting a valid minium value, a valid maximum value, or a valid range (between minium and maximum, including both). This capability is supported through the use of square brackets ([ ]). If you want to specify both minimum and maximum values you must list the minimum value first (`2.2`), followed by a comma, followed by the maximum value (`5.7`), for example, `[2.2,5.7]`.

To specify only the maximum value, provide a comma as a place holder, followed by the maximum value (`7.25`), for example, `[,7.25]`.

Square brackets around an individual value, for example, `[2.5]`, is all that is required to specify a minimum value (`2.5`) only. White space is optional in all cases. Minimum and maximum values can be expressed only using constant values.

A WGL fragment showing a MinMax setting for a variable follows. The variables with MinMax settings are identified by annotations.

Start Example

```
exprset AC_20mhz
   tclk     :=    20nS;
   tempDegC :=    70;
   Vcc      :=    4.5V;
   V1       :=    Vcc/2;
   Vih      :=    Vcc-1 [, 5.5V];         {maximum value specified here }
   Vil      :=    Vih-3 [0.25V];          {minimum value specified here}
   t1       :=    tempDegC/20*1.1nS + tclk;
   write_cycle := tclk*6 [60nS, 600nS];   {min and max specified here}
   cycle_time := 100nS;
end
```

End Example

2.5.4.3 EquationDefaults

The EquationDefaults block establishes which ExprSet sub-blocks are to be used as defaults for calculations. The syntax of the WGL EquationDefaults block is:

```
equationdefaults
DefaultsDecl
end
```

A complete BNF syntactical representation of the EquationDefaults block follows:

```
EquationDefaults ::= “equationdefaults” DefaultsDecl
“end”

DefaultsDecl  ::= <equationSheetName> “:” <exprSetName>
{ “,” <equationSheetName> “:” <exprSetName> } “;”
```

The EquationSheet blocks named by the <equationSheetName> and ExprSet sub-blocks named by the <exprSetName> must be defined before they are referenced in an EquationDefaults block.

All EquationSheet blocks are active in the database but only one ExprSet sub-block per EquationSheet block is active for calculations. EquationSheet blocks and their active ExprSet sub-blocks are explicitly identified through the use of the EquationDefaults block and are specified using a comma-separated list of pairs ending with a semi-colon. These “equation sheet/expression set pairs” are specified by listing the EquationSheet name first, followed by a colon (:), followed by the ExprSet sub-block name. White space is optional.

An example of an EquationDefaults block is shown below with two equation sheet/expression set pairs. In this example, the ExprSet sub-block `SET1` is associated with EquationSheet `AC` and the ExprSet sub-block `Control_20mhz` is associated with the EquationSheet `AC_control`.

Start Example

```
EquationDefaults
   AC:SET1;
   AC_control:Control_20mhz;
end
```

End Example

The EquationDefaults block is not required. If this block is not used, the last ExprSet sub-block declared within each EquationSheet supplies the variable values used for calculations.

If the EquationDefaults block is used, but is not fully specified by explicitly defining an expression set for each equation sheet in the WDB, the variable values assigned in the last ExprSet sub-block declared in the EquationSheet block are used.

If you use more than one EquationDefaults block in your WGL file, the equation sheet/expression set pairs defined in the last EquationDefaults block in the WGL file override any other equations sheet/expression set pairs in that EquationSheet block.

If any EquationSheet block is not specified in the EquationDefaults block(s), the variables in the EquationSheet block obtain their assigned values from the last ExprSet sub-block in that EquationSheet block.

Using more than one EquationDefaults block in your WGL program is not necessary, and sometimes leads to confusion. For example, the following WGL fragment shows what happens when you use two EquationDefaults blocks:

Start Example

```
EquationDefaults
   AC : Set2;
end
EquationDefaults
   timing : eq1;
end
```

End Example

Assume that the only EquationSheet blocks in this WGL file are AC and timing. The first EquationDefaults block sets the default ExprSet sub-block for the EquationSheet block AC to Set2, and the second EquationDefaults block sets the default ExprSet sub-block for the EquationSheet block timing to eq1. However, since every EquationSheet block in a WGL file is active, there is an implicit equation sheet/expression set pair for timing in the first EquationDefaults block, and a similar implicit equation sheet/expression set pair for AC in the second Equationdefaults block. It would be much clearer in this case to define both defaults in a single EquationDefaults block, as shown below:

Start Example

```
EquationDefaults
   AC : Set2;
   timing : eq1;
end
```

End Example

A valid reason for using more than one EquationDefaults block in your WGL program is in the case of incremental test program development. For example, you might want to generate a test program using one set of defaults, then, after evaluating your output, you might add another EquationDefaults block containing different values. You would comment out the previous EquationDefaults block, so that you could keep a record of which defaults you had used during test development. The following example uses such a technique:

Start Example

```
# THE FOLLOWING DEFAULT BLOCK WAS USED FOR TEST 6170_g
#EquationDefaults
#  AC : Set1;
#  timing : eq1;
#end
#
# THE FOLLOWING DEFAULT BLOCK WAS USED FOR TEST 6170_h
#EquationDefaults
#  AC : Set2;
#  timing : eq1;
#end
#
#THE FOLLOWING DEFAULT BLOCK WAS USED FOR TEST 6170_i
```

```
#EquationDefaults
#  AC : Set2;
#  timing : eq2;
#end
#
# THE FOLLOWING DEFAULT BLOCK WAS USED FOR TEST 6170_k
EquationDefaults
   AC : Set1;
   timing : eq2;
end
```

End Example

The above example records the defaults that were used for test 6170_g, 6170_h, and 6170_i. The last EquationDefaults block will specify the defaults for test 6170_k when it is run. Note that the pound signs denoting comment lines do not include the last EquationDefaults block, therefore leaving the last block uncommented and active.

An example of a typical WGL program, using many of the equation support constructs discussed in the previous sections of this chapter, is shown next:

Start Example

```
waveform equation_test_case

signal
   clk   :input;
   ale   :input;
   RE    :input;
   OE    :input;
   dbus[0..3]:output;
end

equationsheet AC_control
   exprset worst
      Vcc := 4.75V;
      TempDegC := 70;
      Textern := 10nS;
   end
   exprset best
      Vcc := 5.5V;
      TempDegC := 0;
      Textern := 0nS;
   end
   exprset typical
```

```
      Vcc := 5V;
      TempDegC := 20;
      Textern := 5nS;
   end
end

equationsheet AC_timing
   exprset eq1
      Vil := Vcc - 3.0;
      Vih := Vcc - 1.0;
      cycle_time := TempDegC/100*1nS + 5V/Vcc*1nS + 100nS;
      tclk1 := 20nS;
      tclk2 := tclk1 + 20nS;
      t1 := TempDegC/100*1nS + 5V/Vcc*1nS + Textern + 10nS;
      t2 := 20nS + t1;
      t3 := t2 + tclk1;
      t4 := cycle_time - 30nS;
      t5 := cycle_time - 10nS;
   end
end

equationdefaults
   AC_timing:eq1;
   AC_control:typical;
end

timeplate ts1 period cycle_time
   clk := input[0pS:D, tclk1:U, tclk2:D, 90nS:U];
   ale := input[0pS:D, t1:S, 80nS:D];
   RE := input[0pS:D, t2:S, t3:D];
   OE := input[0pS:P, 10nS:S];
   dbus[0..3] := output[0pS:X, t4:Q, t5:X];
end

pattern group_ALL (clk, ale, RE, OE, dbus)
   vector(0, ts1) := [- 1 1 1 1011];
   vector(0, ts1) := [- 0 0 0 XXXX];
   vector(0, ts1) := [- 0 0 0 XXXX];
   vector(0, ts1) := [- 1 1 1 1111];
end

end
```

End Example

2.5.5 Tester-Specific Program Blocks

This section discusses the specific syntax for each of the tester-specific program blocks that have not been discussed previously. Use the following tester-specific program blocks to define WDB objects that contain information specific to your tester:

Formats
Registers
Pin Groups
TimeGens
TimingSets

The tester-specific program blocks are presented in the likely order of use when creating a WDB.

2.5.5.1 Formats

The Formats block is used to define tester-specific waveform shapes. A waveform shape describes the general outline of a portion of a waveform. No timing information regarding placement of waveform edges is conveyed in this program block.

The syntax of the WGL Formats block is:

```
format
FormatDecl
end
```

A complete BNF syntactical representation of the Formats block follows:

```
Formats ::=  “format” { FormatDecl } “end”

FormatDecl ::= <formatName> “:” “[” <TDSstate> { “,”
<TDSstate> } “]” “;”
```

FormatDecl is composed of a <formatName>, such as `non_return_to_zero`, followed by a colon (:), followed by one or more of the TDS state characters enclosed in brackets ([ ]). The <formatName> must generally conform to the naming conventions of your tester.

Table 2-7 lists TDS state characters. State characters must be expressed using the proper case, as shown.

Table 2-7. TDS logic states

TDS Logic State Characters	*Meaning*
D	Force logic low
U	Force logic high
N	Force logic unknown
Z	Force logic high impedance
S	Force logic substituted from pattern
C	Force complement of substituted shape
P	Force logic using previous format shape
L	Compare logic low
H	Compare logic high
X	Compare logic unknown (don’t care)
T	Compare logic high impedance
Q	Compare logic substituted from pattern
R	Compare complement of substituted format shape
0	Unknown direction, logic low
1	Unknown direction, logic high
F	Unknown direction, logic high impedance
?	Unknown direction, logic unknown

When the WDB you create in WaveMaker is viewed or edited in WGL format, the force and compare low, high, unknown, and high-impedance TDS logic state characters map to WGL pattern state characters as listed in Table 2-8.

NOTE

The placeholder character (-) is used when no Q, R, S, or C appears in the TimePlate and timing track used for that cycle.

Table 2-8. WGL-pattern-state to TDS-logic-state mapping

WGL Pattern State Characters	*TDS Logic State Characters*	*Meaning*
0	D	Force logic low
1	U	Force logic high
X	N	Force logic unknown
Z	Z	Force logic high impedance
–	not applicable	Placeholder
0	L	Compare logic low
1	H	Compare logic high
X	X	Compare logic unknown (don't care)
Z	T	Compare logic high impedance

There can be multiple instances of *FormatDecl*. Each instance is separated by a semicolon (;).

An example of a WGL Formats block is:

Start Example

```
format
   non_return_to_zero : [S];
   delayed_non_return_to_zero : [P,S];
   return_to_zero : [D,S,D];
   return_to_one :[U,S,U];
   return_to_inhibit : [Z,S,Z];
   surround_by_complement : [C,S,C];
   force_then_compare : [D,S,D,X,Q,X];
end
```

End Example

2.5.5.2 Registers

The Registers block is used for testers that use registers to control the formats applied to particular tester pins.

Format registers are potentially as wide as the number of ATE pins declared in the preceding Signals block. On input, the Registers block pin list may specify any subset of the ATE pins. On output, the WGL Out Converter adds every declared ATE pin to the pin list. Each column of each register may contain a format name declared in a preceding Formats block or a hyphen character (-) indicating unspecified contents. The binding of formats to pins is determined by the correspondence of the position in the register declaration to the position in the pin list. Each register has a name that must be unique among all the registers. Specific register names, as well as format names, and ATE pin names, are tester specific.

The syntax of the Registers block is:

```
register ( PinList )
RegisterDecl
end
```

A complete BNF syntactical representation of the Registers block follows:

```
Registers ::=  “register” “(” PinList “)” { RegisterDecl }
“end”

PinList ::= <atepinName> { “,” <atepinName> }

RegisterDecl ::= <registerName> “:” “[” { FormatSpec } “]”
“;”

FormatSpec ::= ( <formatName> | “-” )
```

Where <atepinName> is an identifier or string previously declared in the **atepin** clause of a Signals block, <registerName> is an identifier or string unique among the register declarations, and <formatName> is an identifier or string previously declared in a Formats block.

An example of a WGL Registers block is:

Start Example

```
register (atepin1, atepin2, atepin3, atepin4)
   ForceReg1 : [ - non_return_to_zero return_to_zero - ];
   ForceReg2 : [ return_to_one - - - ];
   CompareReg1 : [ - - - return_to_inhibit];
end
```

End Example

2.5.5.3 Pin Groups

The Pin Groups block is used to associate ATE pins named in the Signals block with entities called pin groups.

A pin group is a collection of tester pins that share a common format and set of timing generators (or strobes). Pin group assignments are normally made during the resource allocation phase of a WaveBridge run. Pin group names and attributes, however, are defined in the **pingroup** sub-block of the ATE Constraint block of the TCL file. Some testers may have different formatting and timing capabilities associated with pins on pin cards. Those testers organize their pin groups along the lines suggested by the pin cards. See Chapter 3: *Test Control Language* in this guide for more information on how to name pin groups and assign attributes.

A complete BNF syntactical representation of the Pin Groups block follows:

```
PinGroups :=  “pingroup” { PinGroupDecl } “end”

PinGroupDecl := <pinGrpName> “:” “[” [ PinGroupList ] “]”
“;”

PinGroupList :=  <pinElemName> { “,” < pinElemName > }
```

Any pin that is not explicitly assigned to a named pin group defined in the TCL file is assigned automatically to the appropriate default pin group, listed in Table 2-9.

Table 2-9. Default pin groups

Pin Group	*Function*
IPIN	Used as a synonym for all ATE pins that have the direction **input** and that are not explicitly assigned to another pin group.
OPIN	Used as a synonym for all ATE pins that have the direction **output** and that are not explicitly assigned to another pin group.
IOPIN	Used as a synonym for all ATE pins that have the direction **bidir** and that are not explicitly assigned to another pin group.

NOTE

The functions listed in Table 2-9 apply only to automatically defined pin groups; by definition the pins in these groups are not specifically assigned to another group.

Below is an example of a Signals block mapping signals to ATE pins, with a Pin Groups block associating the ATE pins named in the Signals block with pin groups defined in the Pin Groups block.

An example Signals block mapping signals to ATE pins follows:

Start Example

```
signal
   clk  : input  atepin[P1:1 tg[BCLK1, CCLK1]];
   sig1 : input  atepin[P2:2  tg[ACLK1]];
   sig2 : input  atepin[P3:3  tg[ACLK1]];
   sig3 : output atepin[P4:4  tg[WSTRB1]];
   sig4 : output atepin[P5:5  tg[WSTRB1]];
   sig5 : bidir  atepin[P6:6  tg[BCLK2, CCLK2, WSTRB2,
      DREL1, DRET1]];
end

pingroup
   IPIN  : [P1, P2, P3];
   OPIN  : [P4, P5];
   IOPIN : [P6];
   GRP0  : [P1];
   GRP1  : [P2, P3];
   GRP2  : [P4, P5];
   GRP3  : [P6];
end
```

End Example

It is an error if a pin group element name has not been previously defined as an ATE pin of a signal in the Signals block.

2.5.5.4 TimeGens

The TimeGens block is used to define the tester-specific timing generators for a tester. A timing generator is used to specify the time values for edge placement in waveform formats.

The syntax of the WGL TimeGens block is:

timegen
TgDecl
end

A complete BNF syntactical representation of the TimeGens block follows:

```
TimeGens ::=  “timegen” { TgDecl } “end”
```

```
TgDecl ::= <timeGenName> [ “[” <edgeCount> “]” ] “:”
TgType “;”

TgType ::= ( “force” | “compare” | “direction” )
```

TimeGenDecl is composed of a <timeGenName>, such as WSTRB1[2], followed by an optional edge count specifier, followed by a colon (:), followed by one of the following reserved words: force, compare, or direction.

An example of a WGL TimeGens block is:

Start Example

```
timegen
   ACLK1: force;
   BCLK1: force;
   CCLK1: force;
   WSTRB1[2]: compare;
   DRE1[2]: direction;
end
```

End Example

2.5.5.5 TimingSets

The TimingSets block is used to define the tester-specific timing edges required to represent the timing waveforms of the hardware design on a tester. Each timing set has a number and a set of values for the timing generators.

The syntax of the WGL TimingSets block is:

timeset <*tsNumber*>
TgAssign
end

A complete BNF syntactical representation of the TimingSets block follows:

```
TimingSets ::=  “timeset” <tsNumber> { TgAssign } end”

TgAssign ::= <timeGenName> [ “[” <edgeNumber> “]” ] “:=”
TimeReference [ “repeat” <repeatCount> } “;”

TimeReference ::= ( Time | <variableName> )

Time ::= <timeValue> Unit
```

```
Unit ::= ( “ps” | “ns” | “us” | “ms” | “sec” )
```

TgAssign is composed of the following collection of elements:

- An existing timing generator name, which you define in the TimeGens block (see *TimeGens* on page 2-77)
- An optional numeric value for edge number enclosed in brackets ([ ])
- An assignment operator (:=)
- Either a numeric value for time expressed in a supported unit of measurement or a variable that is defined in the ExprSet sub-block of an EquationSheet block (see *ExprSet* on page 2-55.)

NOTE

A variable that is used in the TimingSets block must have a value that is meaningful when expressed in units of time.

An example of a WGL TimingSets block is:

Start Example

```
timeset 1
   ACLK1:= 10ns;
   BCLK1:= 20ns;
   CCLK1:= 80ns;
   WSTRB1[1]:= 30ns;
   WSTRB1[2]:= 80ns;
end

timeset 2
   ACLK1:= 10ns;
   BCLK1:= 50ns;
   CCLK1:= 20ns;
   WSTRB1[1]:= 40ns;
   WSTRB1[2]:= 60ns;
end
```

End Example

You can use variables in the place of literal time values in the TimingSets block. The variables must have been previously defined in an ExprSet sub-block of an EquationSheet block (see *ExprSet* on page 2-55.)

You can also substitute variables for the literal time value associated with a previously defined timing generator (see *TimeGens* on page 2-77.) You can intermix literal time values and variables in the TimeSets block.

The following example shows how variables that were defined in an EquationSheet block can be used in a TimeSets block. The use of variables is highlighted by **Bold** typeface.

Start Example

```
timeset 0 {ts1}
   tgf1 [1] := 0pS;
   tgf1 [2] := 20nS;
   tgc1 [1] := tclk;
   tgc1 [2] := 90nS;
   tgd1 [1] := 0pS;
   tgd1 [2] := 100nS;
   tgf2 [1] := t1;
   tgf2 [2] := t2;
   tgd2 [1] := 0pS;
   tgf3 [1] := 25nS;
   tgf3 [2] := 45nS;
   tgd3 [1] := 0pS;
   tgf4 [1] := 30nS;
   tgd4 [1] := 0pS;
   tgc5 [1] := t3;
   tgc5 [2] := 52nS;
   tgd5 [2] := 0pS;
end
```

End Example

2.6 Additional Features

WGL supports additional features that can provide further control over the data contained in the WDB. These features let you use predefined WGL statements in various places throughout the WGL program, bring data into the current WGL file from other WGL files, and insert comments into the WGL file.

2.6.1 Macros

A WGL macro is a body of valid WGL statements that you can save for later use by giving the body of statements a macro name (<macroName>). The WGL statements become the body of the macro, (<macroBody>). This process defines the contents of the macro. You can recall the contents of the macro that you defined by using a macro invocation. Invoking a macro is essentially calling on your defined macro by name. Neither the macro definition nor the macro invocation becomes part of a WDB.

Using a macro is a two-step process. You must first define the macro with a macro definition. After you have defined the macro, you can invoke it as many times as you want, in any syntactically correct place in the WGL program, with the macro invocation.

2.6.1.1 Macro Definition

The Macro Definition feature follows the same block structure format used by the WGL program blocks. The following rules apply to the macro definition:

- You cannot define other macros within a <macroBody>.
- You cannot invoke a macro recursively; you must not define a macro that invokes itself.
- You can use a parameter in the macro to indicate places in the macro definition where values are to be substituted when the macro is invoked and expanded.
- You can define macros anywhere in the WGL program, but for ease of WGL program maintenance, it is a good idea to define macros at the beginning of the WGL file, right after the beginning program delimiter, **waveform**.
- You can define a macro that invokes another, previously defined macro.

The syntax of the WGL Macro Definition feature is:

```
macro <macroName> ( MacroParameterList )
<macroBody>
endmacro
```

A complete BNF syntactical representation of the Macro Definition feature follows:

```
MacroDefinition ::= “macro” <macroName> [ “(”
MacroParameterList “)” ]
<macroBody> “endmacro”
```

```
MacroParameterList ::= <macroParameter> { “,”
<macroParameter> }
```

In its simplest form, the Macro Definition feature allows you to store a text string under a reference name. (See the example on page 2-94.) The text string may be quite lengthy, cumbersome, and difficult to remember. You can retrieve the text string by calling upon the reference name. This is what happens when you create a macro definition and call up the contents of the <macroBody> using the Macro Invocation feature. Calling up the contents of the macro is often referred to as “expanding” the macro because the contents of the macro are inserted in-line into the code at the place they are called.

A parameter substitution is specified by the ampersand character (@), followed by the <macroParameter> from the MacroParameterList. The value to be substituted into the @<macroParameter> is taken from the MacroParameterList, on the first line of the macro definition. The values for the MacroParameterList are supplied from a list of arguments in the macro invocation. Each Macro Definition can have a maximum of 128 <macroParameter>s.

2.6.1.2 Macro Invocation

The Macro Invocation feature is the counterpart to the Macro Definition feature. To invoke a defined macro, use the name of the defined macro (<macroName>) followed by an optional list of arguments, the contents of which can be substituted into the optional macro parameter list of the Macro Definition feature. If you use the argument list, the macro parameter list must be correspondingly defined in the macro definition.

The syntax of the WGL Macro Invocation feature is:

```
<macroName> [(ArgumentList)]
```

A complete BNF syntactical representation of the Macro Invocation feature follows:

```
MacroInvocation ::= <macroName> [ “(” ArgumentList “)” ]

ArgumentList ::= <identifier> { “,” <identifier> }
```

2.6.1.3 Definition and Invocation without Parameters

Displayed below is an example of a simple macro definition without parameter substitution from a macro parameter list. This example shows four separate macros: `add`, `sub`, `mul`, and `div`.

Start Example

```
macro add
   00011111
endmacro

macro sub
   10101101
endmacro

macro mul
   11100001
endmacro

macro div
   10111000
endmacro
```

End Example

An example of the macro invocation without parameter substitution is:

Start Example

```
pattern load1 (instBus)
   vector(1)  := [add];
   vector(2)  := [sub];
   vector(3)  := [mul];
   vector(4)  := [div];
   vector(5)  := [add];
   vector(6)  := [add];
   vector(7)  := [mul];
   vector(8)  := [sub];
end
```

End Example

An example of the values that exist after macro expansion is:

Start Example

```
pattern load1 (instBus)
   vector(1)  := [00011111];
   vector(2)  := [10101101];
   vector(3)  := [11100001];
   vector(4)  := [10111000];
   vector(5)  := [00011111];
   vector(6)  := [00011111];
   vector(7)  := [11100001];
   vector(8)  := [10101101];
end
```

End Example

2.6.1.4 Definition and Invocation with Parameters

You can invoke a macro and substitute values into the macro parameter list by using the optional argument list with the macro invocation. This gives you added flexibility when using the macro to perform a repetitive task, such as filling vectors with pattern data.

The following is a macro definition with parameter substitution from a macro parameter list. This example uses a macro to fill vectors with pattern data. The <macroParameter> s receives a value from a list of arguments in the macro invocation `diagonal_fill` displayed in the subsequent example.

An example of a macro definition with parameter substitution from the MacroParameterList follows:

Start Example

```
macro diagonal_fill (s)
   vector(+)  : [0000000@s];
   vector(+)  : [000000@s0];
   vector(+)  : [00000@s00];
   vector(+)  : [0000@s000];
   vector(+)  : [000@s0000];
   vector(+)  : [00@s00000];
   vector(+)  : [0@s000000];
   vector(+)  : [@s0000000];
endmacro
```

End Example

An example of a macro invocation with the argument list for substitution into the macro parameter list of the macro definition follows:

Start Example

```
signal
   data[7..0]  : input radix binary;
end

pattern memCheck (data)
   diagonal_fill(0);
   diagonal_fill(1);
   diagonal_fill(Z);
   diagonal_fill(X);
end
```

End Example

An example of the values that exist for the first three macro invocations after expansion of the macro in the previous example is:

Start Example

```
pattern memCheck (data)
   vector(+)  : [00000000];
   vector(+)  : [00000000];
   vector(+)  : [00000000];
   vector(+)  : [00000000];
   vector(+)  : [00000000];
   vector(+)  : [00000000];
   vector(+)  : [00000000];
   vector(+)  : [00000000];

   vector(+)  : [00000001];
   vector(+)  : [00000010];
   vector(+)  : [00000100];
   vector(+)  : [00001000];
   vector(+)  : [00010000];
   vector(+)  : [00100000];
   vector(+)  : [01000000];
   vector(+)  : [10000000];

   vector(+)  : [0000000Z];
```

```
   vector(+)  : [000000Z0];
   vector(+)  : [00000Z00];
   vector(+)  : [0000Z000];
   vector(+)  : [000Z0000];
   vector(+)  : [00Z00000];
   vector(+)  : [0Z000000];
   vector(+)  : [Z0000000];
   .     .     .

end
```

End Example

2.6.2 Include Files

Data that you use repeatedly, for many different WGL programs, can be stored in separate ASCII files and called upon by WGL programs. This lets you create a library of such data files, with each file containing specific types of data in WGL syntax. To include this data into a WGL program, you use the Include file feature of WGL.[1]

Like a WGL macro, Include files are called by an invocation statement, in this case an “include” invocation. Also like WGL macros, when the WGL In Converter is run, Include files are not translated and saved to the database.

You can only invoke a currently existing WGL file that contains syntactically correct WGL statements. The Include file can contain any valid WGL statements.

The syntax of the Include Invocation feature is:

include *<file name>*;

A complete BNF syntactical representation of the Include file feature follows:

```
IncludeInvocation ::= “include” <fileName> “;”
```

An example file named `buses`, that can be invoked in a WGL program to be used as an Include file:

1. Binary pattern files cannot be included in the WGL program via an Include file statement . *Binary WGL* on page 2-103 for information on how to include binary formatted files in a WGL file.

Start Example

```
data [31..0]  : birdir;
addr [31..0]  : bidir;
```

End Example

Use the WGL reserved word `include` to invoke an Include file. When you invoke the Include file, you must specify the file name. You can also use an absolute or relative path when naming the file to be included. The entire invocation is called an include invocation. There cannot be any other WGL syntax, including comments or annotations, on the same line as an include invocation.

The following is an example WGL program with an Include file invocation for a file named `buses.dat`:

Start Example

```
waveform busArbitration
    signal
        include “busses.dat”;
end
```

End Example

2.6.3 Annotations

The Annotations feature allows you to insert comments that are translated for inclusion in the WDB when the WGL In Converter is run. It is possible to view these annotations either in the WGL file or by using WaveMaker’s editors to view the corresponding WDB.

The annotations are enclosed within braces ({ }). Generally speaking, if the annotation occupies the same line as another WGL statement, the annotation is associated with the characteristic described by the WGL statement. If the annotation occupies a line exclusively, with no other WGL statement on the same line, the annotation is associated with the WGL statement immediately following.

The syntax of the Annotations feature is:

```
{ . . . }
```

A complete BNF syntactical representation of the Annotations feature follows:

```
Annotation ::= “{” <any explanatory text> “}”
```

An example of annotations in a WGL program is:

Start Example

```
timeplate read period 300ns
   clock := input [0ps:D, 50ns:U, 100ns:D, 150ns:U, 200ns:D,250ns:U];
   in   := input [0ps:D, 30ns:U]; {in to clock Tsu is 10ns..40ns}
   {Don't expect data on out until at least 20ns after clock
         rising edge}
   out   := output [0ps:X, 70ns:H];
end
```

End Example

WGL associates the annotations with WDB entities (or "objects") in the database. If you add annotations to the WDB using WGL, you must take care that the annotations are placed precisely in the WGL program in areas that support the retention of annotations, or the annotations may be lost or associated with the wrong object. For a complete explanation of how WGL annotations work, see *Using Annotations in WGL* on page 2-99.

2.6.4 Global Mode

The Global Mode feature is used to control attributes of the WDB globally, or in every occurrence of the object with which the attribute is associated. Currently, the only attribute you can control with the Global Mode feature is the pmode attribute, described next.

2.6.4.1 pmode Attribute

The pmode attribute sets the method of interpretation of the initial cycle states that are determined by a previous cycle rather than the current cycle. This feature permits you to tailor the initial state value of waveforms that, by default, derive their initial state value from the previous cycle. Once a P mode has been set in a WDB, you can edit it with the Waveform Editor in WaveMaker, which is described in Section 12.3.2 of the *WaveMaker User's Guide*.

Table 2-10 describes the supported pmode attribute options and the resulting state values. Refer to Table 2-7, on page 2-72, for a complete list of TDS state characters.

NOTE

If a shape contains more than one P state, all P states for that cycle resolve to the same value.

Mux, or muxbus, input or bidir signals (when in input mode) have input tracks starting with "P" and receive their values from the forcing value of the last mux member in the previous cycle.

Table 2-10. P Mode definitions

P Mode Setting	*Replacement*	*Definition*
Previous Force `(last_force)`	a force state (D, U, N, or Z)	If force pattern data for the current cycle (same signal) is Z: P is replaced by Z. Otherwise, P is replaced by the last force state (D, U, N, or Z) in the previous cycle (same signal).
Previous Driving `(last_drive)`	D, U, or Z	If force pattern data for the current cycle (same signal) is Z: P is replaced by Z. Otherwise, P is replaced by the last D or U state in the previous cycle (same signal).
Previous, if Force, else Z `(force_or_z)`	last force state value, else Z	If the last state for the previous cycle (same signal) is D, U, or N: P is replaced by that state. Otherwise, P is replaced by Z.
Advantest `(advantest)`	a force state	If the force pattern data for the current cycle (same signal) is Z: P is replaced by Z. Otherwise, the following conditions are analyzed and the correspoinding value for P is assigned: If the last state in the previous cycle (same signal) is D, U, N, or Z: P is replaced by that state. If the last state in the previous cycle (same signal) is L or T: P is replaced by D. If the last state in the previous cycle (same signal) is H: P is replaced by U. If the last state in the previous cycle (same signal) is X: P is replaced by U, except that X states that follow force states are ignored, unless the entire previous cycle was X.

Table 2-10. P Mode definitions (continued)

P Mode Setting	*Replacement*	*Definition*
IMS `(ims)`	last force state value, else Z	**For scalar (non-multiplexed) signals**, if the last state in the previous cycle (same signal) is D, U, or N: P is replaced by that state; otherwise, P is replaced by Z. **For multiplexed signals**, P substitution is done after multiplexing. Thus, P substitution for a P state on a multiplex member depends on states of other mux members.
Don't care `(dont_care)`	N	P is replaced by N state.

The syntax of the WGL P Mode attribute is:

pmode [*PModeOption***];**

A complete BNF syntactical representation of the P Mode Attribute feature follows:

```
GlobalMode ::=  “pmode” “[” PmodeOption “]” “;”

PmodeOption ::= ( “dont_care” | “last_force” |
“last_drive” | “force_or_z” |
“advantest” | “ims” )
```

An example of a pmode attribute definition is:

Start Example

```
waveform  test.wdb
   pmode[dont_care];
   signal
      a : bidir;
   end
   timeplate io period 500ns
      a := input [0ps:D, 200ns:S, 300ns:D];
      a := output [0ps:P, 250ns:Q, 400ns:T];
   end
end
```

End Example

2.7 Examples

2.7.1 Using WGL Macros and Include Files

You can use WGL macros and include files to simplify your test development. The following examples illustrate the use of include files and macros in a WGL program for a microprocessor. The WGL program, `example_Test_Chip.wgl`, contains only the beginning and ending statements and four include statements.

An example WGL program using Include files is:

Start Example

```
#-------------------------------------------------------------------------
# file: example_Test_Chip.wgl
#-------------------------------------------------------------------------
# An example showing the use of macros and include files, used to generate
# a test for a Test_Chip microprocessor
#
waveform Test_Chip_test1
   include “signals_Test_Chip.wgl”
   include “timing_Test_Chip.wgl”
   include “macros_Test_Chip.wgl”
   include “patterns_1_Test_Chip.wgl”
end
```

End Example

An example WGL Include file containing signal data is:

Start Example

```
#-------------------------------------------------------------------------
# file: signals_Test_Chip.wgl
#-------------------------------------------------------------------------
signalAS        : output;
   AVEC      : input;
   A[0..31]  : output radix hexadecimal;
   BERR      : input;
   BG        : output;
   BGACK     : input;
   BR        : input;
   CDIS      : input;
   CLK       : input;
```

```
   DBEN      : output;
   DS        : output;
   DSACK0    : input;
   DSACK1    : input;
   D[0..31]  : bidir radix hexadecimal;
   ECS       : output;
   FC[0..2]  : input;
   HALT      : bidir;
   IPEND     : output;
   IPL[0..2] : input;
   OCS       : output;
   RESET     : bidir;
   RMC       : output;
   ”R/W”     : output;
   SIZ[0..1] : output;
#
# We divide the data bus up into the instruction and data groups
#
    Inst [D[0..15]] : radix hexadecimal;
    Data [D[16..31]] : radix hexadecimal;
end
```

End Example

An example WGL Include file containing timing data is:

Start Example

```
#-------------------------------------------------------------------------
# file: timing_Test_Chip.wgl
#-------------------------------------------------------------------------
timeplate read period 120nS
   CLK       := input[0pS:U, 20nS:D, 40nS:U, 60nS:D, 80nS:U, 100nS:D];
   A[0..31]  := output[0pS:X, 20nS:Q, 115nS:X];
   FC[0..2]  := input[0pS:P, 20nS:S];
   SIZ[0..1] := output[0pS:X, 20nS:Q, 115nS:X];
   ECS, OCS  := output[0pS:X, 8nS:L, 25nS:X];
   AS        := output[0pS:X, 40nS:L, 100nS:X];
   DS        := output[0pS:X, 40nS:L, 100nS:X];
   ”R/W”     := output[0pS:X, 10nS:H, 115ns:X];
   DSACK0, DSACK1    := input[0pS:U, 70nS:D, 110nS:U];
   Inst,Data := bidir[0pS:X, 80nS:S, 130nS:X];
   DBEN      := output[0pS:X, 50nS:L, 115nS:X];
   BERR, HALT, RESET := input[0pS:U, 80nS:D];
# asynch inputs
```

```
   AVEC, BGACK, BR, CDIS, IPL[0..2]  := input[0pS:N, 45nS:D, 75nS:N];
# asynch outputs
   BG, IPEND, RMC   := output[0pS:X];
end

timeplate write period 120nS
   CLK       := input[0pS:U, 20nS:D, 40nS:U, 60nS:D, 80nS:U, 100nS:D];
   A[0..31]  := output[0pS:X, 20nS:Q, 115nS:X];
   FC[0..2]  := input[0pS:P, 20nS:S];
   SIZ[0..1] := output[0pS:X, 20nS:Q, 115nS:X];
   ECS, OCS  := output[0pS:X, 8nS:L, 25nS:X];
   AS        := output[0pS:X, 40nS:L, 100nS:X];
   DS        := output[0pS:X, 60nS:L, 100nS:X];
   ”R/W”     := output[0pS:X, 10nS:L, 115ns:X];
   DSACK0, DSACK1    := input[0pS:U, 65nS:D, 110nS:U];
   Inst,Data := output[0pS:X, 40nS:Q, 130nS:X];
   DBEN      := output[0pS:X, 25nS:L, 115nS:X];
   BERR, HALT, RESET := input[0pS:U, 80nS:D];
# asynch inputs
   AVEC, BGACK, BR, CDIS, IPL[0..2] := input[0pS:N, 45nS:D, 75nS:N];
# asynch outputs
   BG, IPEND, RMC := output[0pS:X];
end

timeplate idle period 40nS
   CLK       := input[0pS:U, 20nS:D];
   A[0..31]  := output[0pS:X];
   FC[0..2]  := input[0pS:P];
   SIZ[0..1], ECS, OCS, AS, DS, “R/W” := output[0pS:X];
   DSACK0, DSACK1 := input[0pS:U];
   Inst, Data     := output[0pS:X];
   DBEN      := output[0pS:X];
   BERR, HALT, RESET := input[0pS:U];
# asynch inputs
   AVEC, BGACK, BR, CDIS, IPL[0..2] := input[0pS:N];
# asynch outputs
   BG, IPEND, RMC := output[0pS:X];
end

timeplate reset period 40nS
   CLK       := input[0pS:U, 20nS:D];
   A[0..31]  := output[0pS:X];
   FC[0..2]  := input[0pS:N];
   SIZ[0..1], ECS, OCS, AS, DS, “R/W” := output[0pS:X];
```

```
   DSACK0, DSACK1 := input[0pS:N];
   Inst, Data     := output[0pS:X];
   DBEN      := output[0pS:X];
   BERR, HALT := input[0pS:N];
   RESET := input[0pS:D];
# asynch inputs
   AVEC, BGACK, BR, CDIS, IPL[0..2] := input[0pS:N];
# asynch outputs
   BG, IPEND, RMC := output[0pS:X];
end
```

End Example

An example WGL Include file containing macros is:

Start Example

```
#-------------------------------------------------------------------------------
# file: macros_Test_Chip.wgl
#-------------------------------------------------------------------------------
#
# Here are macros defining read and write cycles in terms of only the data
# that changes, in the order that you might want to fill them out.
macro readcycle(instr, addr, data16_32, fc0_2, size)
    vector(+, read) :=
      [- - @addr - - - - - - - - - -
           @instr ----
           @data16_32 ---- -
           @fc0_2 - - - --- - - - - -
           @size ];
endmacro
macro writecycle(instr, addr, data16_32, fc0_2, size)
    vector(+, write) :=
      [- - @addr - - - - - - - - - -
      ---- @instr
      ---- @data16_32 -
           @fc0_2 - - - --- - - - - -
```

```
          @size ];
endmacro
macro idlecycle
    vector(+, idle)  := [- - -------- - - - - - - - - - - ---- ----
      ---- ---- - --- - - - --- - - - - - -- ];
endmacro
macro resetcycle
    vector(+, reset) := [- - -------- - - - - - - - - - - ---- ----
      ---- ---- - --- - - - --- - - - - - -- ];
endmacro
```

End Example

NOTE

The hyphens (-) in the previous example are placeholders for pattern data supplied for the macros readcycle, writecycle, idelcycle, and resetcycle by the WGL Include file shown in the example below.

An example WGL Include file containing pattern data is:

Start Example

```
#-------------------------------------------------------------------------
# file: patterns_1_Test_Chip.wgl
#-------------------------------------------------------------------------
#
# here are the patterns for test1
pattern group_ALL (AS,AVEC,A,BERR,BG,BGACK,BR,CDIS,CLK,DBEN,DS,DSACK0,DSACK1,

Inst:I,Inst:O,Data:I,Data:O,ECS,FC,HALT:I,HALT:O,

IPEND,IPL,OCS,RESET:I,RESET:O,RMC,R/W,SIZ)
repeat 512 resetcycle
readcycle(B61B, B6EE13D6, FCA3, 100, 00)
writecycle(9691, F0201827, A308, 111, 10)
idlecycle
readcycle(4281,F0201827,4314,111,10)
writecycle(30C2,E4394013,4460,011,11)
readcycle(EB3C,86F78F4C,F616,100,11)
writecycle(EE53,9C32C7BA,E9EC,101,00)
```

```
readcycle(BF16,D44C5EB1,DF57,000,11)
writecycle(8D54,E7AB41EC,2927,100,00)
readcycle(7ABC,8316DF68,0744,001,10)
writecycle(69D0,AE31A3A2,0DF0,001,01)
idlecycle
readcycle(7A64,D3B28D8E,A4D6,011,11)
writecycle(4F7E,CFFE12F7,4850,011,11)
readcycle(9A5F,225D2C89,F66B,010,11)
writecycle(619D,7721483A,4862,000,10)
end
```

End Example

2.7.2 WGL and Scan Test Hardware

This example WGL file illustrates a simple scan test using the scan hardware associated with the device shown in Figure 2-3.

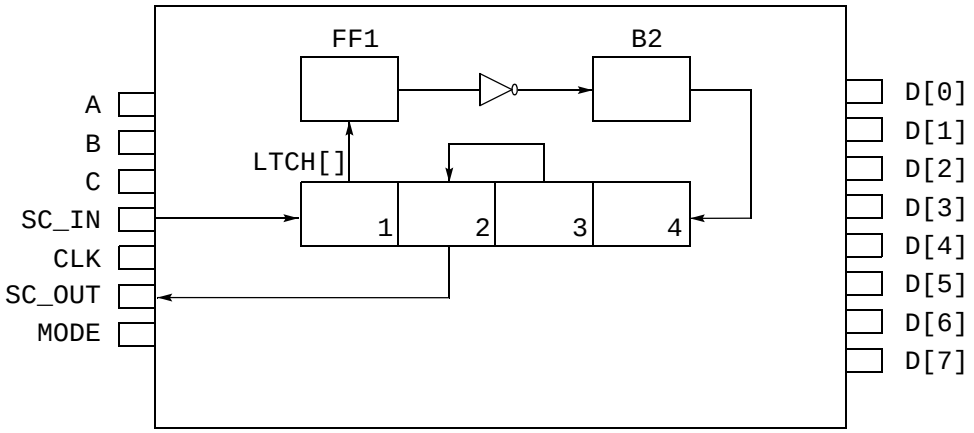

Figure 2-3. Example device with scan hardware

The device in Figure 2-3 has a number of input, output, and bidirectional signals, including `CLK`, `MODE`, `SC_IN`, and `SC_OUT`. Internal cells on the scan chain are declared in the `scanCell` block of the following example WGL files.

A partial example WGL file supporting scan test is:

Start Example

```
waveform scan_example
   signal
      A       : input;
      B       : input;
      C       : output;
      SC_IN   : input;
      SC_OUT  : output;
      CLK     : input;
      MODE    : input;
      D[0..7] : bidir;
   end
   scanCell
      FF1 ;
      B2 ;
      LTCH[1..4] : radix hexadecimal;
   end
   scanchain
      chain1 [SC_IN, LTCH[1], FF1, !, B2, LTCH[4], LTCH[3], LTCH[2], SC_OUT];
   end
   scanState
      stateX := ;
      state1 := FF1(1) B2(0) LTCH(A);
      state2 := FF1(1) B2(1) LTCH(X);
      state3 := ALLSCAN(010101);
   end
   .   .   .
```

End Example

The scan chain shift order is described in the scanchain block above. Note the inverter placed in the chain between cells FF1 and B2. The states that are set in these cells by scan-in operations or tested during scan-out operations are declared in the scanState block.

The test waveform consists of two parallel vectors, followed by a six-cycle scan sequence that shifts a new state into the internal cells while simultaneously sampling the scan chain output and comparing it with another expected state. At the end of the scan operation, two more parallel vectors are applied and the scan is repeated with different input and output states.

A partial example of WGL file with scan entities is:

Start Example

```
   timeplate tp1 period 500nS
      A, B, SC_IN, MODE, D := input[0pS:P, 100nS:S];
      C, SC_OUT, D := output[0pS:X, 300nS:Q, 400nS:X];
      CLK := input[0pS:D, 250nS:U];
   end
   timeplate scanPlate period 500nS
      A, B, SC_IN := input[0pS:P, 100nS:S];
      SC_OUT:= output[0pS:X, 300nS:Q, 400nS:X];
      D := input[0pS:P];
      MODE := input[0pS:P, 100nS:U];
      C, D := output[0pS:X];
      CLK := input[0pS:D, 250nS:U];
   end
   pattern group_ALL (A, B, C, SC_IN, SC_OUT, MODE, D:I, D:O)
      vector(tp1) :=       [1 0 X X X 0 11010000 --------];
      vector(tp1) :=       [1 0 0 X X 0 -------- 11111110];
      scan(scanPlate) :=   [0 1 - - - - -------- --------],
                           input[chain1:state1], output[chain1:stateX];
      vector(tp1) :=       [1 1 X X X 0 00011101 --------];
      vector(tp1) :=       [1 1 0 X X 0 -------- 01010101];
      scan(scanPlate) :=   [0 0 - - - - -------- --------],
                           input[chain1:state3], output[chain1:state2];
      vector(tp1) :=       [0 0 X X X 0 11010011 --------];
      vector(tp1) :=       [1 1 0 X X 0 -------- 01010101];
   end

end
```

End Example

In the example above, two TimePlates are used: `tp1` and `scanPlate`. `tp1` is used on parallel pattern rows. `scanPlate` is used during scan operations. Note that S and Q shapes appear on those tracks associated with scan in and out signals. Signals `A` and `B` use pattern data defined in the scan rows.

The WGL Patterns block illustrates parallel vectors interspersed with scan operations. The `scan` vectors refer to the scan TimePlate and specify which states are scanned in and out using the specified chain. For example, the first scan vector scans in `state1` and simultaneously scans out `stateX`. Since the specified chain is six cells in length, the scan vectors each have a duration of six cycles.

2.7.3 Using Annotations in WGL

In WGL syntax, annotations are “legal” anywhere, as long as they are enclosed in braces ({ }). In this sense, annotations are treated exactly like WGL comments. However, if these annotations are not placed precisely, they may be excluded from the WDB created when you run the TDS WGL In Converter. If you then run the TDS WGL Out Converter to change the WDB back to a WGL file that is editable, you may find that some of the annotations have been lost.

The example below shows a WGL file with annotations added in various locations throughout the file. The WGL file is converted to a WDB using the WGL In Converter, and then converted back to a WGL representation (as shown in the next example) of the same, unmodified WDB. You can see that, depending on their original location in the WGL file, some of the annotations remain unchanged, some have been moved, and some have been lost.

Annotations have been added to the example WGL file named `anno.wgl`. All of the annotations that have been added are syntactically legal, but those that are lost after conversion to a WDB are labeled `{ lost }`.

An example WGL file with annotations, before conversion to WDB is:

Start Example

```
{ lost }
waveform wdb1 { lost }

{ lost }

signal { lost }
   a { a1 } : input;
   b        : input; { b1 }
   c        : {c1} input;
{c2}d        : input;
   e[1..10{e1}] : input;
end  { append to last sig }

scancell
   cell1; { sc1 }
   cell2; { sc2 }
   reg1;  { reg1 }
end  { lost }

scanchain
```

```
   chain1 {c1} [ a, cell1 {c2} ]; { lost }
end { lost }

scanstate
   state1 {moved} := cell1(1) {moved} cell2(1);  {s3}

end { lost }

timeplate tp1 {lost} period {t2} 200ns {t3}
   a{s1},b{lost} := input[0ps:D {lost}, 50ns:S, 100ns:D]; {s4}
   c{s5},d{lost} := input[0ps:D {lost}, 50ns:S, 100ns:D]; {s6}
end

pattern load1 (a)
   vector      (+,tp1)     :=      [1];  {v1}
   vector      (+,tp1)     := {v2} [1];
   vector {v3} (+,tp1)     :=      [1];
   vector      (+,tp1{v4}) :=      [1];
end

end  {lost }
```

End Example

The following example shows the results of converting the original WGL file, `anno.wgl`, to a WDB and then back to a WGL file using the WGL Converters. All annotations labeled `{ lost }` in the original are deleted.

Start Example

```
waveform anno.wdb

signal
    a : input;    { a1 }
    b : input;    { b1 }
    c : input;    {c1}
{c2}
    d : input;
    e [1..10] : input;    {e1}
{ append to last sig }
end

scanCell
    cell1 ;    { sc1 }
    cell2 ;    { sc2 }
```

```
    reg1 ;    { reg1 }
end

scanChain
    chain1 [a, cell1];    {c1}
{c2}
end

scanState
    state1 := ALLSCAN(11X);    {moved}
{moved}
{s3}
end

timeplate tp1 period 200nS    {t2}
{t3}
{s1}
    a, b := input[0pS:D, 50nS:S, 100nS:D];    {s4}
{s5}
    c, d := input[0pS:D, 50nS:S, 100nS:D];    {s6}
end

pattern load1 (a,b,c,d,e)
    vector(0, 0pS, tp1) := [1 - - - ---------- ];    {v1}
    vector(1, 200nS, tp1) := [1 - - - ---------- ];    {v2}
    vector(2, 400nS, tp1) := [1 - - - ---------- ];    {v3}
    vector(3, 600nS, tp1) := [1 - - - ---------- ];    {v4}
end

end
```

End Example

2.7.3.1 Master TimePlate Annotation Example

The SequenceMatch conditioner uses the concept of a master MatchPlate. A master MatchPlate is identified as such via the WDB annotation "`masterMP`". When WGL is the source of the timing WDB used by the SequenceMatch condidtioner, then the annotation is required as in the example below.

Start Example

```
timeplate otherSignals  period 100ns    { masterMP }
   b := input [0ns:P, 20ns:S, 60ns:D];
   b := input [0ns:P, 20ns:S, 60ns:U];
   b := output [0ns:X, 70ns:Q, 85ns:X];
   b := output [0ns:X];
   b := bidir [0ns:P, 50ns:X];
   b := bidir [0ns:X, 50ns:S];
end
```

End Example

Track Match Priority Example

For a given signal the match priority of the tracks within the master MatchPlate is implied by the order in which they appear– the first track that appears has top priority with the priority descending from that point forward. Below is an example depicting track priority:

Start Example

```
timeplate wft  period 100ns    { masterMP }
   b := input [0ns:D, 20ns:S, 60ns:D];     { 1st input priority }
   b := input [0ns:P, 20ns:S, 60ns:D];     { 2nd input priority }
   b := input [0ns:N, 20ns:S, 60ns:N];     { 3rd input priority }
   b := output [0ns:X, 70ns:Q, 85ns:X];    { 1st output priority }
   b := output [0ns:X, 72ns:Q, 83ns:X];    { 2nd output priority }
   b := output [0ns:X, 75ns:Q, 83ns:X];    { 3rd output priority }
   b := output [0ns:X, 75ns:Q, 80ns:X];    { 4th output priority }
end
```

End Example

Match/Spec Track Examples

Match tracks and spec tracks are associated with each via the "`matchTrack`" and "`specTrack`" annotations. When WGL is used the created the timing WDB, the annotations for a matchtrack/spectrack pair, the following syntax must be used:

```
{ matchTrack: <key> }
{ specTrack: <key> }
```

Where <*key*> is the same string of any printable characters not including space or right brace (}). This key is used to indicate that these two tracks are associated with each other. Consider the following example:

Start Example

```
timeplate wft  period 100ns  { masterMP }
   a := [  0ns:N, 40ns:S ];           { matchTrack: 1 }
   a := [  0ns:P, 25ns:S ];           { specTrack: 1 }
   b := [  0ns:P, 20ns:N, 40ns:S ];   { matchTrack: b }
   b := [  0ns:P, 30ns:S ];           { specTrack: b }
   c := [  0ns:P, 20ns:N, 40ns:S ];   { matchTrack: tds }
   c := [  0ns:P, 35ns:S ];           { specTrack: tds }
end
```

End Example

2.8 Binary WGL

In place of ASCII pattern data, a binary format of the pattern vectors is supported within WGL. This capability allows you to use WGL binary pattern data from a CAE simulation[1] as input to TDS.

The binary pattern data in the Pattern section provides a compact data representation for users who are not concerned about readability but who are concerned about file size and TDS run time. WGL binary pattern data has the following advantages over WGL ASCII data:

- A large number of vectors take up less disk space.
- The WGL In Converter reads binary data quicker than ASCII data.
- Scan state vector information is provided directly on a vector row. (In ASCII form, scan state vector information cannot be provided directly on a vector row in the pattern

1. Various CAE simulators output the binary formatted pattern data as specified in this section.

section but must be de-referenced through a scan state name. This results in large amounts of scan data in the upper portion of the WGL file, making it less readable.)

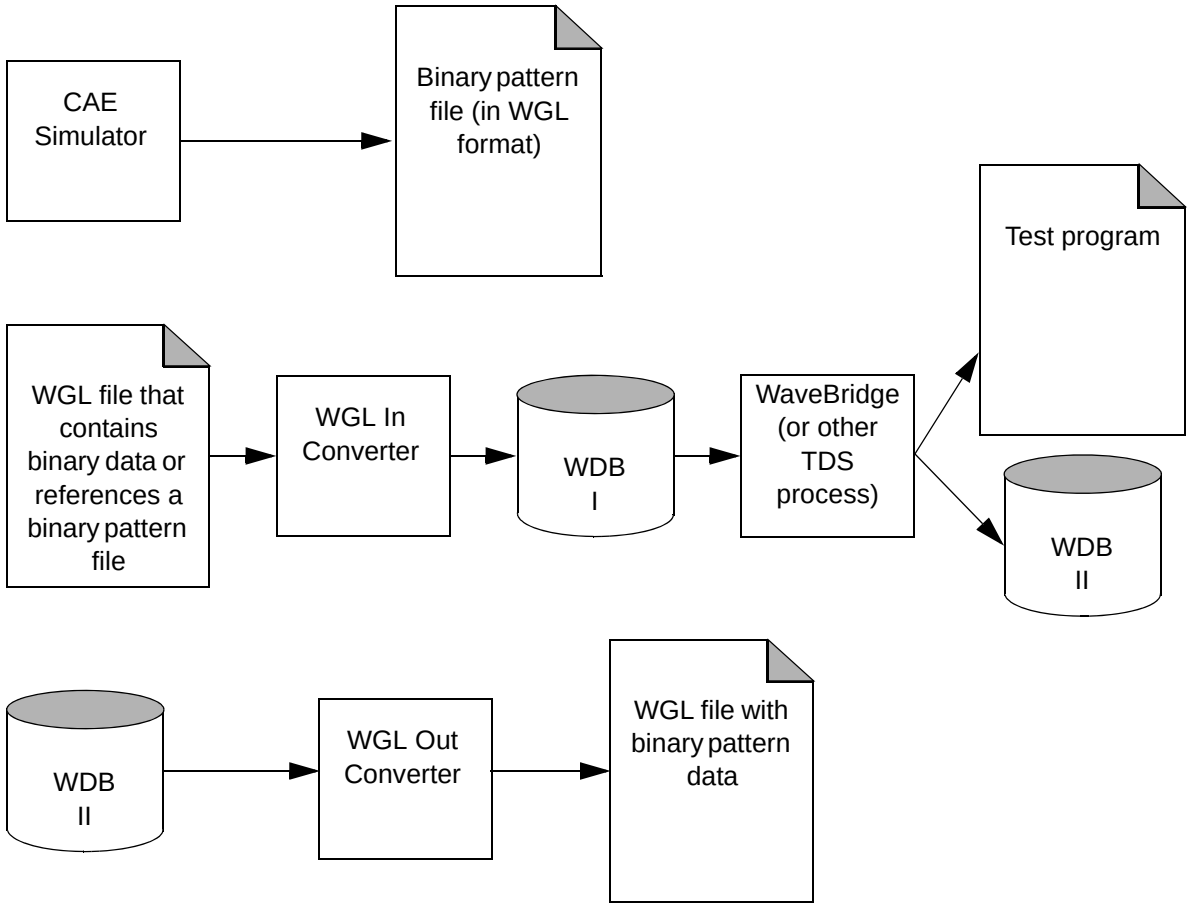

Figure 2-4. Using Binary Pattern Data

2.8.1 WGL Binary Interface

Binary pattern data may be specified in a separate file (preferred) or included in the WGL file.[1] Binary pattern files are included in the WGL program via a *BinaryPattern file* command, not via an Include file statement. (You cannot mix ASCII pattern vectors with binary pattern data.)

1. Do *not* edit a WGL file that has binary pattern data; null pattern bits may be deleted by the editor.

Binary WGL is a subset of ASCII WGL and there is not an exact one-to-one correspondence between ASCII and binary WGL. Some WGL structures are not supported in binary, including symbolic assignments, macros, vector labels, and comments.

The binary pattern data can be viewed with WaveMaker and saved in WDB format. In addition, using the WGL Out Converter, the binary pattern data can be saved in ASCII format within a WGL Patterns block.

2.8.1.1 Including Binary Files

To signify that binary pattern data is supplied in place of the Patterns block within WGL, use the *BinaryPattern* command, followed by the binary data.

```
BinaryPattern; <carriage return>
```

If the binary pattern data is supplied in a file separate from the WGL file, then the *file* parameter must also be specified, followed by the file name where the binary pattern file resides.

```
BinaryPattern file:=binary.data; <carriage return>
```

The following example WGL file shows the *BinaryPattern* command. WGL statements (including the ScanState and Patterns block) that are not used with binary pattern data are shown as comments. (That is, preceded with a #.)

Start Example

```
waveform scan_example

signal
   SC_IN   : input;
   SC_OUT  : output;
   SC_IN2  : input;
   SC_OUT2 : output;
   CLK     : input;
end

scanCell
   FF1 ;
   B2 ;
      C1 ;
      D1 ;
   LTCH[1..4] : radix hexadecimal;
```

```
end

scanchain
   chain1 [SC_IN, LTCH[1], LTCH[4], LTCH[3], LTCH[2], SC_OUT];
   chain2 [SC_IN2, FF1, B2, C1, D1, SC_OUT2];
end

#scanState
#   state1 := chain1(1101) chain2(1001);
#       state2 := chain1(1011) chain2(0001);
#       state3 := chain1(0X00) chain2(1X10);
#       state4 := chain1(0X00) chain2(1XXX);
#       state5 := chain1(0101) chain2(0000);
#       state6 := chain1(XXXX) chain2(XXXX);
#end

timeplate tp1 period 500nS
   SC_IN, SC_IN2 := input[0pS:P, 100nS:S];
   SC_OUT, SC_OUT2 := output[0pS:X, 300nS:Q, 400nS:X];
   CLK := input[0pS:D, 250nS:U];
end

timeplate scanPlate period 500nS
   SC_IN2, SC_IN := input[0pS:P, 100nS:S];
   SC_OUT2, SC_OUT:= output[0pS:X, 300nS:Q, 400nS:X];
   CLK := input[0pS:D, 250nS:U];
end

binarypattern file := testd.tmp;

#pattern group_ALL (CLK, SC_IN, SC_OUT, SC_IN2, SC_OUT2)
#    vector(tp1) := [-  X X X X ];
#    vector(tp1) := [-  X X X X ];
#    scan(scanPlate) := [- - - - - ],
#                        input[chain1:state1], output[chain1:state3],
#                        input[chain2:state1], output[chain2:state3];
#    vector(tp1) := [- X X X X ];
#    vector(tp1) := [- X X X X ];
#    scan(scanPlate) := [- - - - - ],
```

```
#                        input[chain1:state2], output[chain1:state4],
#                        input[chain2:state2], output[chain2:state4];
#    vector(tp1) := [- X X X X ];
#    scan(scanPlate) := [- - - - - ],
#                        input[chain1:state5], output[chain1:state6],
#                        input[chain2:state5], output[chain2:state6];
#    vector(tp1) := [- X X X X ];
#end

end
```

End Example

2.8.2 Binary File Format

The following sections illustrate ASCII WGL formats and equivalent binary WGL formats. If you are reading binary format files (including binary pattern data in a WGL file), you do not need to know this information. However, if you will be writing binary files, you must adhere to the following formats.

The following format conventions are used in this section:

- For readability, characters are shown with the entire string in quotes. In the binary file, the characters are in binary format.
- Numbers are shown in hexadecimal, instead of binary; the 0x preceding a value indicates hexadecimal notation.
- Spaces are added for clarity.
- Braces and brackets are used as described in *WGL Syntax Notation Conventions* on page 2-4.

The binary format is processed using standard I/O routines; the binary file is not parsed. In addition, the binary file is not context sensitive.

2.8.2.1 Definitions

To ensure that the binary format is machine independent, data bits must be written out consistently across machines. The following definitions are required to ensure machine independence.

Table 2-11. Binary Definitions

Item	*Description*
byte	8 bits (unsigned) MSB to LSB
short	16 bits (unsigned) MSB to LSB
long	32 bits (unsigned) MSB to LSB
char	8 bits (unsigned) MSB to LSB
chars	Multiple characters

2.8.2.2 Line Format

All lines in the WGL binary section conform to the following format.

```
byte_count line_type {rest-of-line}
```

Table 2-12. Components of Line Format

Item	*Type*	*Description*
byte_count	short	The length of the line_type and rest-of-line in bytes (excludes byte_count)
line_type	short	Byte which describes the line type (See Table 2-13.)
rest-of-line		Varies depending on the line type (See Table 2-14 through Table 2-32.)

The line length is specified by the byte_count at the beginning of each line. (No specific line termination is provided.)

2.8.2.3 Line Type

The line_type field is an unsigned short which specifies the intent of the line. Table 2-13 shows the mapping.

Table 2-13. Hexadecimal Values for Each Line Type

Hexadecimal	*Line Type*
0x0000	Vector Line
0x0001	Subroutine
0x0002	End Pattern
0x0003	Loop
0x0004	End Loop
0x0005	Subroutine Call
0x0006	Skip
0x0007	Scan Parallel
0x0008	Scan Chain
0x0009	Repeat
0x000a	Pattern Header
0x000b	Annotation
0x000d	Map Key
0x000e	End Subroutine
0x000f	End Binary (ASCII WGL statements follow)
0x00ff	Version Control

2.8.2.4 Line Type Ordering

The binary pattern information must follow the same ordering restrictions required by ASCII WGL (see *Patterns* on page 2-38.) That is, the pattern header is followed by the vectors, which are followed by the subroutine definitions. In addition, the following restriction must be followed:

- The version control line is required to be the first line in the file, if a separate binary file is supplied. Otherwise, the version control line is expected to immediately follow the *BinaryPattern* declaration in the WGL file.
- Binary WGL requires unique end statements for subroutines, loops, and patterns.

2.8.2.5 Line Type Description

The following discussion describes the syntax for each of the line types.

Version Control

The version control line denotes the binary file version. It is required to be the first line in the WGL binary section. (Although not planned, it is possible that future versions of the binary file may have a different format. All future readers, however, will be expected to read earlier versions of binary files.) The format is:

```
byte_count line_type version_number version_extension
```

Table 2-14. Version Control Line Type

Item	*Type*	*Description*
line_type	short	0x00ff
version_number	short	Version 1 is described in this document.
version_extension	short	Extension number; initially 0

Start Example

```
0x0006 0x00ff 0x0001 0x0000
```

End Example

Pattern Header

The WGL Pattern block begins with a pattern header line. This line defines a pattern name, and a list of signals and directions. The binary format would be an encoding of this. The general syntax would be:

```
byte_count line_type name_len name signal_columns
{signal_dir signal_len signal_name bus_flag [begin_range
end_range]}
```

Table 2-15. Pattern Header Line Type

Item	*Type*	*Description*
line_type	short	0x000a
name_len	short	Number of characters in pattern group name
name	chars	Pattern group name
signal_columns	short	Total number of signal columns for the vectors
signal_dir	byte	Column direction where: 0x00 = input column for a bidir signal, 0x01 = output column for a bidir signal, 0x02 = column direction is not required because signal is input or output but not bidirectional
signal_len	short	Number characters in signal name
signal_name	chars	Signal name
bus_flag	byte	Indicates if a signal is a bus: 0x00 = no; 0x01 = yes
begin_range	short	First value in range; this field is read only when bus_flag = 0x01
end_range	short	Second value in range; this field is read only when bus_flag = 0x01

Example WGL:

```
Start Example
pattern burst (sigA:I, sigA:O, BX)
End Example
```

Equivalent binary:

```
Start Example
0x0021 0x000a 0x0005 "burst" 0x0003 0x00 0x0004 "sigA" 0x00 0x01 0x0004 "sigA"
0x00 0x02 0x0002 "BX" 0x00
End Example
```

Example WGL, illustrating multiplexed signals: The Signal block contains a multiplexed parent and four multiplexed children.

Start Example

```
signal
   muxsig1 [sig1_1, sig1_2, sig1_3, sig1_4]: mux input;
end
pattern group_ALL (sig1)
```

End Example

Equivalent binary, illustrating multiplexed signals: signal_columns is set to four, indicating the total number of columns of pattern bit information associated with any vector in the pattern block.

Start Example

```
0x001a 0x000a 0x0009 “group_ALL” 0x0004 0x02 0x0007 “muxsig1” 0x00
```

End Example

Example WGL, illustrating a bus with no range specification: A data bus can be listed in the pattern header without specifying the range and order of the bits. (The range and order specified for a signal within the Signal block is used if none is given on the pattern header.)

Start Example

```
signal
   sig1 : input;
   data[0..7] : input radix binary;

pattern group_ALL (sig1, data)
```

End Example

Equivalent binary, illustrating a bus with no range specification: As specified in the Signal block, the range for this bus is from 0 to 7. The binary format does not require the range to be specified on the pattern header if vector information for the bus adheres to this ordering. signal_columns is set to 8 to indicate the total number of columns of pattern bit information associated with all vectors in the Pattern blocks.

Also, notice that the bus flag is not set to 0x01 in this example. The bus flag is set to 0x01 only when a range is being specified for output on the pattern header.

Start Example

```
0x001f 0x000a 0x0009 “group_ALL” 0x0009 0x02 0x0004 “sig1” 0x00 0x02 0x0004
“data” 0x00
```

End Example

Example WGL, illustrating a bus with a range specification: The bus vector information is found in a different order than as specified in the Signal block. Notice that for the bus addr, the begin_range values are 4, 0, and 5 and the end_range values are 3, 2, and 7.

Start Example

```
signal
   sig1 [sig1_1, sig1_2, sig1_3, sig1_4]: mux input;
   addr[0..7] : input radix binary;
end
pattern group_ALL (sig1, addr[4..3], addr[0..2], addr[5..7])
```

End Example

Equivalent binary, illustrating a bus with a range specification: signal_columns is set to twelve to indicate the total number of columns of pattern bit information associated with all vectors in the pattern block. In each case where the range is specified, the bus flag is set to 0x01.

Start Example

```
0x003B 0x000a 0x0009 “group_ALL” 0x000c 0x02 0x0004 “sig1” 0x00 0x02 0x0004
“addr” 0x01 0x0004 0x0003 0x02 0x0004 “addr” 0x01 0x0000 0x0002 0x02 0x0004
“addr” 0x01 0x0005 0x0007
```

End Example

Individual bus elements may be specified by setting both the `begin_range` and the `end_range` to the bus element number.

End Pattern

The WGL Pattern block terminates with an end statement.

```
byte_count line_type
```

Table 2-16. End Pattern Line Type

Item	*Type*	*Description*
line_type	short	0x0002

Example WGL:

```
Start Example
end
End Example
```

Equivalent binary:

```
Start Example
0x0002 0x0002
End Example
```

Subroutine Header

A WGL Subroutine block begins with a subroutine header line that defines the name of the subroutine. This name is referenced when the subroutine is called.

```
byte_count line_type name
```

Table 2-17. Subroutine Header Line Type

Item	*Type*	*Description*
line_type	short	0x0001
name	chars	Characters in subroutine name

Example WGL:

```
Start Example
subroutine subr0()
End Example
```

Equivalent binary:

```
Start Example
0x0007 0x0001 "subr0"
End Example
```

End Subroutine

Subroutine blocks require an end statement.

```
byte_count line_type
```

Table 2-18. End Subroutine Line Type

Item	*Type*	*Description*
line_type	short	0x000e

Example WGL:

```
Start Example
end
End Example
```

Equivalent binary:

```
Start Example
0x0002 0x000e
End Example
```

NOTE

ASCII WGL has one end statement for both Subroutines and Patterns blocks, while the binary form explicitly provides separate statements for each.

Vector

Vector statements define the parallel, pattern vectors.

```
byte_count line_type tp_name_len tp_name map_key vectors
```

Table 2-19. Vector Line Type

Item	*Type*	*Description*
line_type	short	0x0000
tp_name_len	short	Number of characters in TimePlate name
tp_name	chars	TimePlate name
map_key	byte	Selects the map key
vectors	a	Vector pattern data

a. Defined by map_key (*Map Key* below). 0s are used to pad the data until the last byte is complete.

Map Key

A map key is referenced in all vector and scan lines, defining the mapping between WGL pattern characters and their equivalent binary format. (See Table 2-20 through Table 2-23.) Different map keys can be used for different pattern lines within the same file. For example, use map key 3 (Table 2-23) for all vector and scan pattern row lines and use map key 2 (Table 2-21) for all scan state vector information.

Map key 0 uses three binary bits for every WGL character. It supports all the state characters: 0, 1, Z, and X.

Table 2-20. Map Key 0: Default General Mapping (map_key = 0x00)

Character	*Bit Map*
0	000
1	001
Z	010
X	011
-	111

Map key 1 provides for representation of scan data although it is not restricted to scan data. Mapping a WGL character into one bit of information provides for more compact

data files. This mapping is suggested for scan test cases that do not contain Z or X data, only 0 and 1.

Table 2-21. Map Key 1: Intended for Scan Use (map_key = 0x01)

Character	*Bit Map*
0	0
1	1
Z	Not used
X	Not used
-	Not used

Map key 2 provides for representation of scan data that contains the pattern character X in addition to 0 and 1. A WGL character is mapped into two bits of information.

Table 2-22. Map Key 2: Intended for Scan Use (map_key = 0x02)

Character	*Bit Map*
0	00
1	01
Z	Not used
X	11
-	Not used

Map key 3 provides general mapping for test cases that do not contain Z data. A WGL character is mapped into two bits of information

.

Table 2-23. Map Key 3: General Mapping (map_key = 0x03)

Character	*Bit Map*
0	00
1	01

Table 2-23. Map Key 3: General Mapping (map_key = 0x03) (continued)

Character	*Bit Map*
Z	Not used
X	10
-	11

Example WGL:

```
Start Example
# for the pattern header
# pattern group_ALL (sig1, sig2, sig3, sig4)
# this vector row would be encoded:
vector(tp1) := [0 1 1 0];
End Example
```

Equivalent binary with a map key of 0:

```
Start Example
0x000a 0x0000 0x0003 "tp1" 0x00 000 001 001 000 0000
                           ^^^^ pad bits
End Example
```

Alternate equivalent binary with a map key of 1: A more compact vector representation could have been done using a different map key.

```
Start Example
0x0009 0x0000 0x0003 “tp1” 0x01 0 1 1 0 0000
                     ^^^^ pad bits
End Example
```

Loop

In ASCII WGL, the loop statement supports an optional loop name. In the binary format, the optional loop name is not supported. The binary equivalent of the loop count is expressed as a 32-bit, unsigned long allowing for the maximum size of loop count.

```
byte_count line_type loop_count
```

Table 2-24. Loop Line Type

Item	*Type*	*Description*
line_type	short	0x0003
loop_count	long	Integer loop count

Example WGL:

```
Start Example
Loop 5
End Example
```

Equivalent binary:

```
Start Example
0x0006 0x0003 0x00000005
End Example
```

End Loop

In ASCII WGL, the loop end statement supports an optional loop name. In binary format, the optional loop name is not supported.

```
byte_count line_type
```

Table 2-25. End Loop Line Type

Item	*Type*	*Description*
line_type	short	0x0004

Example WGL:

```
Start Example
end
End Example
```

Equivalent binary:

```
Start Example
0x0002 0x0004
End Example
```

Subroutine Call

Subroutine calls are followed by the subroutine name.

```
byte_count line_type name
```

Table 2-26. Subroutine Call Line Type

Item	*Type*	*Description*
line_type	short	0x0005
name	chars	Subroutine name

Example WGL:

```
Start Example
call subr0();
End Example
```

Equivalent binary:

```
Start Example
 0x0007 0x0005 "subr0"
End Example
```

Repeat

Repeat is used with vectors, loops, or call constructs. Its primary use is on vector lines. This command always indicates that the next command is to be repeated the specified number of times. This line type is always followed by a 32-bit, unsigned integer.

```
byte_count line_type repeat_count
```

Table 2-27. Repeat Line Type

Item	*Type*	*Description*
line_type	short	0x0009
repeat_count	long	Number of times to repeat next statement.

Example WGL:

```
Start Example
repeat 5
End Example
```

Equivalent binary:

```
Start Example
 0x0006 0x0009 0x00000005
End Example
```

Scan Parallel

Two binary line types are required to support a single scan vector as defined in ASCII WGL. In the binary format, the scan parallel line defines the parallel vector states of all the pins in the same format as the vector line. This line does not contain any of the scan chain or scan state vector information. (See *Scan Chain* on page 2-122 for state and chain information.)

```
byte_count line_type tp_name_len tp_name map_key vector
```

Table 2-28. Scan Parallel Line Type

Item	*Type*	*Description*
line_type	short	0x0007
tp_name_len	short	Number of characters in TimePlate
tp_name	chars	TimePlate group name

Table 2-28. Scan Parallel Line Type (continued)

Item	*Type*	*Description*
map_key	byte	Selects the map key
vector	[a]	Parallel vector data

a. Defined by map_key (*Map Key* on page 2-116). 0s are used to pad the data until the last byte is complete.

Example WGL:

```
_____________________________     Start Example     _____________________________
scan(read)   := [0 0 - - ]
_____________________________      End Example      _____________________________
```

Equivalent binary:

```
_____________________________     Start Example     _____________________________
0x000b 0x0007 0x0004 "read" 0x00 000 000 111 111 0000
                            ^^^^ pad bits
_____________________________      End Example      _____________________________
```

Scan Chain

In ASCII WGL, a scan vector references a scan run which consists of a scan chain, the direction of the chain, and a state vector. In ASCII WGL, all state vectors are defined within the ScanState block prior to the pattern block. In addition, the scan state defines the values of all scan cells in the device in ASCII WGL.

The binary format differs from the ASCII representation. In the binary format, the scan chain and scan chain direction are still required. But instead of referencing a specific state vector, the state vector data follow in-line. The in-line scan state information represents only the data which is to be loaded or observed by the specified scan chain.

The scan chain line must follow either a scan parallel line or another scan chain line. The last_chain field identifies the end of the scan chain information.

```
byte_count line_type last_chain chain_dir name_len
```

```
chain_name state_bits map_key scan_states
```

Table 2-29. Scan Chain Line Type

Item	*Type*	*Description*
line_type	short	0x0008
last_chain	byte	0x00 if another chain follows, 0x01 if last in series
chain_dir	byte	Scan chain direction where: 0x00 = input chain, 0x01 = output chain, 0x0f = input/output (feedback) chain
name_len	short	Number of characters in chain name
chain_name	chars	Chain name
state_bits	short	Number of data bits in the scan state vector for this chain. That is, the number of data bits to be loaded or observed for this chain.
map_key	byte	Selects the map key
scan_states	[a]	Scan run pattern data

a. Defined by map_key (*Map Key* on page 2-116). 0s are used to pad the data until the last byte is complete.

Example WGL: In the ASCII WGL file, ssi_1 refers to a scan state vector containing 011100 as data bits for chain ch1 on input and sso_1 refers to a state vector containing 011011 as data bits for chain ch1 on output. These state vectors are previously defined within the ScanState block in the ASCII WGL file.

Start Example

```
scan(read) := [0 0 - -] {this portion of the vector has already been specified
            by the scan parallel binary equivalent }
   input[ch1 : ssi_1],
   output[ch1 : sso_1];
```

End Example

Equivalent binary: The output scan chain and its corresponding scan state are translated into binary format using the map key 1 whereas the input chain uses map key 2.

```
Start Example
0x000e 0x0008 0x00 0x00 0x0003 "ch1" 0x0006 0x02 00 01 01 01 00 00 00 00
                                        ^^ ^^ pad
0x000d 0x0008 0x01 0x01 0x0003 "ch1" 0x0006 0x01 0 1 1 0 1 1 00
                                      ^^ pad bits
End Example
```

See *Example 1* on page 2-126 for an example of scan chains of different lengths.

Skip

The reserved word skip provides for the declaration of a time period when the waveform state is unspecified. In the binary format, the time value, including time units, is provided as a string.

```
byte_count line_type time_string
```

Table 2-30. Skip Line Type

Item	*Type*	*Description*
line_type	short	0x0006
time_string	chars	Time value, including units, for skip duration

Example WGL:

```
Start Example
skip 400ns;
End Example
```

Equivalent binary:

```
Start Example
0x0007 0x0006 “400ns”
End Example
```

Annotations

Annotations are attached to the previous line.

```
byte_count   line_type   annotation
```

Table 2-31. Annotation Line Type

Item	*Type*	*Description*
line_type	short	0x000b
annotation	chars	Annotation string

Example WGL:

```
Start Example
{this is an annotation}
End Example
```

Equivalent binary:

```
Start Example
0x0017 0x000b "this is an annotation"
End Example
```

End Binary

To terminate the binary section of the WGL file, use this command. The parser then expects ASCII WGL to follow. No WGL equivalent exists for this statement.

```
byte_count line_type
```

Table 2-32. End Binary Line Type

Item	*Type*	*Description*
line_type	short	0x000f

Binary format:

```
Start Example
0x0002 0x000f
End Example
```

2.8.3 Examples of ASCII and the Equivalent Binary

Two examples are provided to illustrate the use of binary pattern data. The first example shows the handling of scan vectors, and the second example shows subroutine call, loop, and skip statements. Within each example:

- The original WGL file is shown, followed by
- The WGL file without the pattern block but including a reference to the separate binary file
- An ASCII version of what the binary portion of the file would look like
- Finally, the binary representation of the pattern block

2.8.3.1 Example 1

This example contains two scan chains of different lengths.

Example WGL file:

Start Example

```
waveform patternload
pmode[dont_care];
signal
   sig1   :bidir;
   sig2   :input;
   sig3   :output;
   SC_IN   : input;
   SC_OUT  : output;
   SC_IN2  : input;
   SC_OUT2 : output;
end

scanCell
   a; b; c; d; e; f; g; h; ii; j; k; l; m; n; oo; p; q; r; s; t; u; v; w; x;
   a1; b1; c1; d1; e1; f1; g1; h1; i1; j1; k1; l1; m1; n1; o1;
end

scanChain
   ch1 [SC_IN, a, b, c, d, e, f, g, h, ii, j, k, l, m, n, oo, p, q, r, s, t, u,
v, w, x,  SC_OUT];
```

```
   ch2 [SC_IN2, a1, b1, c1, d1, e1, f1, g1, h1, i1, j1, k1, l1, m1, n1, o1, SC_OUT2];
end

scanState
   TDS_state0 := ch1(110011100001001000110100) ch2(110011100001001);
   TDS_state1 := ch1(11X01X10000100X000110X00);
   TDS_stateX := ;
end

timeplate tp1 period 200ns
   sig1 := bidir[0ps:D, 75ns:S, 95ns:D, 100ns:X, 120ns:Q, 175ns:X];
   sig2 := input[0ps:S];
   sig3 := output[0ps:X, 75ns:Q, 95ns:X];
   SC_IN, SC_IN2:= input[0pS:D];
   SC_OUT, SC_OUT2 := output[0pS:X];
end

timeplate scanPlate period 500nS
   SC_IN2, SC_IN := input[0pS:P, 100nS:S];
   SC_OUT2, SC_OUT:= output[0pS:X, 300nS:Q, 400nS:X];
   sig1 := input[0pS:S];
   sig2 := input[0pS:D];
   sig3 := output[0pS:X];
end

pattern pattern0 (sig1:I, sig1:O, sig2, sig3, SC_IN, SC_OUT, SC_IN2, SC_OUT2)
   vector (0, 0pS, tp1) := [ 0 1 X Z - - - - ];
   scan(scanPlate) := [1 - - - - - - - ],
      input[ch1:TDS_state0], output[ch1:TDS_state1],
      input[ch2:TDS_state0], output[ch2:TDS_stateX];
end
end
```

End Example

WGL file referencing binary pattern file: The above WGL file is changed slightly to include a *binarypattern file* statement that references the binary pattern file named wgl.bin. Notice that the ScanState and the Pattern blocks are no longer included in the WGL file.

Start Example

```
waveform patternload
pmode[dont_care];
signal
   sig1   :bidir;
   sig2   :input;
   sig3   :output;
   SC_IN   : input;
   SC_OUT  : output;
   SC_IN2  : input;
   SC_OUT2 : output;
end

scanCell
   a; b; c; d; e; f; g; h; ii; j; k; l; m; n; oo; p; q; r; s; t; u; v; w; x;
   a1; b1; c1; d1; e1; f1; g1; h1; i1; j1; k1; l1; m1; n1; o1;
end

scanChain
   ch1 [SC_IN, a, b, c, d, e, f, g, h, ii, j, k, l, m, n, oo, p, q, r, s, t, u,
v, w, x,  SC_OUT];
   ch2 [SC_IN2, a1, b1, c1, d1, e1, f1, g1, h1, i1, j1, k1, l1, m1, n1, o1, SC_OUT2];
end

timeplate tp1 period 200ns
   sig1 := bidir[0ps:D, 75ns:S, 95ns:D, 100ns:X, 120ns:Q, 175ns:X];
   sig2 := input[0ps:S];
   sig3 := output[0ps:X, 75ns:Q, 95ns:X];
   SC_IN, SC_IN2:= input[0pS:D];
   SC_OUT, SC_OUT2 := output[0pS:X];
end

timeplate scanPlate period 500nS
```

```
   SC_IN2, SC_IN := input[0pS:P, 100nS:S];
   SC_OUT2, SC_OUT:= output[0pS:X, 300nS:Q, 400nS:X];
   sig1 := input[0pS:S];
   sig2 := input[0pS:D];
   sig3 := output[0pS:X];
end

binarypattern file := wgl.bin;

end
```

End Example

ASCII representation of the binary pattern file wgl.bin: This section is only an illustration of what the binary WGL looks like. It shows the unique line types and their ordering. Scan information follows the scan row and contains a direction, a chain name, and the state information. End statements for the completion of the pattern section and the binary file are required.

Start Example

```
{ Version "1.0" }
pattern pattern0 (sig1:I, sig1:O, sig2, sig3, SC_IN, SC_OUT, SC_IN2, SC_OUT2)
vector(tp1) := [0 1 X Z - - - -];
scan(scanplate) := [1 - - - - - - -]
input[“ch1”: 1 1 0 0 1 1 1 0 0 0 0 1 0 0 1 0 0 0 1 1 0 1 0 0 ],
output[“ch1”: 1 1 X 0 1 X 1 0 0 0 0 1 0 0 X 0 0 0 1 1 0 X 0 0 ],
input[“ch2”: 1 1 0 0 1 1 1 0 0 0 0 1 0 0 1 ],
output[“ch2”:X X X X X X X X X X X X X X X];
end { pattern }
end { binary }
```

End Example

Binary representation: The following is the binary equivalent for the pattern section shown above. For simplicity, signal names, TimePlate names, and scan chain names are shown here as strings instead of in binary, and the 0x notation, indicating hexadecimal, is not included.

In this example, vector information for tp1 and scanPlate is specified using map key 0. The input state vector information for ch1 and ch2 is specified using map key 1. The output state vector information for ch1 and ch2 is specified using map key 2.

Start Example

```
0006 00ff 0001 0000
0056 000a 0008 "pattern0"  0008 00 0004 "sig1" 00 01 0004 "sig1" 00 02 0004 "sig2" 00 02 0004 "sig3" 00 02 0005 “SC_IN” 00 02 0006 “SC_OUT” 00 02 0006 “SC_IN2” 00 02 0007 “SC_OUT2 00 ”
000b 0000 0003 "tp1" 00 05 af ff
0011 0007 0009 "scanPlate" 03 7f ff
000f 0008 0000 0003 “ch1” 00 18 01 ce 12 34
0012 0008 0001 0003 “ch1” 00 18 02 5c 74 01 0c 05 30
000e 0008 0000 0003 “ch2” 00 0f 01 ce 12
0010 0008 0101 0003 “ch2” 00 0f 02 ff ff ff fc
0002 0002
0002 000f
```

End Example

2.8.3.2 Example 2

This example has subroutine, loop, and skip statements, and an annotation.

Example WGL file:

Start Example

```
waveform patternload
pmode[dont_care];
signal
   sig1   :bidir;
   sig2   :input;
   sig3   :output;
end

timeplate tp1 period 200ns
   sig1 := bidir[0ps:D, 75ns:S, 95ns:D, 100ns:X, 120ns:Q, 175ns:X];
   sig2 := input[0ps:S];
   sig3 := output[0ps:X, 75ns:Q, 95ns:X];
end

timeplate read1 period 200ns
   sig1 := bidir[0ps:D, 75ns:S, 95ns:D, 100ns:X, 120ns:Q, 175ns:X];
   sig2 := input[0ps:U];
   sig3 := output[0ps:X];
end

timeplate write period 200ns
```

```
   sig1 := bidir[0ps:X, 75ns:Q, 95ns:X, 100ns:X, 120ns:Q, 175ns:X];
   sig2 := input[0ps:S];
   sig3 := output[0ps:X, 75ns:Q, 95ns:X];
end

pattern pattern0 (sig1:I, sig1:O, sig2, sig3)
   vector (0, tp1) := [ 0 1 X Z ];
   vector (+, read1) := [ 1 1 - - ];  {this is commentA}
   loop 5
   vector (+, write) := [ X X X X ];
   vector (+, read1) := [ 1 0 X -];
   {DXY test}
   end
   call sub0();
end

subroutine sub0()
   skip 400ns;
   vector (+, write) := [ 0 0 0 0 ];
end

end
```

End Example

WGL file referencing binary pattern file:

Start Example

```
waveform patternload
pmode[dont_care];
signal
   sig1   :bidir;
   sig2   :input;
   sig3   :output;
end

timeplate tp1 period 200ns
   sig1 := bidir[0ps:D, 75ns:S, 95ns:D, 100ns:X, 120ns:Q, 175ns:X];
   sig2 := input[0ps:S];
   sig3 := output[0ps:X, 75ns:Q, 95ns:X];
end

timeplate read1 period 200ns
   sig1 := bidir[0ps:D, 75ns:S, 95ns:D, 100ns:X, 120ns:Q, 175ns:X];
```

```
   sig2 := input[0ps:U];
   sig3 := output[0ps:X];
end

timeplate write period 200ns
   sig1 := bidir[0ps:X, 75ns:Q, 95ns:X, 100ns:X, 120ns:Q, 175ns:X];
   sig2 := input[0ps:S];
   sig3 := output[0ps:X, 75ns:Q, 95ns:X];
end

binarypattern file:=wgl.bin;

end
```

End Example

ASCII representation of the binary pattern file wgl.bin: This section is only an illustration of what the binary WGL looks like. It shows the unique line types and their ordering.

Start Example

```
{ Version "1.0" }
pattern pattern0 (sig1:I, sig1:O, sig2, sig3)
vector(tp1) := [0 1 X Z ];
vector(read1) := [1 1 - - ];
{ this is commentA }
loop 5
vector(write) := [X X X X ];
vector(read1) := [- - - - ];
{ DXY test }
end
call sub0();
end

subroutine sub0()
skip 400ns;
vector(write) := [0 0 0 0 ];
end
end
```

End Example

Binary representation: The following is the binary equivalent for the pattern section shown above. For simplicity, signal names, TimePlate names, and subroutine names are

shown here as strings instead of in binary, and the 0x notation, indicating hexadecimal, is not included. The vector information is specified using map key 0.

Start Example

```
0006 00ff 0001 0000
002e 000a 0008 "pattern0"  0004 00 0004 "sig1" 00 01 0004 "sig1" 00 02 0004
"sig2" 00 02 0004 "sig3" 00
000a 0000 0003 "tp1" 00 05 a0
000b 0000 0005 “read1” 03 5f
0012 000b “this is commentA”
0006 0003 0000 0005
000b 0000 0005 “write” 02 ff
000b 0000 0005 “read1” 03 4b
000a 000b “DXY test”
0002 0004
0006 0005 “sub0”
0002 0002
0006 0001 “sub0”
0007 0006 “400ns”
000c 0000 0005 “write” 00 00 00
0002 000e
0002 000f
```

End Example

2.9 Glossary of WGL Terminology

All user-defined identifiers, such as <TDSstate>, used in the WGL BNF representation are found in this glossary. (A *string* is a sequence of characters surrounded by double quotation marks. Embedded double quotation marks and back slashes must be preceded by a back slash.)

any explanatory text

The text of a comment.

atepinName

An identifier or string previously declared as an ATE pin name in the Signals block.

bitNumber

A number specifying a single bit of a multi-bit bus.

If you specify a range (<bitNumber> .. <bitNumber>), the first *bitNumber* defines the most significant bit (MSB); the second *bitNumber* defines the least significant bit (LSB). There is no restriction on which number is larger. (The bits of the register may be labeled in increasing or decreasing order.)

cellName

An identifier or string naming a scan cell. Must be unique among all signals, buses, groups, scan chains, scan registers, and other cells.

chainName

An identifier or string naming a scan chain. Must be unique among all signals, buses, groups, scan cells, scan registers, and other scan chains.

cycleNumber

The numeric cycle number of a pattern vector.

edgeCount

A number indicating the number of edges associated with a timing generator.

edgeNumber

The index of a particular edge of a timing generator.

end-of-line

The end of line indicator.

equationSheetName

An identifier or string naming an EquationSheet block.

exprSetName

An identifier or string naming an ExprSet sub-block.

fileName

The alphanumeric include file name. May be optionally enclosed in double quotation marks (“ ”) or angle brackets (< >).

floatingPointValue

A number containing the digits 0 - 9 and one decimal point (.).

formatName

An identifier or string naming a tester-specific format. Must be unique among all format names.

identifier

The alphanumeric name of a signal, bus, group, TimePlate, format, timegen, pattern, subroutine, et cetera. Identifiers are made up of a sequence of characters that does not include any of the following delimiters: # { } “ ” .. ( ) + , : ; [ ] or white space. Identifiers may not begin with a digit or exactly match any reserved keyword. Names that violate these rules may generally be used provided they are enclosed in double quotation marks and any embedded double quotation mark or back slash characters are preceded with a back slash.

integerValue

A number containing the digits 0 - 9.

loopCount

A number specifying the iteration count of a pattern loop.

loopName

An identifier tagging a pattern loop begin and end statements. These are for documentation purposes only.

macroBody

The text that makes up the body of a macro definition.

macroName

An identifier used in a macro definition or its invocation. (See the example on page 2-94.)

macroParameter

An identifier used as a parameter in a macro definition.

MuxPartName

An identifier associating a particular ATE resource as a source for pattern data to a multiplexed signal or bus. Within a Signals block, reference a <MuxPartName> only once.

patternIdentifier

An identifier assigned to a particular pattern expression in a symbolic block that may be used in pattern and subroutine blocks as an alias for that pattern expression.

patternName

An identifier naming a pattern block that also may identify a tester-specific pattern load (also called a burst). <patternName>s are saved in the database.

patternNameStr

An identifier naming a pattern block that also may identify a tester-specific pattern load (also called a burst). String notation allows the use of characters not otherwise permitted. <patternNameStr>s are saved in the database.

pinElemName

A string identifying an ATE pin.

pinGrpName

A unique identifier for a group.

pinName

An identifier, string, or number identifying the name of a DUT or ATE pin.

pinNumber

An identifier, string, or number identifying the number of a DUT or ATE pin.

registerName

An identifier or string naming a tester-specific format register. Must be unique among all register names.

repeatCount

A number specifying the number of times a pattern vector is to be repeated.

signalName

An identifier or string specifying the name of a signal, group, or bus.

stateName

An identifier or string naming a particular set of logic state values stored in all scan cells. Must be unique among all other state names.

stateString

A sequence of pattern state characters or numbers appearing in a pattern row interpreted according to the width, direction, and radix of the corresponding pattern parameter.

subroutineName

An identifier naming a subroutine declaration or invocation.

timeGenName

An identifier or string naming a tester-specific timing generator.

timeplateName

An identifier naming a TDS timing template. It is defined in a TimePlate block that is referenced in a vector address in a pattern block. Must be unique among all TimePlate names.

timeValue

A number, optionally including a decimal point, specifying a particular time.

TDSstate

A single character that can be any of D, U, N, Z, S, C, P, L, H, X, T, Q, R, 0, 1, F, ?. Case is significant.

tsNumber

A numeric value used to identify individual timing sets.

validityClause

A signal name and state value as used in a Signal Definition file. (For the syntax requirements of the Signal Definition file, see Chapter 4 in the *Getting Started Guide*.) Use this clause within the strobe clause to specify the direction of a signal based on another signal's state value.

variableName

An identifier or string naming an equation variable.

vectorLabel

An identifier or string ...

waveFormName

An identifier or string naming the waveform program. This name is for documentation purposes only. It is not stored in the WDB database.

Chapter 3
Test Control Language

3.1 Introduction

The Test Control Language (TCL) provides control over WaveBridge module processing. Also, TCL files can control the SequenceMatch and TimePlate Match Conditioners.

Three types of files are written in TCL:

- Tester files (required)

  A Tester file describes tester-specific characteristics such as the range of legal pin numbers, maximum number of pattern rows allowed, and minimum cycle length. Default Tester files are provided with the TDS software, but you can specify a customized Tester file if you want. For more information, see Section 4.10 of the *Getting Started Guide*.

- User TCL files (optional)

  A User TCL file, which is user-defined, specifies setup parameters for a WaveBridge run. It also overrides corresponding TCL parameters that were defined in the Tester file. You can also use a User TCL file to control SequenceMatch or TimePlate Match Conditioner runs. For more information, see Section 4.9.1 of the *Getting Started Guide*.

- Override TCL files (optional)

  An Override TCL file, which is also user-defined, overrides corresponding TCL parameters that were defined in the Tester file. For more information, see Section 4.9.2 of the *Getting Started Guide*.

3.2 When to Use TCL

In general, you can use TCL files to more precisely control the execution of WaveBridge modules. Use TCL when you want to:

- Modify existing ATE parameter settings in the Tester file by copying and editing the existing Tester file
- Override existing ATE parameter settings in the Tester file by addressing the same parameters in an optional input TCL file
- Control the name and number of input Standard Events Format (SEF) databases or Waveform DataBases (WDB) to be matched with cycle boundary data or analyzed for tester compatibility
- Control the name and number of the timing templates (TimePlates) carrying the cycle boundary data to be matched against input SEF or WDB databases

NOTE

A TimePlate is a data storage construct contained within a WDB. The TimePlate carries data that determines the shape and timing of each signal within the cycle.

- Control the name and number of the output WDBs
- Control the name and number of the test program pattern loads
- Control the various formatting options for the test programs generated
- Control the tester resources that WaveBridge allocates (formats and timing generators)

3.3 TCL Language Conventions

The TCL language is free-form and case-insignificant except where a user-defined identifier is required. In such cases, the identifier may contain special characters and spaces if the entire identifier is enclosed in double quotation marks (“ ”). Also, any user-defined identifier that is the same as a TCL reserved word must be enclosed in double quotation marks. Multiple white space characters are treated as a single space, and line returns are ignored (except for comments).

3.3.1 TCL Syntax Notation Conventions

In this chapter, TCL syntax is described either in a text paragraph (for example, when discussing how and when to use various TCL constructs) or in a syntactical formalism (as when showing all potential syntactic combinations). Each type of description uses certain notation conventions to make it easier to properly identify the TCL constructs under discussion.

3.3.1.1 Text Descriptions

When describing the elements of TCL syntax in a textual description (rather than in a syntactical formalism), the following notation conventions are used:

- The use of *Emphasis* typeface in text description of a TCL syntactical element indicates higher-level Backus-Naur Formalism (BNF) constructs. Such constructs are expanded to their full definition in the BNF accompanying the reference.

  In the example below, references to *PinList* and *Pin* would appear in the appropriate BNF production as follows:

  ```
  PinList ::= “pins” Pin { “,” Pin }

  Pin ::= <pinNumber> [ “..” <pinNumber> ]
  ```

  Both *PinList* and *Pin* are high-level BNF constructs. These higher-level constructs would be fully defined in the complete BNF production for this TCL block or sub-block.

- The use of **Bold** typeface denotes a TCL parameter or a TCL reserved word or symbol.

  In the example below, taken from a text description, the TCL parameter **VernierRange** is shown in bold typeface.

```
The VernierRange parameter accepts a time value.
```

As in other TDS user documentation, `courier` typeface denotes an actual value assigned to a parameter, reserved word, or excerpt of a display.

3.3.1.2 Backus-Naur Formalism

In describing the full range of possible TCL syntax, the following variation of the Backus-Naur Formalism (BNF) is used:

- Two colons followed by an equivalence sign (::=) denote a high-level syntactic category-to-syntactic rules relationship.

  In the example below, `StartTime` is a high-level syntactic category composed of the low-level TCL reserved word **start**; a TCL reserved typographical symbol (:=); another high-level syntactic category, `TimeSpec`; and a terminal TCL reserved typographical symbol (;).

  ```
  StartTime ::= “start” “:=” TimeSpec “;”
  ```

  An actual TCL fragment using this syntax follows:

  ```
  start := 50ns;
  ```

- Angle brackets (< >) denote a user-defined identifier. The identifier can be an alphanumeric name, an integer, or a floating number.

  In the example below, <repetitionCount> is a user-defined identifier. User-defined identifiers, such as <repetitionCount>, are defined in the TCL programming block in which they occur.

  ```
  Repetition ::= “repetition” “:=” <repetitionCount>
  ```

  An actual TCL fragment using this syntax follows:

  ```
  repetition := 512
  ```

  In the example, <repetitionCount> is `512`.

- Braces ({ }) denote an unspecified repetition of the enclosed syntax.

  In the example below, the high-level syntactic category *MatchSpec* (itself composed of low-level TCL reserved words not shown here) may be repeated as often as required.

```
MatchBody ::= { MatchSpec }
```

An actual TCL fragment using this syntax follows:

```
match

   events
      directory := “sef1”;
      start := begin;
      stop := end;
   end events

   timing
   directory := “timing_source”;
      persistence := 1;
   end timing

   destination
   directory := “dest”;
      start := begin;
      stop := end;
   end destination

end match
```

In the example, the three sub-blocks (delimited by **events** and **end events**, **timing** and **end timing**, and **destination** and **end destination**, respectively) are three separate occurrences of *MatchSpec*.

n Double quotation marks (“ ”) denote the literal use of a TCL reserved word, typographical symbol, or parameter. If double quotation marks are themselves to be used literally, they are enclosed within single quotation marks (‘ ’).

In the example below, the plus sign (+), to be used literally (in this case, as a line continuation character), is enclosed in double quotation marks (“ ”). The user-defined identifier, <string>, is to be enclosed literally in double quotation marks, so the double quotation marks are themselves shown enclosed in single quotation marks.

```
MessageString ::= ‘“’ <string> ‘”’ { “+” ‘“’ <string> ‘”’
}
```

An actual TCL fragment using this syntax follows:

```
"A force edge at time @edge does not"
+"reside at the required resolution of"
```

- Brackets ([ ]) denote optional syntax.

  In the example below, the TCL fragment composed of a TCL substitution sign (<=) and the high level syntactic category *TimeplateName* (itself composed of low-level TCL reserved words not shown here) can be included in the TCL TimePlate definition statement, as required.

  ```
  Timeplate ::= TimeplateName [ "<=" TimeplateName ]
  ```

  An actual TCL fragment using this syntax follows:

  ```
  tp1 <= read
  ```

  In the example, the first occurrence of *TimeplateName* is `tp1`; the optional second occurrence is `read`.

- A vertical bar (|) denotes separate choices of syntax.

  In the example below, either of the two valid values for the TCL reserved word **ignore** (`informative` or `warning`) may be used to suppress the logging of non-fatal TRC errors.

  ```
  IgnoreWhen ::= "ignore" ":=" ( "informative" | "warning" )
  ```

  An actual TCL fragment using this syntax follows:

  ```
  ignore := warning
  ```

  In the example, `warning` has been selected from the list of valid values.

3.3.2 Comments

As in other programming languages, you can add explanatory comments to a TCL file to aid functional clarity. These comments are preceded by the pound sign (#).

Comments can be inserted into any part of a TCL file. To insert a comment into a TCL file, enter #, followed by a text string. All characters on the line, starting with the pound sign and terminating with the carriage return marking the end of the line, are included in the comment.

The syntax of the Comment feature is:

```
# . . . (end-of-line character)
```

A complete BNF syntactical representation of the Comment feature follows:

Comment ::= "#" text characters "*end-of-line character*"

3.3.3 Reserved Words

TCL reserves certain words as its linguistic set. The following case-insensitive list includes all TCL reserved words and symbols. Excluded are TCL ATE constraint parameters, which are listed separately in *ATE Constraints* on page 3-12.

%	comment	loopbegin	signal
“	compress	loopend	signals
”	compression	match	simcomment
(	control	max	simtime
)	cyclenumber	memoryaddress	socket
*	Cycles	message	source
+	cycles	min	start
-	cyclic	ms	State
..	dcspec	netlist	state
/	design	none	stop
:	destination	notgsharing	string
:=	directory	ns	strobeusage
,	disableunusedchannels	Offset	structure
;	edge	or	subroutinebegin
<	end	pattern	subroutineend
<=	event	per	tabular
<>	events	Periods	terse
=	every	PeriodSetMax	test
>	extension	persistence	testcontrol
>=	false	pingroup	testtime
[	fatal	pinmap	tgusage
]	format	pins	then
acspec	formats	prefix	time
acyclic	formatusage	prevstop	timeplate
after	formwidth	program	timeplates
all	generic	programcontrol	timeset
analysis	if	ps	timing
and	ignore	quiton	timinganalysis
ate	include	relevance	timingdestination
begin	incremental	remedy	trc
both	informative	repeatedcall	true
burst	integer	repeatedvector	us
burstbegin	keySymbolStart	repetition	used
burstend	labelprefix	replace	vector
call	laststate	s	warning
cause	linear	scanrun	wavesource
channelmap	list	select	window
check	loadaddress	setup	
columnformat	location	severity	

3.4 General TCL Syntax

TCL is a block-structured language. The body of the TCL program comprises one large structure, bracketed by opening and closing statements. Within the overall structure are smaller, more specialized structures, or blocks, each bracketed by opening and closing statements.

In its simplest form, a TCL file uses the following syntax:

```
testcontrol <tclName>
   { TclBody }
end testcontrol
```

Valid syntax for the *TclBody* is a list of seven program sections. These sections are referred to as TCL programming blocks. The block names and the WaveBridge processes they control are shown in Table 3-1. The table also lists the page where the block is described in detail.

Table 3-1. TCL Blocks

TCL Block Name	*Purpose*
ATE Constraints on page 3-12	Tester resource limits and control
Pin Groups on page 3-74	Pin cards
Message Overrides on page 3-77	WaveBridge messages
TRC Directives on page 3-80	Tester Rules Checker
Match Directives on page 3-82	TimePlate and Sequence matching
Program Control Directives on page 3-91	Test program format
Pattern Load Directives on page 3-103	Test program pattern bursts

The names identify the beginning of each block. Within each block is data that controls certain aspects of WaveBridge processing. The blocks are optional and can occur in any order, although the order presented in the following example is commonly used. The Pin Groups, Message Directives, Match Directives, and Pattern Load Directives blocks can occur multiple times in a TCL file.

A complete BNF syntactical representation of the TCL file follows:

```
TclProgram ::= “testcontrol” [ <tclName> ]
       { TclBody }
```

```
     “end” “testcontrol”

TclBody ::= ( AteConstraints  | PinGroups |
MessageOverrides
   | TrcDirectives | MatchDirectives |
ProgControlDirectives
   | PatternLoadDirectives )

AteConstraints ::= [ “ate”
       { AteConstraint “;” }
     “end” “ate”]

PinGroups ::= { “pingroup” <PinGroupName>
       { PinGroupBody “;” }
     “end” “pingroup” <PinGroupName>}

MessageOverrides ::={ “message” <messageName>
MessageParams
       MessageBody
     “end” “message” [ <messageName> ] }

TrcDirectives ::=[ “trc”
       { TrcSpec “;” }
     “end” “trc”]

MatchDirectives ::= { “match”
       MatchBody
     “end” “match” }

ProgControlDirectives ::= [ “programcontrol”
       Control
     “end” “programcontrol” ]

ProgControlDirectives ::= { “programcontrol”
     { ControlSpecs }
   “end” “programcontrol” }

PatternLoadDirectives ::={ “burst” [ <burstName> ]
       BurstBody
     “end” “burst” [ <burstName> ] }
```

The following is a complete TCL file that uses only the Match Directives, Program Control Directives, and Pattern Load Directives blocks.

Start Example

Beginning TCL program delimiter

```
testcontrol Advantest
```

Beginning Match Directives block delimiter

```
match
   events
      directory := wave_source;
      start := begin;
      stop := end;
end events
   timing
      directory := timing_source;
      timeplates := ts1, ts2, ts3;
      persistence := 1;
   end timing
   destination
      directory := wave_dest;
   end destination
```

Ending Match Directives block delimiter

```
end match
```

Beginning and ending Program Control Directives block delimiters

```
programcontrol
   formwidth := 72;
   columnformat := every 10;
   comment := every 10 simTime,testTime,cycleNumber,
    memoryAddress,simComment;
end programcontrol
```

Beginning Pattern Load Directives block delimiter

```
burst
   source
      directory := wave_dest;
      start := begin;
      stop := end;
      compress := True;
   end source
   destination
    prefix := adv.prog,
    socket := adv.sock,
    pattern := adv.lpat;
   end destination
```

Ending Pattern Load Directives block delimiter

```
end burst
end testcontrol
```

End Example

3.5 General Program Block Syntax

All TCL program blocks begin with a TCL reserved word that names one of the seven TCL programming blocks. Each TCL program block terminates with the reserved word **end**, followed in some cases by the reserved word block name, such as **pattern**. Between the delimiting reserved words is a definition structure composed of one or more declaration statements. The declaration statements themselves are subdivided into smaller structures that address more specific operations.

The specific syntax description for each program block, presented in the likely order of occurrence in a typical TCL file, follows.

3.6 ATE Constraints

The ATE Constraints block addresses the following tester-specific control categories, listed in alphabetical order. These categories are addressed in detail as either ATE Constraints parameters or sub-blocks, later in this chapter.

When WaveBridge processes a WDB for a tester, it calls upon the Tester file named in the TDS interface. For details, see Chapter 3 in the *Tester Bridges Overview Guide*.

The parameters contained in the Tester file are written in TCL. When running WaveBridge, the required Tester file is read before the optional TCL file. If the optional TCL file is used, and if the optional ATE Constraints block of the TCL file is used, the values specified therein override the corresponding values in the Tester file.

The syntax of the TCL ATE Constraints block is:

```
ate
   {AteConstraint}
end ate
```

AteConstraint is composed of an ATE parameter with a modifier and an expressed constraint. The constraint is a tester-specific value, derived from your ATE's specification manual.

The following example shows a TCL ATE Constraints block.

Start Example

```
ate
   PinInOutMax := 256;        # legal pin numbers are between 1 and 256.
   TimeSetMax := 16;          # Max # of timing sets
   CycleMin := 50ns;          # Min cycle length
   CycleMax := 1.048ms;       # Max cycle length
   CycleResolution := 1ns;    # Cycle length resolution
   PatternRowMax := 65535;    # Max # of pattern rows
   PatternCompression := yes; # Compression search on
   CompareWindowRange   := 1..2; # WSTRBs - # of window strobes
   # CompareEdgeRange   := 1..8; # Optional on T3320
   # CompareWindowRange := 1..4; # Optional on T3320
   CompareType := Window;     # Strobe type: Window, Edge, or Both
   CompareResolution := 125ps;   # Compare edges must be modulo resolution
   CompareWindowMin := 10ns;  # Min window width between T1 and T2
   DelayChannelRange   := 1..4;  # ACLKs - # of force TGs for “edge delay”
   ForceChannelRange   := 1..12; # BCLK + CCLKs - # of forcing TGs
   # ForceChannelRange := 1..16; # Optional BCLK + CCLKs count
   ForcePulseMin := 10ns;     # Min pulse width between T1 and T2
   ForceResolution := 125ps;  # Force edges must be modulo resolution
   ForceConstraint[1] := T1..T1(1) >= 50ns; # Min time between T1 and T1(1)
   ForceConstraint[2] := T2..T2(1) >= 50ns; # Min time between T1 and T1(1)
   CompareEdgeRange     := 1..4; # STRBs - # of edge strobes
   CompareConstraint[1] := T1..T1(1) >= 50ns; # Min time between T1 and T1(1)
   CompareConstraint[2] := T2..T2(1) >= 50ns; # Min time between T1 and T1(1)
   DriveChannelRange := 1..4; # Max # of drivers for I/O switching
   DrivePulseMin     := 10ns; # Driver Specs: 2pf, 1Mohm load
   DriveResolution   := 125ps;   # Drive times must be modulo resolution
   DriveOnMin        := 10ns; # Min ON time during I/O
   DriveOffMin       := 20ns; # Min OFF time during I/O
   DriveConstraint[1] := T1..T1(1) >= 32ns; # Min time between T1 and # T1(1)
   MultiClockCountMax := 128; # Max # of periods in cycle
   MultiClockRateMin := 32ns; # Min multiclock rate (t3 duty cycle)
   MultiClockRateMax := 1.048ms; # Max multiclock rate (t3 duty cycle)
   MultiClockRateResolution := 1ns;   # periods (t3) must be modulo this
      # resolution
   MultiClockLeadingMin := 0ns;  # Min time between T0 and T1
   MultiClockLeadingMax := 16ns; # Max time between T0 and T1
   MultiClockTrailingMin := 0ns; # Min time between T0 and T2
   MultiClockTrailingMax := 1.048ms; # Max time between T0 and T2
   # if MultiClockRate < 64ns TrailMax := 32ns
   #  else TrailingMax := MultiClockRate;
   MultiClockEdgeResolution := 125ps; # edges T1 and T2 must be modulo
      # resolution
```

```
   MuxConversion :=;          # converts mux style automatically
   SubroutineCompression := true;   # can do subroutine compression
   SubroutineNestMax := 0;    # Max subr definition nesting
   SubroutineRowMax := 65535; # Max pattern rows in a subr definition
   SubroutineRowMin := 4;     # Min pattern rows in a subr definition
   SubroutineAtStartLegal := false;   # Is call legal at start of test
      # program?
   SubroutineAtEndLegal := false; # Is call legal at end of test program?
   SubroutineAfterRepeatLegal:= false;# Is call legal on single-row repeat?
   SubroutineRepeatCountMax := 0; # Max count for repeat on subr call
   SubroutineSpacingMin := 0;     # Min # rows required between subr calls
   LoopCompression := true;   # Can do multi-row loop compression
   LoopCountMin := 2;         # Min times the multi-row loop can iterate
   LoopCountMax := 65535;     # Max times the multi-row loop can iterate
   LoopNestMax  := 0;         # Max levels of multi-row loop nesting
   LoopRowMax   := 65535;     # Max pattern rows in multi-row loop
   LoopRowMin   := 4;         # Min pattern rows in multi-row loop
   LoopAtStartLegal := true;  # Is multi-row loop legal at program start?
   LoopRepeatCountMin := 2;   # Min count for repeat in multi-row loop
   LoopRepeatCountMax := 65535;  # Max count for repeat in multi-row loop
   RepeatCompression := true;    # Can do single-row repeat compression
   RepeatAtStartLegal := true;   # Is single-row repeat legal at program start?
   RepeatAtEndLegal   := true;   # Is single-row repeat legal at program end?
   RepeatInSubrLegal := true;    # Is single-row repeat legal in subr ?
   RepeatAtSubrStartLegal := true; # Is single_row repeat legal at subr
      # start?
   RepeatAtSubrEndLegal := false; # Is single_row repeat legal at subr end?
   RepeatInLoopLegal := true;    # Is single-row repeat legal in multi-row
      # loop?
   RepeatCountMin    := 2;    # Min repeat count for a single-row repeat
   RepeatCountMax    := 65535;      # Max repeat count for a single-row repeat
end ate
```

End Example

The following is a complete list of all ATE parameter names that can be used in the ATE Constraints block. These parameters correspond to categories of ATE characteristics that you can modify to suit the needs of your specific test requirements. Explanations of the function and type can be found in the tables accompanying the detailed description of each of the ATE Constraints parameters. Since these parameters address specific capabilities available on test equipment, not all of the parameters are available for all testers.

AteVersion
BurstRowMax
CompareConstraint
CompareEdgeRange
CompareMatchResolution
CompareMultiplexRange
CompareResolution
CompareType
CompareWindowMin
CompareWindowRange
CompressionMemRowMax
CompressionSpacing
CompressionThreshold
CycleMatchResolution
CycleMax
CycleMin
CycleResolution
CycleResolutionTolerance
CycleSteal
DelayChannelRange
DriveChannelRange
DriveConstraint
DriveMatchResolution
DrivePulseMin
DriveResolution
FixtureOffset
ForceChannelRange
ForceConstraint
ForceMatchResolution
ForcePulseMin
ForceResolution
FormatCharMap
FormatSetMax
HizRowMax
IORowMax
LocalTimeSetMax
LoopAtEndLegal
LoopCompression
LoopCountMax
LoopCountMin
LoopNestMax
LoopOutOfSubrLegal
LoopRepeatCountMax
LoopRepeatCountMin
LoopRowMax
LoopRowMin
LoopSpacingMin
MaskRowMax
MemoryModel
MicroCodeCallCost
MicroCodeLoopCost
MicroCodeRepeatCost
MicroCodeRowMax
MicroCodeSubrCost
MonitorModeDeadMin
MultiClockChannelRange
MultiClockConstraint
MultiClockCountMax
MultiClockEdgeResolution
MultiClockLeadingMax
MultiClockLeadingMin
MultiClockPulseMin
MultiClockRateMax
MultiClockRateMin
MultiClockRateResolution
MultiClockTrailingMax
MultiClockTrailingMin
MultiClockType
MultiClockUniqueCountMax
MuxConversion
PatternBoundary
PatternCompression
PatternRowMax
PinInMax
PinInOutMax
PinInOutMin
PinInVoltage
PinOutCurrent
PinOutMax
PinOutVoltage
PostTestCycle
ProbeCloseHold
ProbeCloseSetup
ProbeConstraint
ProbeOpenHold
ProbeOpenMax
ProbeOpenMin
ProbeOpenSetup
ProbeWindowMin
PulseProbeCloseDefault
PulseProbeOpenDefault
RepeatAtEndLegal
RepeatAtStartLegal
RepeatAtSubrEndLegal
RepeatAtSubrStartLegal
RepeatCompression
RepeatCountMax
RepeatCountMin
RepeatInLoopLegal
RepeatInSubrLegal
ScanChannelMax
ScanConstraint
ScanCycleMax
ScanCycleMin
ScanCycleResolution
ScanInSubrLegal
ScanMode
ScanPatternMax
ScanPatternMin
ScanPatternResolution
ScanRegistersOnly
ScanType
SettledProbeCloseDefault
SettledProbeOpenDefault
SingleTimeSets
SpikePulseMin
SocketType
SubroutineAfterRepeatLegal
SubroutineAtEndLegal
SubroutineAtStartLegal
SubroutineCompression
SubroutineDefnMax
SubroutineInLoopLegal
SubroutineNestMax
SubroutineRepeatCountMax
SubroutineRowMax
SubroutineRowMin
SubroutineSpacingMin
TilerTimePlateOrderCost
TimeSetMax
TimeSetMerging
VernierRange

A high-level BNF syntactical representation of the ATE Constraints block follows:

AteConstraints ::= [“ate”
{ AteConstraint “;” }
“end” “ate”]

TCL produces the following messages identifying syntactic errors associated with this block:

```
Duplicate ATE constraint.

Illegal value for this ATE constraint.
```

For details of the complete BNF ATE Constraints syntax, see the appropriate ATE Constraints parameter section that follows.

3.6.1 Compression Spacing Constraints

The Compression Spacing Constraints parameter is used to specify the minimum separation (in pattern rows) of the various type of compression constructs and linear pattern rows.

Table 3-2 lists Compression Spacing Constraints.

Table 3-2. Compression Spacing Constraints

TCL Parameter	*Data Type*	*Description*
CompressionSpacing	integer	Pairwise compression spacing requirements

A complete BNF syntactical representation of the Compression Spacing Constraints parameter follows:

CompressionAdjacency ::= CompressionSpacing

CompressionSpacing ::= “compressionspacing” CompressionItem “..”
CompressionItem “:=” <spacing>

CompressionItem ::= (“burstbegin” | “burstend” | “vector”
| “repeatedvector” | “loopbegin” | “loopend”
| MoreCompressionItems)

MoreCompressionItems ::= (“call” | “repeatedcall” | “subroutinebegin” | “subroutineend” | “scanrun”)

CompressionSpacing gives the minimum separation (in rows) of the various types of compression constructs and normal vectors.

<spacing> is an integer greater than or equal to one.

All *CompressionItems* TCL parameters can be paired in any combination. For example,

```
compressionspacing vector..loopend := 10
```

specifies that between any single vector and the end of a loop, there must be a minimum of ten pattern rows.

3.6.2 Configuration Controls

The Configuration Controls parameters let you tailor WaveBridge output to take advantage of certain features available on specific testers. See the chapter for your WaveBridge in the appropriate TDS tester guide to see if these features are supported for your tester.

Table 3-3 lists Configuration Controls.

Table 3-3. Configuration Controls

TCL Parameter	*Data Type*	*Description*
ATEVersion	string	Tester compiler version
MemoryModel	string	Pattern memory segments
MuxConversion	string	Multiplexing style conversion

A complete BNF syntactical representation of the Configuration Controls parameters follows:

ATEVersion ::= “AteVersion” “:=” ‘ “ ’ <versionString> ‘ ” ’ “;”

MemoryModel ::= “MemoryModel” “:=” <memoryModel> “;”

MuxConversion ::= “MuxConversion” “:=” (“OLDtoNEW” | ““NEWtoOLD” | “NONE”) “;”

The **ATEVersion** parameter lets you specify which test program syntax to use if you have several versions of compilers for your tester and these different versions require different test program syntax.

<versionString> is a string identifying which version of the compiler you want to use. Supported strings are defined in tester-specific WaveBridge chapters; if this parameter is not mentioned in the chapter for your WaveBridge module, your WaveBridge module does not need to support multiple versions of your tester's compiler.

The **MemoryModel** parameter lets you specify different pattern memory segments if supported by your tester. Valid character strings for <memoryModel> are tester-specific.

The **MuxConversion** parameter allows WaveBridge to recognize both styles of specifying multiplexed signals. Of the two styles for specifying multiplexing, one is old and one is new. In old-style multiplexing, signals that have names that are identical except for trailing apostrophes (') indicate a set of multiplexed signals. For example, `in1`, `in1'`, and `in1''` specify a set of multiplexed signals. The new style of multiplexing uses the facilities provided in WGL and WaveMaker's Signal Definition Editor.

In the test file for each tester, the **MuxConversion** parameter is set to convert whichever style of multiplexing is found in the database to the style required by the WaveBridge. If the **MuxConversion** parameter is not used (or is set to `NONE`), older WaveBridges recognize only the old style, and newer WaveBridges recognize only the new style. The style of multiplexing that results is saved in the destination database.

3.6.3 Cycle Constraints

The Cycle Constraints parameters are used for constraints on TimePlate periods. The parameters **CycleMin**, **CycleMax**, and **CycleResolution** create cycle constraints.

Table 3-4 lists Cycle Constraint statements.

Table 3-4. Cycle Constraint statements

TCL Parameter	*Data Type*	*Description*
CycleMin	time	Minimum cycle length
CycleMax	time	Maximum cycle length
CycleResolution	time	Cycle length resolution
CycleResolutionTolerance	time	Cycle length resolution tolerance
CycleMatchResolution	time	TimePlate matching resolution
ScanCycleMax	time	Maximum scan cycle length
ScanCycleMin	time	Minimum scan cycle length
ScanCycleResolution	time	Scan cycle length resolution

A complete BNF syntactical representation of the Cycle Constraints Controls parameter follows:

Cycle := (CycleMinimum | CycleMaximum | CycleResolution
|CycleResolutionTolerance |CycleMatchResolution
| ScanCycleMax | ScanCycleMinScanCycleResolution)

CycleMinimum ::= “cyclemin” “:=” Time

CycleMaximum ::= “cyclemax” “:=” Time

CycleResolution ::= “cycleresolution” “:=” Time

CycleResolutionTolerance ::= “cycleresolutiontolerance” “:=” Time

CycleMatchResolution ::= “cyclematchresolution” “:=” Time

ScanCycleMax ::= “scancyclemax” “:=” Time

ScanCycleMin ::= “scancyclemin” “:=” Time

ScanCycleRes ::= “scancycleresolution” “:=” Time

Time ::= (<intTime> | <floatTime>) [TimeUnit]

TimeUnit ::= (“ps” | “ns” | “us” | “ms” | “s”)

The **CycleResolutionTolerance** parameter defaults to 0s. Use this parameter when frequency and period do not exactly match due to rounding.

The **CycleMatchResolution** parameter describes the tolerance around the cycle boundary in which TimePlate matching can occur. For more information about cycle resolution and TimePlate matching, see page 5-14 in the *WDB Conditioners Guide*. Note that this parameter is not recognized by the SequenceMatch Conditioner.

The **ScanCycleMax**, **ScanCycleMin**, and **ScanCycleResolution** parameters can be used in the Cycle Constraints parameter syntax. The **ScanCycleMax**, **ScanCycleMin**, and **ScanCycleResolution** parameters are discussed in *Scan Controls* on page 3-59.

The other cycle parameters can be graphically portrayed as in Figure 3-1.

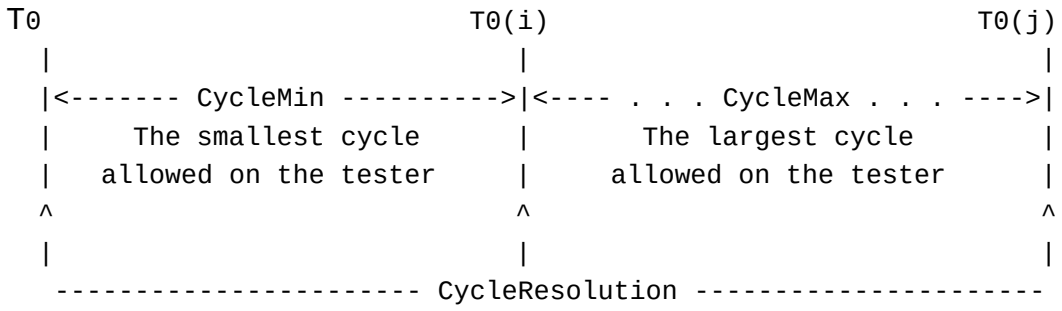

Figure 3-1. Cycle boundary time resolution

3.6.4 Signal Pin DC Controls

You can set group logic threshold levels in your TCL file using the **PinInVoltage**, **PinOutVoltage**, and **PinOutCurrent** parameters. See the chapter for your WaveBridge in the appropriate TDS tester guide to see if these features are supported for your tester.

You can use these parameters to set up voltage levels for input, output, and bidirectional signals.

PinInVoltage specifies an acceptable range of voltage for all input signals.

PinOutVoltage specifies an acceptable range of voltage for all output signals.

PinOutCurrent specifies an acceptable range for output current. This value is expressed in milliamperes, microamperes, and amperes.

A complete BNF syntactical representation of the DC Controls parameter follows:

DC ::= (“pininvoltage” | “pinoutvoltage” | “pinoutcurrent”) “:=” DCvalues

DCvalues ::= [DClabel] “[” DCvalue { “,” DCvalue } “]”

DClabel ::= (<dcName> | <dcString>)

DCvalue ::= [DCsign] <voltsOrAmps> DCunit

DCsign ::= (“+” | “-”)

DCunit ::= (“ma” | “ua” | “a” | “v”)

The following shows an example of the use of the DC Controls parameters.

Start Example

```
ate
   PinInVoltage := Vi [0.5v, 4.5v];
   PinOutVoltage := Vo [0.45v, 4.75v];
   PinOutCurrent := [6.0ma,-10.0ma];
end ate
```

End Example

<dcName> and <dcString> are user-defined alphanumeric strings that let you attach a name to help identify a defined voltage or current range. <voltsOrAmps> is a numeric value associated with voltage level.

DCunit specifies whether the unit of electrical measurement is to be milliampere, microampere, ampere, or volt.

DCsign indicates whether the value represented by <voltsOrAmps> is a positive or negative value.

The defaults for these parameters are tester-specific. To find the default DC Controls values for your tester, use your system’s text editor to view the Tester file for your WaveBridge. The Tester file is located in the directory that you set as the TDSDIR environment variable when you installed your TDS software.

3.6.5 Signal Sequence Control

The Signal Sequence Controls are used to customize the signal sequencing capabilities of test systems. Signal sequencing is the capability to control signal channels and their associated resources as they are presented to a DUT prior to any testing.

You use a global TCL parameter, **SeqSignalTzero**, to define a window of time in which all signal sequencing must occur. You can then define up to eight sequence times and associated tester resources for the WDB signals. Figure 3-2 represents an example of signal sequencing where a global sequence time window and four sequence time/window combinations are defined.

The TCL parameters **SeqSignal1** through **SeqSignal8** define the sequence times and **SeqResource1** through **SeqResource8** define the tester resources. The tester resource is mapped to the sequence time according to the number suffixes on each parameter (**SeqSignal1** maps to **SeqResource1**, and so on).

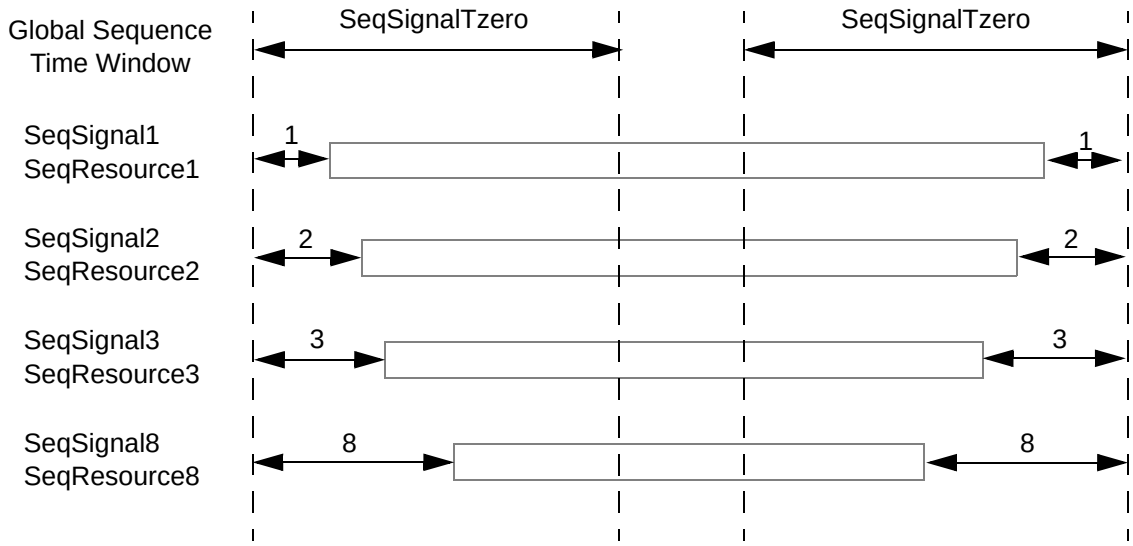

Figure 3-2. Signal Sequencing

The mechanism for associating the WDB signals with sequence times and tester resources is specific to each WaveBridge. One such mechanism uses eight reserved pin groups (one for each time-resource pair), with the pin groups being associated with the time-resource pairs according to a number suffix in the reserved pin group name.

The BNF syntactical representation for the Signal Sequence Control parameters is as follows.

Start Example

```
SeqSignalTzero := TimeValue;

SeqSignal1 := TimeValue;
SeqSignal2 := TimeValue;
SeqSignal3 := TimeValue;
SeqSignal4 := TimeValue;
SeqSignal5 := TimeValue;
SeqSignal6 := TimeValue;
SeqSignal7 := TimeValue;
SeqSignal8 := TimeValue;

TimeValue := ( <intTime> | <floatTime> ) [ TimeUnit ]
TimeUnit := ( “ps” | “ns” | “us” | “ms” | “s” )

SeqResource1 := <ResourceName>;
SeqResource2 := <ResourceName>;
SeqResource3 := <ResourceName>;
SeqResource4 := <ResourceName>;
SeqResource5 := <ResourceName>;
SeqResource6 := <ResourceName>;
SeqResource7 := <ResourceName>;
SeqResource8 := <ResourceName>;
```

End Example

<ResourceName> is a string that specifies the resource being set. The actual strings are specific to each Tester WaveBridge that supports signal sequencing. Refer to the WaveBridge documentation for the specific string values.

3.6.6 Power Supply DC Controls

The Power Supply DC Controls parameters allow you to establish the default power supply settings for up to eight power supplies (if your tester and WaveBridge support this capability).

Table 3-5. Power Supply DC Controls

TCL Parameter	*Data Type*	*Description*
PsVoltage	voltage	Power supply default output voltage value.
PsVoltageMax	voltage	Power supply maximum output voltage value.
PsVoltageMin	voltage	Power supply minimum output voltage value.
PsVoltageRange	string	Power supply default voltage range.
PsMeasurementRange	string	Power supply default current measurement range.
PsNegCurrentClamp	current	Power supply default negative current clamp value.
PsNegCurrentClampMax	current	Power supply maximum negative current clamp value.
PsNegCurrentClampMin	current	Power supply minimum negative current clamp value.
PsPosCurrentClamp	current	Power supply default positive current clamp value.
PsPosCurrentClampMax	current	Power supply maximum positive current clamp value.
PsPosCurrentClampMin	current	Power supply minimum positive current clamp value.
PsSlewRate	time	Power supply default voltage slew rate.
PsSlewRateMax	time	Power supply maximum voltage slew rate.
PsSlewRateMin	time	Power supply minimum voltage slew rate.
PsSequenceTime	time	Power supply default sequence time.
PsSequenceWindow	time	Power supply global window for sequence time.
PsFilter	Boolean	Power supply filter control.

The complete TCL BNF syntactical representation for Power Supply DC Controls parameters is as follows:

PsSequenceWindow := [TimeValue]

PsVoltage := [“Supply”] SupplyNumber, VoltageValue

PsVoltageMin := [“Supply”] SupplyNumber, VoltageValue

PsVoltageMax := [“Supply”] SupplyNumber, VoltageValue

PsMeasurementRange := [“Supply”] SupplyNumber, Value

PsVoltageRange := [“Supply”] SupplyNumber, Value

PsNegCurrentClamp := [“Supply”] SupplyNumber, CurrentValue

PsNegCurrentClampMin := [“Supply”] SupplyNumber, CurrentValue

PsNegCurrentClampMax := [“Supply”] SupplyNumber, CurrentValue

PsPosCurrentClamp := [“Supply”] SupplyNumber, CurrentValue

PsPosCurrentClampMin := [“Supply”] SupplyNumber, CurrentValue

PsPosCurrentClampMax := [“Supply”] SupplyNumber, CurrentValue

PsSlewRate := [“Supply”] SupplyNumber, TimeValue

PsSlewRateMin := [“Supply”] SupplyNumber, TimeValue

PsSlewRateMax := [“Supply”] SupplyNumber, TimeValue

PsSequenceTime := [“Supply”] SupplyNumber, TimeValue

PsFilter := [“Supply”] SupplyNumber, BooleanValue

SupplyNumber := integer

BooleanValue := (“TRUE” | “FALSE”)

TimeValue := (<intTime> | <floatTime>) [TimeUnit]

CurrentValue := [Sign] <float> CurrentUnit

VoltageValue := [Sign] <float> VoltUnit

Sign := (“+” | “-”)

VoltUnit := (“v”)

TimeUnit := (“S” | “mS” | “uS”)

CurrentUnit := (“ua” | “ma” | “a”)

Value := String

The following TCL excerpt defines the power supply DC parameters for one power supply.

Start Example

```
ate

PsSequenceWindow        := 10mS;

PsVoltageRange          := Supply 1, “R8V”;
PsMeasurementRange      := Supply 1, “M80MA”;
PsVoltage               := Supply 1, 5.0V;
PsVoltageMin            := Supply 1, 10.0V;
PsVoltageMax            := Supply 1, -10.0V;
PsPosCurrentClamp       := Supply 1, 500.0mA;
PsPosCurrentClampMin    := Supply 1, 0.0A;
PsPosCurrentClampMax    := Supply 1, 550.4mA;
PsNegCurrentClamp       := Supply 1, -500.0mA;
PsNegCurrentClampMin    := Supply 1, -550.4mA;
PsNegCurrentClampMax    := Supply 1, 0.0A;
PsSlewRate              := Supply 1, 1mS;
PsFilter                := Supply 1, TRUE;
PsSequenceTime          := Supply 1, 0ms;

end ate
```

End Example

Note the relationship between **PsSequenceWindow** and **PsSequenceTime**. **PsSequenceWindow** defines a window of time in which all power supply sequencing must occur, so the **PsSequenceTime** value must be within that window of time. This approach to controlling power supply sequencing is similar to the approach for controlling signal sequencing, which is described in *Signal Sequence Control* on page 3-23.

3.6.7 Fixture Controls

The Fixture Controls parameters let you control custom test fixtures if they are available on your tester.

Table 3-6. Fixture Controls

TCL Parameter	*Data Type*	*Description*
FixtureOffset	time	Delay introduced by cables to fixture
SocketType	string	Name of supported fixture
VernierRange	time	Edge adjustment range (per pin)

A complete BNF syntactical representation of the Fixture Controls parameters follows:

Fixture ::= (SocketType | VernierRange | FixtureOffset)

VernierRange ::= “vernierrange” “:=” Time

SocketType ::= “sockettype” “:=” <socketTypeName>

FixtureOffset ::= “fixtureoffset” “:=” [“-”] Time

Time ::= (<intTime> | <floatTime>) [TimeUnit]

TimeUnit ::= (“ps” | “ns” | “us” | “ms” | “s”)

The **FixtureOffset** parameter accepts a time value, and the value can be negative as well as positive. Use this value to specify an offset caused by a custom test fixture. The default is `0ns`.

The **SocketType** parameter specifies the type of socket used by the tester. Legal values are tester-specific and are documented in the chapter for your specific WaveBridge.

The **VernierRange** parameter accepts a time value. The default is `0ns`.

NOTE

Since all of the Fixture Controls parameters are tester-specific in nature, the following descriptions and examples are for general discussion only. See the WaveBridge chapter for your specific tester to see if Fixture Controls parameters are supported.

Each tester pin has a set of timing verniers that allow you to adjust the edges of the timing generator driving that pin. The value of the **VernierRange** TCL parameter expresses the range within which you can adjust base timing generator values on tester pins. The default value may vary depending upon the tester.

Each edge on each pin has an adjustment range of plus or minus half the value for **VernierRange**. Each **SocketType**, as defined for the specific socket available with supported testers, has an implied time value (based upon factory specifications) that is associated with it. You cannot change this implied value. The implied value is divided by two and subtracted from all force edges and added to all compare edges to yield the available range of adjustment for the pin.

A similar adjustment is also done with the **FixtureOffset** parameter; the value for **FixtureOffset** value is subtracted from all force edges and added to all compare edges, but the **FixtureOffset** value is not divided by two before this operation.

To calibrate for the path delay of the fixture and socket, the **pathdelay** TDS Standard Relational Format (SRF) keyword in the TDS Pin Assignment file supplies the delay value. This value is subtracted from all force edges and added to all compare edges. The value you provide for **pathdelay** specifies the total delay for a signal.

An example using the signal `enable` follows.

A TCL file fragment with Fixture Control parameters:

Start Example

```
ate
   SocketType := ”OPEN_IO”;
   VernierRange := 20ns;
   FixtureOffset := 2ns;
end ate
```

End Example

Corresponding Pin Assignment file entry showing the relation for the signal `enable`:

Start Example

```
#% signal   pathdelay   edge1_delay edge2_delay
   enable   -1.5ns      -100ps      50ps
```

End Example

NOTE

*The relations expressed by the edge<**n**>_delay values in the Pin Assignment file do not affect the VernierRange adjustment.*

Table 3-7 shows the final vernier adjustment range available at various stages of the definition process.

Table 3-7. Example Vernier adjustment ranges

Stage of Definition Process	*Range*	*Range*
Initial **VernierRange** value (20ns)	-10ns, +10ns	-10ns, +10ns
After **SocketType** value (implied value of 1.20ns) is applied	-9.4ns, +10.6ns	-10.6ns, +9.4ns
After **FixtureOffset** value (2ns) is applied	-7.4ns, +12.6ns	+7.4ns, -12.6ns
After **pathdelay** value (-1.5ns) is applied	-4.4ns, +15.6ns	+4.4ns, -15.6ns

3.6.8 Force/Compare/Drive Constraints

The Force/Compare/Drive Constraints parameters are used for constraints on the corresponding type of tester waveform.

The term “drive” describes when a switch from forcing to comparing occurs within one cycle. The minimum pulse or window width is expressed with the **Pulse** and **Window** modifiers. The resolution of the three parameter types is described using the **Resolution** modifier. Additionally, strobe types being Forced, Compared, or Driven can be distinguished using the **IsBiDir**, **IsEdge**, **IsInput**, **IsOutput**, or **IsWindow** qualifiers used to differentiate the type of strobe type being evaluated.

Table 3-8 lists the Force/Compare/Drive Constraint statements.

Table 3-8. Force/Compare/Drive Constraint statements

TCL Parameter	*Data Type*	*Description*
ForcePulseMin	time	Minimum pulse width between T1 and T2
ForceResolution	time	Forcing edges must be modulo this resolution
DelayChannelRange	Range	Number of forcing resources used for “edge delay”
ForceChannelRange	Range	Number of forcing tester resources

Table 3-8. Force/Compare/Drive Constraint statements (continued)

TCL Parameter	*Data Type*	*Description*
ForceMatchResolution	time	TimePlate matching force edge resolution
CompareWindowMin	time	Minimum window width between T1 and T2
CompareResolution	time	Compare edges must be modulo this resolution
CompareEdgeRange	Range	Maximum number of edge strobes
CompareMatchResolution	time	TimePlate matching compare edge resolution
CompareType	Edge | Window | Both	Edge or window strobes only or both
CompareWindowRange	Range	Range of compare window tgs
IOswitchDeadZone	time	No events can exist in the specified time around I/O
Drive2CompareConstraints	time	Specifies a time distance to maintain between compare time and drive time.
DriveOnMin	time	Minimum time between T1 and T2 (driver enabled)
DriveOffMin	time	Minimum time between T2 and T0(2) (driver disabled)
DriveResolution	time	Drive enable/disable times must be modulo this resolution
DriveMatchResolution	time	TimePlate matching drive edge resolution
DriveChannelRange	Range	Number of I/O switching tester resources
IsBiDir	None	Qualifies strobe type is BiDirectional
IsEdge	None	Qualifies strobe type is an Edge
IsInput	None	Qualifies strobe type is Input
IsOutput	None	Qualifies strobe type is Output
IsWindow	None	Qualifies strobe type is a Window
SpikePulseMin	time	Minimum pulse width between T1 and T2 (TDS ProbeBridge only)
MonitorModeDeadMin	time	Minimum time between force T1 and compare T1

Figure 3-3 is a conceptual model of the application of Force/Compare/Drive Constraints.

```
T0                     .......                                  T0(1)
|                      :    --:-----------------                    |
|<-- T0..T1 >= min ---->:  |<-- ForcePulseMin -->|                  |
|-------------------------   :                   ---------------|
|                      :  ^  : ForceResolution  ^                   |
|                      :.....:                                      |
|<-- T0..T1 <= max ----------->                                     |
|                                                                   |
|                        .......                                    |
|                        :     :                                    |
|<-- T0..T1 >= min ------>:   -----------------------               |
|- - - - - - - - - - - - -:- |<-- CompareWindowMin -->|-- - - - - - |
|                        :   -----------------------                |
|                        :  ^  : CompareResolution   ^               |
|                        :.....:                                    |
|<-- T0..T1 <= max ------------->                                   |
|                                                                   |
|                      ......  -------             --------          |
|<-- T0..T1 >= min ---->:    : |       |      - - - |////////|- -  |
| - - - - - - - - - - - :- ----         ------       --------       |
|                      :  ******************                        |
|                      :  <-- DriveOnMin -->|<-- DriveOffMin --->|
```

Figure 3-3. Example Force/Compare/Drive Constraints

A complete BNF syntactical representation of the Force Constraints parameter follows:

Force ::= (ForceChannelRange | ForcePulseMinimum | ForceConstraints
| ForceResolution | ForceMatchResolution |DelayChannelRange)

ForceChannelRange ::= “forcechannelrange” Subscript “:=” Range

DelayChannelRange ::= “delaychannelrange” Subscript “:=” Range

Subscript ::= [“[” <constIndex> “]”]

Range ::= IntRange [“pins” PinRange] [“per” <perValue>]

IntRange ::= <lowerBound> “..” <upperBound>

PinRange ::= <lowerBound> “..” <upperBound>

ForcePulseMinimum ::= “forcepulsemin” “:=” Time

ForceConstraints ::= “forceconstraint” Subscript “:=” TimeExpr

ForceResolution ::= “forceresolution” “:=” Time

ForceMatchResolution ::= “forcematchresolution” “:=” Time

Time ::= (<intTime> | <floatTime>) [TimeUnit]

TimeUnit ::= (“ps” | “ns” | “us” | “ms” | “s”)

A complete BNF syntactical representation of the Compare Constraints parameter follows:

Compare ::= (CmpControl | CmpConstraints | CmpMatchResolution |CmpResolution | CmpWindowMinimum)

CmpControl ::= (CmpEdgeRange | CmpWindowRange | CmpStrobeType)

CmpEdgeRange ::= “compareedgerange” Subscript “:=” Range

CmpWindowRange ::= “comparewindowrange” Subscript “:=” Range

Range ::= IntRange [“pins” PinRange] [“per” <perValue>]

CmpStrobeType ::= “comparetype” “:=” (“edge” | “window” | “both”)

CmpConstraints ::= “compareconstraint” Subscript “:=” TimeExpr

CmpMatchResolution ::= “comparematchresolution” “:=” Time

CmpResolution ::= “compareresolution” “:=” Time

CmpWindowMinimum ::= “comparewindowmin” “:=” Time

Time ::= (<intTime> | <floatTime>) [TimeUnit]

TimeUnit ::= (“ps” | “ns” | “us” | “ms” | “s”)

A complete BNF syntactical representation of the Drive Constraints parameter follows:

Drive := (DriveControl | DrivePulseMinimum | DriveResolution)

DriveControl ::= (DriveChannelRange | DriveMatchResolution | DriveConstraints | DriveOnMinimum | DriveOffMinimum)

DriveChannelRange ::= “drivechannelrange” Subscript “:=” Range

Range ::= IntRange [“pins” PinRange] [“per” <perValue>]

DrivePulseMinimum ::= “drivepulsemin” “:=” Time

DriveResolution ::= “driveresolution” “:=” Time

DriveMatchResolution ::= “drivematchresolution” “:=” Time

DriveConstraints ::= “driveconstraint” Subscript “:=” TimeExpr

Drive2CompareConstraints := drive2compareconstraints” “:=” Time

DriveOnMinimum ::= “driveonmin” “:=” Time

DriveOffMinimum ::= “driveoffmin” “:=” Time

IOswitchDeadZone ::= “ioswitchdeadzone” “:=” <time>

Time ::= (<intTime> | <floatTime>) [TimeUnit]

TimeUnit ::= (“ps” | “ns” | “us” | “ms” | “s”)

The <constIndex> is a numeric value enclosed in brackets ([ ]) that is used to differentiate among constraints when there are multiple instances of the constraint allowed, as in **ForceConstraint**[**1**]. See page 3-36 for an example of the use of <constIndex>.

The **ForceMatchResolution**, **CompareMatchResolution**, and **DriveMatchResolution** parameters are used by the TimePlate matching step in WaveBridge, TimePlate Match Conditioner, and SequenceMatch Conditioner. The values represented by these parameters default to 0ns if they are not specified in the TCL file. The settings describe the tolerance around force, compare, and I/O switching edges that is used to give the TimePlate matching process in WaveBridge more leeway in matching. The **Channel** modifier is used to describe the number of timing resources that are available for the various types of channels.

The **ForceChannelRange** parameter describes the number of available forcing timing generators.

The **CompareChannelRange** parameter describes the number of available comparing timing generators.

The **Drive2CompareConstraints** parameter (sometimes referred to as a “round-trip” delay) performs a DriveOff to Compare strobe timing measurement where a signal originates in the Pin driver, goes to the DUT, and then returns back to the Comparator. No

other time measurement allows the interaction between the fundamental types, Drive and Compare.

This parameter is used on testers that require that the Compare time be kept a certain distance from the Drive timing due to DUT and Pin driver cable delays.

The **DriveChannelRange** parameter describes the number of available driver timing resources.

The **DelayChannelRange** parameter describes the number of forcing timing generators used for programming a delay edge on those testers that support delay edges.

The first part of the range describes the inclusive set of timing generators that are available. The optional pin clause describes how a range of timing generators are restricted to a range of pins.

The **Channel** parameters allow arrays to be specified so that several timing generator-to-pin constraints can be described.

The **per** reserved word begins a TCL clause that modifies **CompareChannelRange**, **DriveChannelRange**, and **ForceChannelRange** TCL parameters.

The **per** reserved word permits you to assign available timing generators (channels) to groups of pins within the range of available pins using the <perValue> to specify the group size. For example:

```
CompareChannelRange[2] := 3..6 pins 1..256 per 32;
```

means that you are assigning timing generators 3 through 6 to be available for pins 1 through 256, with unique assignment of timing generators within sub-ranges of 32-pin groups.

The **IOswitchDeadZone** parameter specifies the amount of time on either side of an I/O switch in which no events can exist. An example follows:

Start Example

```
IOswitchDeadZone := 5ns;
```

End Example

The **MonitorModeDeadMin** parameter specifies the time from the leading edge of the input data region to the leading edge of the compare window, and also from the trailing edge of the compare window to the trailing edge of the input data region.

The example below, shows that timing generators 1 through 8 inclusive can be used for either **Force** or **Compare** channels with no pin restrictions. Timing generators 9 through 16 are restricted to pins 1 to 64, 19 through 26 are restricted to pins 65 to 128, and 27 through 34 are restricted to pins 129 to 192.

Example of channel modifier usage:

Start Example

```
ForceChannelRange[1] := 1..8;
ForceChannelRange[2] := 9..16 pins 1..64;
ForceChannelRange[3] := 19..26 pins 65..128;
ForceChannelRange[4] := 27..34 pins 129..192;
CompareChannelRange[1] := 1..8;
CompareChannelRange[2] := 9..16 pins 1..64;
CompareChannelRange[3] := 19..26 pins 65..128;
CompareChannelRange[4] := 27..34 pins 129..192;
DriveChannelRange[1] := 13..18;
DriveChannelRange[2] := 35..36;
```

End Example

You can impose your own restrictions on how timing generators are assigned to pins using the channel parameters (available on testers that require more than one timing generator per pin). A simpler alternative is to use the **tgusage** statement in the Pattern Load Directives block.

Table 3-9 lists the Timing Constraints supported.

Table 3-9. Timing Constraints supported

TCL Parameter	*Data Type*	*Description*
ForceConstraint	TimeExpr	Timing constraint expression
CompareConstraint	TimeExpr	Timing constraint expression
DriveConstraint	TimeExp	Timing constraint expression

The **ForceConstraint**, **CompareConstraint**, and **DriveConstraint** parameters let you describe multiple timing constraints by specifying an array. The example below shows some of these parameters combined with timing expressions.

Example of Force, Compare, and Drive parameters combined with timing expressions:

Start Example

```
ForceConstraint[1] := T1..T1(1) > 50ns;
ForceConstraint[2] := T2..T1(1) > 30ns;
ForceConstraint[3] := T2..T2(1) > 70ns;
ForceConstraint[4] := T2..T0(2) > 300ns;
ForceConstraint[5] := T2..T0(1) > 20ns;
CompareConstraint[1] := T1..T1(1) > 50ns;
CompareConstraint[2] := T2..T1(1) > 30ns;
CompareConstraint[3] := T2..T2(1) > 20ns;
CompareConstraint[4] := T2..T0(2) > 300ns;
CompareConstraint[5] := T3..T1(1) > 50ns
DriveConstraint[1] := T1..T1(1) > 60ns;
DriveConstraint[2] := T2..T2(1) > 60ns;
DriveConstraint[3] := T2..T0(2) > 60ns;
```

End Example

Figure 3-4 lists a conceptual model of timing constraints.

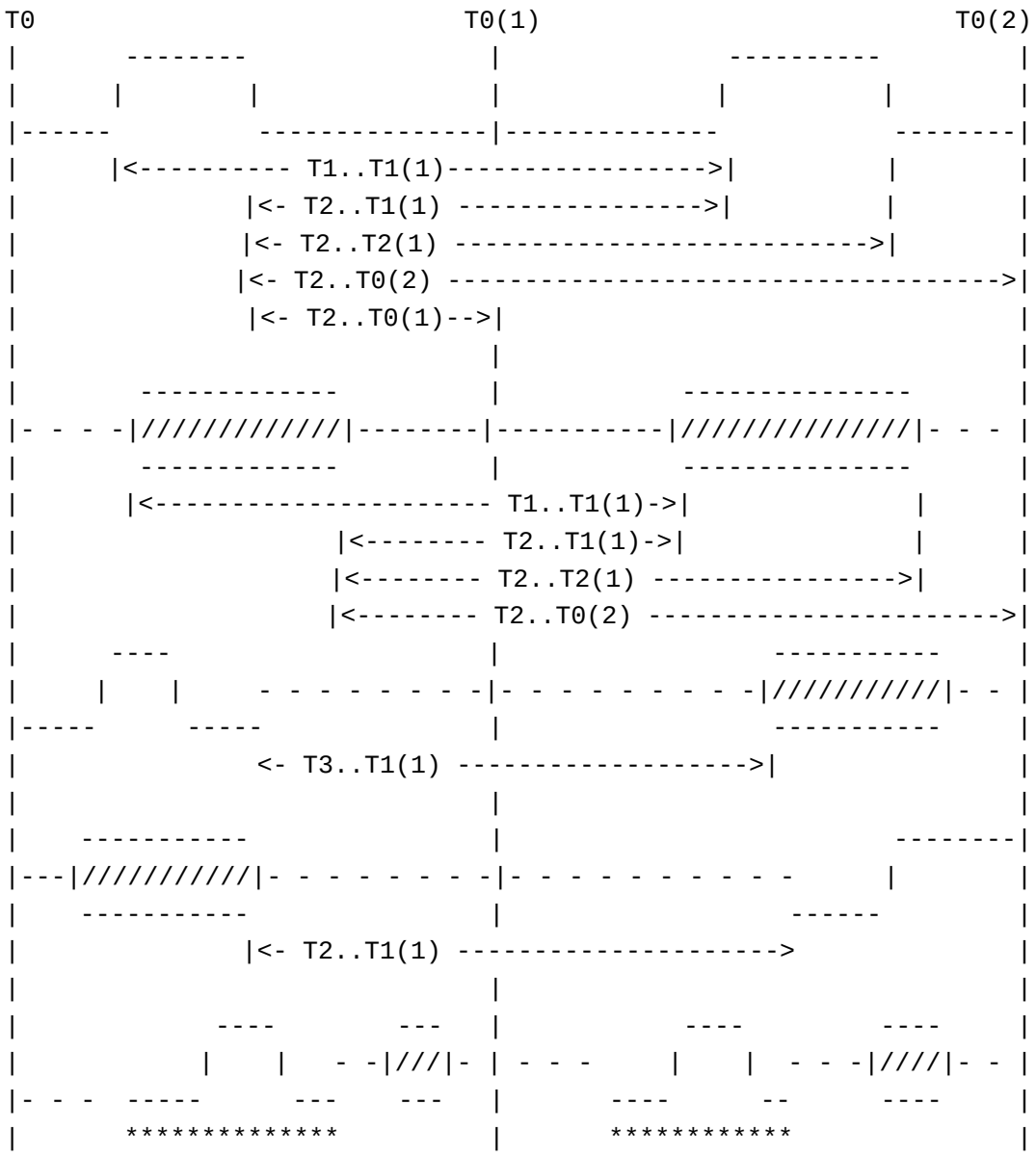

Figure 3-4. A conceptual model of Force, Compare, and Drive parameters combined with timing expressions

TCL produces the following messages identifying syntactic errors associated with the Format/Compare/Drive Constraints block:

```
Range lower bound greater than upper bound.

Range lower bound less than 1.

Range overlaps previous range.

Pin lower bound greater than upper bound.

Pin lower bound less than 1.

Pin upper bound greater than PinInOutMax value.
```

3.6.9 Format Controls

The Format Controls parameters let you control the number of unique sets of formats that a format-on-the-fly test supports.

Table 3-10 lists Format Controls.

Table 3-10. TCL Format Controls

TCL Parameter	*Data Type*	*Description*
CycleSteal	Boolean	Suppress/enable cycle steals by tester
FormatChartMap	string	Redefine set of format characters
FormatSetMax	integer	Maximum number of format sets available

A complete BNF syntactical representation of the Format Controls parameter follows:

CycleSteal ::= “cyclesteal” “:=” Boolean

FormatSet ::= “formatsetmax” “:=” <formatsetMaximum>

FormatCharMap ::= “formatcharmap” “:=” <formatCharMap>

The **CycleSteal** TCL parameter is used to allow WaveBridge to produce test programs that require “dead cycles” on testers that support cycle steal.

When changing format/mask/invert data, micro-instruction execution time on the tester may be longer than one tester cycle (in the case of small cycle lengths). Under this circumstance, the tester holds all inputs at the level they ended the previous cycle, and masks all outputs, for one or more cycles (whatever is required to finish the micro-instructions). Dynamic devices may not work correctly due to this (because of critical timing relationships across cycles). Use the TCL statement:

```
CycleSteal := false;
```

to instruct WaveBridge not to perform microcode operations that require cycle steals on the tester.

On certain testers, the **FormatCharMap** TCL parameter lets you override the set of character definitions, if this is allowed on your tester. The <formatCharMap> expresses the mapping relationship between two character strings. The first string of characters is mapped to the second string of characters. The two strings are separated by colons, before, after, and between, as in the example:

```
FormatCharMap := ":A:r:"
```

See the chapter for your specific WaveBridge in the appropriate TDS tester guide to see if this feature is supported for your tester.

If a tester's architecture allows format switching on the fly, the **FormatSetMax** parameter describes how many unique format sets are available. The **FormatSetMax** parameter is required on testers that can switch formats on-the-fly and is ignored on testers that assign a single format to a pin.

3.6.10 Loop Constraints

The Loop Constraints parameter controls the utilization of loops in your test program, depending on your tester's capabilities. A loop is a labeled collection of pattern rows, subroutine calls, repeats, or loops, that can be repeated a specified number of times.

Example WGL loops:

Start Example

```
waveform loop1
        signal a : input; end
        pattern load (a)
                loop 101
                        vector (+) := [1];
                end
                loop 5
                        vector (+) := [1];
                        vector (+) := [-];
                        vector (+) := [1];
                        vector (+) := [-];
                        repeat 2 vector (+) := [1];
                        repeat 10 vector (+) := [X];
```

```
                end
                loop 5
                        vector (+) := [Z];
                        vector (+) := [Z];
                        vector (+) := [-];
                        vector (+) := [Z];
                        loop 4
                                vector (+) := [-];
                                vector (+) := [Z];
                                vector (+) := [-];
                                vector (+) := [1];
                                loop 3
                                        vector (+) := [X];
                                        vector (+) := [Z];
                                        vector (+) := [1];
                                        vector (+) := [0];
                                end
                        end
                end
        end
end
```

End Example

Table 3-11 lists Loop Constraints.

Table 3-11. Loop Constraints

TCL Parameter	*Data Type*	*Description*
LoopCompression	Boolean	Multi-row loop compression legal
LoopCountMin	integer	Minimum times the multi-row loop must iterate
LoopCountMax	integer	Maximum times the multi-row loop can iterate
LoopNestMax	integer	Maximum levels of multi-row loop nesting
LoopRowMax	integer	Maximum pattern rows in multi-row loop definition
LoopRowMin	integer	Minimum pattern rows in multi-row loop definition
LoopAtEndLegal	Boolean	TRUE if multi-row loop legal at end of test program
LoopAtStartLegal	Boolean	TRUE if multi-row loop legal at start of test program
LoopRepeatCountMin	integer	Minimum count for single-row repeat inside multi-row loop
LoopRepeatCountMax	integer	Maximum count for single-row repeat inside multi-row loop
LoopSpacingMin	integer	Minimum count for pattern rows necessary between adjacent subroutine calls, single-row repeats, or multi-row loops

A complete BNF syntactical representation of the Loop Constraints parameter follows:

Loop ::= (LoopCompression | LoopCountMaximum | LoopCountMinimum | LoopNest | LoopRowMinimum |LoopRowMaximum
| LoopConstraint)

LoopConstraint ::= (LoopSpacingMin | LoopAtStartLegal
| LoopAtEndLegal| LoopRepeatCountMinimum
| LoopRepeatCountMaximum)

LoopCompression ::= “loopcompression” “:=” Boolean

LoopCountMaximum ::= “loopcountmax” “:=” <loopCountMaximum>

LoopCountMinimum ::= “loopcountmin” “:=” <loopCountMinimum>

LoopNest ::= “loopnestmax” “:=” <loopNestMaximum>

LoopRowMinimum ::= “looprowmin” “:=” <loopRowMinimum>

LoopRowMaximum ::= “looprowmax” “:=” <loopRowMaximum>

LoopSpacingMin ::= “loopspacingmin” “:=” <loopSpaceMin>

LoopAtStartLegal ::= “loopatstartlegal” “:=” Boolean

LoopAtEndLegal ::= “loopatendlegal” “:=” Boolean

LoopRepeatCountMinimum ::= “looprepeatcountmin” “:=” <loopRepeatMinimum>

LoopRepeatCountMaximum ::= “looprepeatcountmax” “:=” <loopRepeatMaximum>

If multi-row loop compression is not legal on the tester (or not desired), the **LoopCompression** parameter is set to `false`.

The minimum and maximum restrictions on the number of pattern rows contained in the body of a loop are described by the **LoopRowMin** and **LoopRowMax** parameters. The maximum iteration count of the loop is described by the **LoopCountMax** parameter.

The **LoopNestMax** parameter describes to what level internal loops may be contained within an outermost loop. The default for this parameter is zero (`0`), no nesting allowed.

The **LoopAtEndLegal** parameter describes whether it is legal to have a loop at the very end of the test program’s pattern rows.

The **LoopAtStartLegal** parameter describes whether it is legal to have a loop at the very start of the test program’s pattern rows.

The **LoopRepeatMin** and **LoopRepeatMax** parameters describe limitations on the iteration count of single-row repeats inside of multiple-row loops. Note that **LoopRepeatMin** and **LoopRepeatMax** are essentially the same values described by the **RepeatCountMin** and **RepeatCountMax** parameters but are used to limit repeats occurring within defined loops.

3.6.11 Microcode Constraints

The Microcode Constraints parameter controls your tester’s limitations on the amount of microcode memory used to store test program control information.

Table 3-12 lists Microcode Constraints.

Table 3-12. Microcode Constraints

TCL Parameter	*Data Type*	*Description*
MicroCodeRowMax	integer	Maximum microcode rows in the tester
MicroCodeRepeatCost	integer	Maximum microcode rows used by a single-row repeat
MicroCodeSubrCost	integer	Maximum microcode rows used by a subroutine definition
MicroCodeCallCost	integer	Maximum microcode rows used by a subroutine call
MicroCodeLoopCost	integer	Maximum microcode rows used by a multi-row loop

A complete BNF syntactical representation of the Microcode Constraints parameter follows:

MicroCode ::= (McodeRowMaximum | McodeRepeatCost
| McodeSubrCost | McodeCallCost | McodeLoopCost)

McodeRowMaximum ::= “microcoderowmax” “:=” <microcodeRowMaximum>

McodeRepeatCost ::= “microcoderepeatcost” “:=” <microcodeRepeatCost>

McodeSubrCost ::= “microcodesubrcost” “:=” <microcodeSubrCost>

McodeCallCost ::= “microcodecallcost” “:=” <microcodeCallCost>

McodeLoopCost ::= “microcodeloopcost” “:=” <microcodeLoopCost>

Each of the four microcode control mechanisms has an associated cost in terms of the number of microcode rows they consume, and this is denoted by the **Cost** modifier.

Single row loops are denoted by the **Repeat** modifier and describe a test program row that is repeated some number of times. Multiple row loops are denoted by the **Loop** modifier and describe a sequence of test program rows that are repeated some number of times.

Subroutine definitions are denoted by the **Subr** modifier and describe a block of test program rows that are invoked or called from other locations in a test program. When the subroutine block is finished, control returns to the row after the invocation.

Subroutine invocations are denoted by the **Call** modifier and describe where the flow of control changes to a subroutine block.

The total size of microcode memory is described by the **MicroCodeRowMax** parameter. The integer represents the number of rows (not bytes or words).

3.6.12 Multiple Clocking Constraints

The Multiple Clocking Constraints parameter describes your tester's capability to generate a multiple-pulse forcing waveform.

Figure 3-5 is an example of a multiclock cycle.

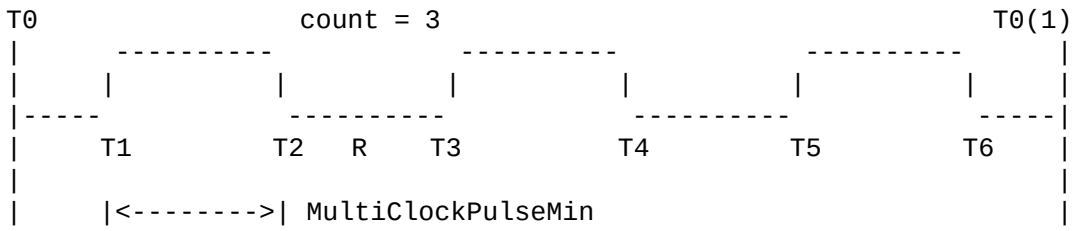

Figure 3-5. Example multiclock cycle

The `T1` and `T2` edges locate the rising and falling edges of the clock and the `R` rate locates the clock period of the multiclock waveform. This clock period is the duty cycle with which the multiclock waveform may repeat itself. The count field describes how many clock periods to fit between `T0` and `T0(1)`.

A complete BNF syntactical representation of the Multiple Clocking Constraints parameter follows:

Mclk ::= (MclkType | MclkRate | MclkLeading | MclkTrailing
| MclkRecovery | MclkEdgeResolution | MclkCountMaximum)

MclkType ::= “multiclocktype” “:=” (“none” | “cyclic” | “acyclic”)

MclkRate ::= (MclkRateResolution | MclkRateMinimum
| MclkRateMaximum)

MclkLeading ::= (MclkLeadingMinimum | MclkLeadingMaximum)

MclkTrailing ::= (MclkTrailingMinimum | MclkTrailingMaximum)

MclkRecovery ::= (MclkPulseMin | MclkConstraints)

MclkEdgeResolution ::= “multiclockedgeresolution” “:=” Time

MclkCountMaximum ::= “multiclockcountmax” “:=” <countMaximum>

MclkRateResolution ::= “multiclockrateresolution” “:=” Time

MclkRateMinimum ::= “multiclockratemin” “:=” Time

MclkRateMaximum ::= “multiclockratemax” “:=” Time

MclkLeadingMinimum ::= “multiclockleadingmin” “:=” Time

MclkLeadingMaximum ::= “multiclockleadingmax” “:=” Time

MclkTrailingMinimum ::= “multiclocktrailingmin” “:=” Time

MclkTrailingMaximum ::= “multiclocktrailingmax” “:=” Time

MclkPulseMin ::= “multiclockpulsemin” “:=” Time

MclkConstraints ::= “multiclockconstraint” Subscript “:=” TimeExpr

Time ::= (<intTime> | <floatTime>) [TimeUnit]

TimeUnit ::= (“ps” | “ns” | “us” | “ms” | “s”)

Table 3-13 lists Multiple Clocking Constraints.

Table 3-13. Multiple Clocking Constraints

TCL Parameter	*Data Type*	*Description*
MultiClockType	None, Cyclic, Acyclic	Tester multiclock type
MultiClockConstraint	TimeExpr	Timing constraint for multiclock
MultiClockCountMax	integer	Maximum number of periods in cycle
MultiClockRateMin	time	Multiclock rate (duty cycle) minimum
MultiClockRateMax	time	Multiclock rate (duty cycle) maximum
MultiClockRateResolution	time	Multiclock period (duty cycle) resolution
MultiClockLeadingMin	time	Minimum time between T0 and T1 (to leading clock edge)
MultiClockLeadingMax	time	Maximum time between T0 and T1
MultiClockTrailingMin	time	Minimum time between T0 and T2 (to trailing clock edge)
MultiClockTrailingMax	time	Maximum time between T0 and T2
MultiClockEdgeResolution	time	T1 and T2 edges must be modulo this resolution
MultiClockPulseMin	time	Minimum time for high pulses

Figure 3-6 shows that the T1 and T2 edges must reside on a boundary defined by the **MultiClockEdgeResolution** parameter. The R rate must reside on a boundary defined by the **MulticCockRateResolution** parameter. The pulse specified by T1, T2, and R, is repeated any number of times that is less than the value of the MultiClockCountMax parameter.

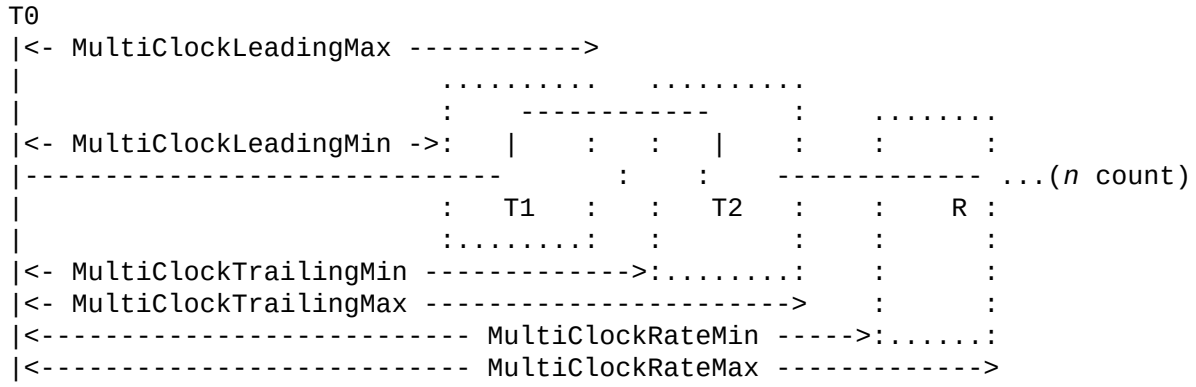

Figure 3-6. Example of Multiclock Constraints

3.6.13 Pattern ATE Controls

The Pattern ATE Controls parameters let you describe your tester's pattern row limitations. Table 3-14 lists Pattern ATE Controls.

Table 3-14. Pattern ATE Controls

TCL Parameter	*Data Type*	*Description*
PatternBoundary	integer	Compression boundary limitation
PatternCompression	YES | NO | THRESHOLD	Compression search starts if pattern rows larger than this
PatternRowMax	integer	Maximum number of pattern rows allowed on tester
BurstRowMax	integer	Maximum number of pattern rows allowed per burst (pattern load)
CompressionMemRowMax	integer	Maximum number of pattern rows allowed in special memory
IORowMax	integer	Maximum number of pattern rows allowed in special memory

Table 3-14. Pattern ATE Controls (continued)

TCL Parameter	*Data Type*	*Description*
MaskRowMax	integer	Maximum number of pattern rows allowed in special memory
HizRowMax	integer	Maximum number of pattern rows allowed in special memory

A complete BNF syntactical representation of the Pattern ATE Controls parameter follows:

Pattern ::= (PatCompression | PatBoundary | PatRowMaximum
| PatBurstMaximum | CompressionMemRowMax
| IORowMax | MaskRowMax | HizRowMask)

PatCompression ::= “patterncompression” “:=” (“yes” | “no”
| “threshold”)

PatBoundary ::= “patternboundary” “:=” <compressionValue>

PatRowMax ::= “patternrowmax” “:=” <patternRowMaximum>

PatBurstMax ::= “burstrowmax” “:=” <burstRowMaximum>

CompressionMemRowMax ::= “compressionmemrowmax” “:=” RowMax

IORowMaximum ::= “iorowmax” “:=” RowMax

MaskRowMaximum ::= “maskrowmax” “:=” RowMax

HizRowMaximum ::= “hizrowmax” “:=” RowMax

RowMax ::= (“patternrowmax” | <rowMaximum>)

The **PatternBoundary** parameter describes the legal limits for compression constructs. Multi-row loops and subroutine bodies may not cross rows that are numeric multiples of this setting. For example:

```
PatternBoundary := 50
```

places boundary limits at pattern row 50, 100, 150, and so on.

The **PatternCompression** parameter specifies whether to allow compression as described by the TCL compression constraints (`YES`), disable compression (`NO`), or compress only

if you require the test program to have a smaller number of pattern rows as is specified in the **PatternRowMax** parameter (THRESHOLD).

The **PatternRowMax** parameter describes the upper limit on tester memory for storing rows of pattern bits. A test program may be partitioned into smaller groups by means of the TCL burst specification described in *Pattern Load Directives* on page 3-103. If there is a tester limit on the size of a burst, the **BurstRowMax** parameter is set to it.

The **PatternCompression** and **PatternRowMax** parameters are required in the tester TCL file. The **BurstRowMax** parameter is only required on those testers that have limitations on a burst's partition size.

The **CompressionMemRowMax** parameter specifies how large (in rows) the special tester memory is that contains compression constructs such as loops and subroutines that may not be legal in the primary tester memory. The user-defined value <rowMaximum> is tester-specific, depending on the size of the memory available on your tester. See the chapter for your WaveBridge for this value.

The **IORowMax**, **HizRowMax**, and **MaskRowMax** parameters let you specify the maximum number of rows for I/O memory, high impedance memory, and mask memory, if these features are supported on your tester. See the chapter for your WaveBridge in the appropriate TDS tester guide to see if your tester supports these features.

The **ScanPatternMin** and **ScanPatternMax** parameters are used in the Pattern Controls parameter syntax. These parameters are discussed in *Scan Controls* on page 3-59.

3.6.14 Timeout Control

The Timeout Control parameters specify the pattern generator timeout value and limits.

The complete TCL BNF syntactical representation for TimeOut, TimeOutMax, and TimeOutMin are as follows.

Start Example

```
TimeOut := TimeValue;
TimeOutMax := TimeValue;
TimeOutMin := TimeValue;
TimeValue := ( <intTime> | <floatTime> ) [ TimeUnit ]
TimeUnit := ( “us” | “ms” | “s” )
```

End Example

The following example shows how to override the default pattern generator timeout value using the TCL Timeout entry.

Start Example

```
Ate
   Timeout := 10S;
End Ate
```

End Example

3.6.15 Pin ATE Controls

The Pin ATE Controls parameters let you define the number of input, output, or bidirectional pins your tester supports.

Table 3-15 lists Pin ATE Controls.

Table 3-15. Pin ATE Controls

TCL Parameter	*Data Type*	*Description*
PinInMax	integer	Forcing pin number range between 1 and integer
PinInOutMax	integer	Pin number range between 1 and integer
PinInOutMin	integer	Pin number range between 0 and integer
PinOutMax	integer	Compare pin number range between 1 and integer

A complete BNF syntactical representation of the Pin ATE Controls parameter follows:

Pin ::= (PinInOutMaximum | PinInOutMinimum | PinInMaximum | PinOutMaximum)

PinInMax ::= “pininmax” “:=” <pinInMaximum>

PinInOutMax ::= “pininoutmax” “:=” <pinInOutMaximum>

PinInOutMin ::= “pininoutmin” “:=” <pinInOutMin>

PinOutMax ::= “pinoutmax” “:=” <pinOutMaximum>

If a tester architecture does not differentiate between input pins and output pins, the **PinInOutMax** parameter is used. **PinInOutMin** can be used to set the minimum number to zero (0), if your tester requires this value.

3.6.16 Probe Constraints

The Probe Constraints parameters are used to control probe windows of board testers.

Probe windows are specialized strobes that are used in conjunction with hand-held probe hardware on board testers. Typically, the probe window requires the use of a dedicated ATE pin. TCL permits control over parts of the cycle containing such a probe window. The following sections of a cycle that contains a probe window can be controlled by the Probe Constraints parameters:

- the minimum time between the leading edge of the probe and the trailing edge of the probe
- the minimum time between the beginning of the cycle and the leading edge of the probe
- the maximum amount of time between the beginning of the cycle and the trailing edge of the probe
- the valid probe ON condition
- the transition time from the valid probe OFF condition to the valid probe ON condition
- the transition time between valid probe ON to valid probe OFF
- the valid probe OFF condition

Table 3-16 lists Probe Constraints.

Table 3-16. Probe Constraints

TCL Parameter	*Data Type*	*Description*
ProbeConstraint	time	Timing constraint expression
ProbeOpenMin	time	Minimum time between T0 and T1
ProbeCloseMax	time	Maximum time between T0 and T2
ProbeWindowMin	time	Minimum time between T1 and T2
ProbeOpenSetup	time	Tolerance prior to T1
ProbeOpenHold	time	Tolerance after T1
ProbeCloseSetup	time	Tolerance prior to T2
ProbeCloseHold	time	Tolerance after T2
SettledProbeOpenDefault	time	Default T1 for settled probe windows
SettledProbeCloseDefault	time	Default T2 for settled probe windows
PulseProbeOpenDefault	time	Default T1 for pulse probe windows
PulseProbeCloseDefault	time	Default T2 for pulse probe windows

A complete BNF syntactical representation of the Probe Constraints parameter follows:

Probe ::= (Probers | Settlers | Pulsers | Monitors | Spikes)

Probers ::= (Probers1 | Probers2)

Probers1 ::= (ProbeConstraints | ProbeOpenMinimum
| ProbeOpenMaximum | ProbeWindowMinimum)

Probers2 ::= (ProbeOpenSetupTime | ProbeOpenHoldTime
| ProbeCloseSetupTime | ProbeCloseHoldTime
| ProbeDefault | ProbeSetup | ProbeHold)

ProbeConstraints ::= “probeconstraint” Subscript “:=” TimeExpr

ProbeDefault ::= “probedefault” “:=” TimeExpr

ProbeSetup ::= “probesetup” “:=” Time

ProbeHold ::= “probehold” “:=” Time

ProbeOpenMinimum ::= “probeopenmin” “:=” Time

ProbeOpenMaximum ::= “probeopenmax” “:=” Time

ProbeWindowMinimum ::= “probewindowmin” “:=” Time

ProbeOpenSetupTime ::= “probeopensetup” “:=” Time

ProbeOpenHoldTime ::= “probeopenhold” “:=” Time

ProbeCloseSetupTime ::= “probeclosesetup” “:=” Time

ProbeCloseHoldTime ::= “probeclosehold” “:=” Time

Settlers ::= (SettledProbeOpenDefTime | SettledProbeCloseDefTime)

SettledProbeOpenDefTime ::= “settledprobeopendefault” “:=” TimeExpr

SettledProbeCloseDefTime ::= “settledprobeclosedefault” “:=” TimeExpr

Pulsers ::= (PulseProbeOpenDefTime | PulseProbeCloseDefTime)

PulseProbeOpenDefTime ::= “pulseprobeopendefault” “:=” TimeExpr

PulseProbeCloseDefTime ::= “pulseprobeclosedefault” “:=” TimeExpr

Spikes ::= (SpikePulseMinimum)

SpikePulseMinimum ::= “spikepulsemin” “:=” Time

Time ::= (<intTime> | <floatTime>) [TimeUnit]

TimeUnit ::= (“ps” | “ns” | “us” | “ms” | “s”)

TCL supports two basic types of probes: edge probes, and window probes. Window probes can be further differentiated as settled probes and pulse probes. A settled probe samples a section of a cycle (“probe window”) that has no transitions within the probe window; a pulse probe samples a section of a cycle in which transitions occur within the probe window. The **Probe**, **SettledProbe**, and **PulseProbe** parameter types are used to control open and close times for the probe edge(s), and to specify guardbands that further limit the section of the probe window from which data is sampled.

The **Probe** parameter type supports edge (as opposed to window) probes. Edge probes sample a signal’s value at only a single specified time, rather than sampling an entire range and evaluating if the signal’s value is stable, as for a window probe.

Figure 3-7 is an example of a cycle with edge probe constraints.

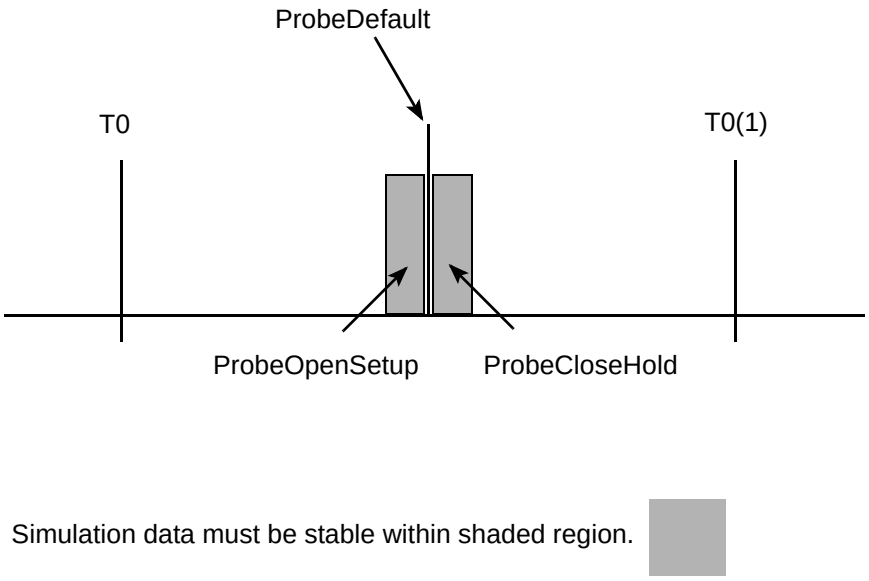

Figure 3-7. Edge probe

The **SettledProbe** parameter type specifies start and stop probe values for testers that support the ability to probe a stable (or “settled”) signal. Settled probes are window probes.

Figure 3-8 is an example of a cycle with settled probe window constraints.

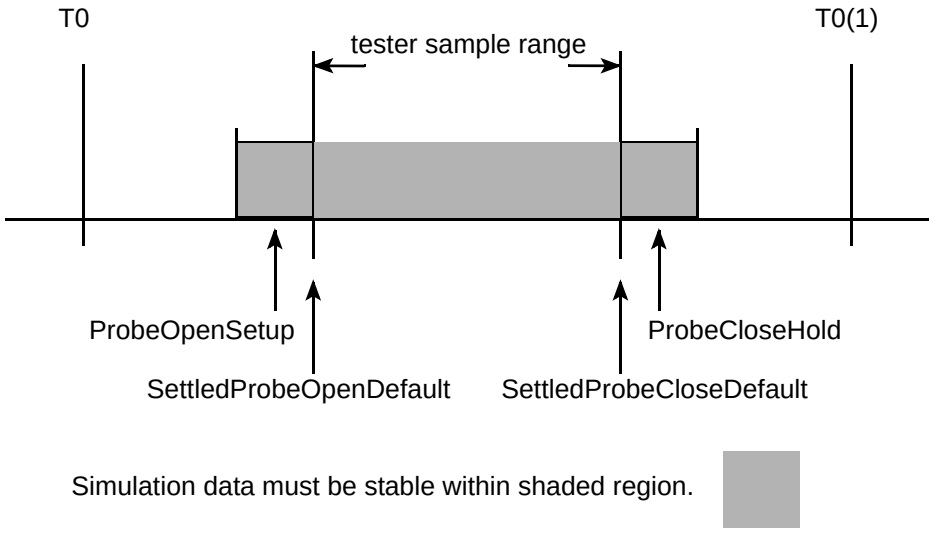

Figure 3-8. Settled probe window parameters

The **PulseProbe** parameter type specifies start and stop probe values and guardband values to limit the region inside the probe window for testers that support the ability to

evaluate signal transitions within the probe window. Pulse probes are window probes. Figure 3-9 is an example of a cycle with pulse probe window constraints.

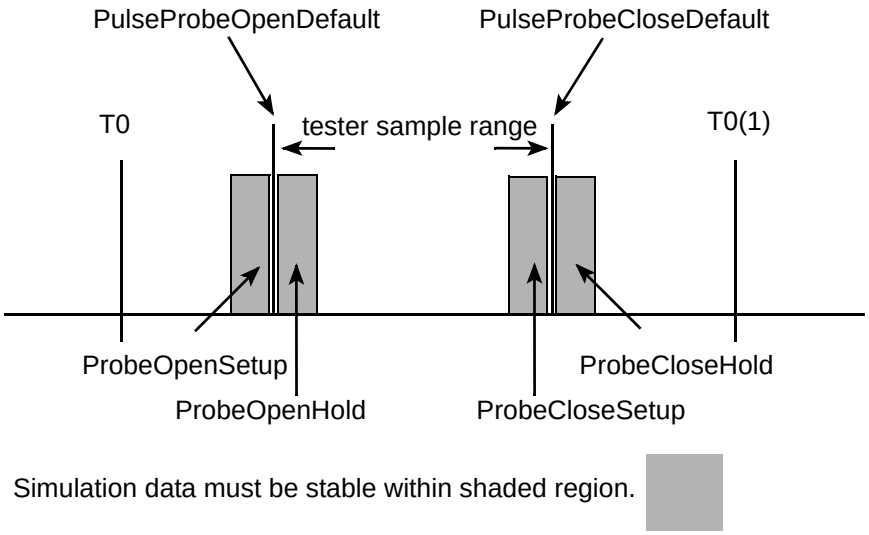

Figure 3-9. Pulse probe window

Open and **Close** modifiers for each of the Probe constraints (as in the **ProbeOpenHold** and **ProbeCloseHold** parameters) denote the opening and closing edges of a probe window. The **Setup** modifier denotes a minimum time prior to an edge. The **Hold** modifier denotes a minimum time after an edge.

The **SpikePulseMinimum** parameter lets you filter out short-duration pulses (spikes) in the simulation data that are too short to have value as a settled probe. The time value specifies the threshold below which a pulse is considered a spike. Cycles that contain spikes can then be masked.

3.6.17 Repeat Constraints

The Repeat Constraints parameter controls the utilization of repeats in your test program, depending on your tester's capabilities. A repeat is a single-row of pattern data or a subroutine call that can have an iteration count.

Below is an example of repeats in a WGL program file.

Start Example

```
waveform repeat1
        signal a : input; end

        pattern load (a)
                repeat 10 vector (+) := [1];
                          vector (+) := [0];
                          vector (+) := [-];
                repeat   2 vector (+) := [0];
                repeat 101 vector (+) := [-];
                          vector (+) := [Z];
                          vector (+) := [X];
                repeat 15 vector (+) := [-];
        end

end
```

End Example

Table 3-17 lists Repeat Constraints.

Table 3-17. Repeat Constraints

TCL Parameter	*Data Type*	*Description*
RepeatCompression	Boolean	Single-row loop compression legal
RepeatAtStartLegal	Boolean	TRUE if single-row loop legal at start of test program
RepeatAtEndLegal	Boolean	TRUE if single-row loop legal at end of test program
RepeatInSubrLegal	Boolean	TRUE if single-row loop legal in subroutine definition
RepeatAtSubrStartLegal	Boolean	TRUE if single-row loop legal at subr def start
RepeatAtSubrEndLegal	Boolean	TRUE if single-row loop is legal in multi-row loop
RepeatInLoopLegal	Boolean	TRUE if single-row loop is legal in multi-row loop
RepeatCountMin	integer	Minimum repeat count for a single-row loop
RepeatCountMax	integer	Maximum repeat count for a single-row loop

A complete BNF syntactical representation of the Repeat Constraints parameter follows:

Repeats ::= (RepeatCompression | RepeatConstraint
| RepeatCountMinimum | RepeatCountMaximum)

RepeatCompression ::= “repeatcompression” “:=” Boolean

RepeatConstraint ::= (RepeatAtStartLegal | RepeatAtEndLegal
| RepeatInSubrLegal | RepeatInLoopLegal
| RepeatAtSubrStartLegal | RepeatAtSubrEndLegal)

RepeatAtStartLegal ::= “repeatatstartlegal” “:=” Boolean

RepeatAtEndLegal ::= “repeatatendlegal” “:=” Boolean

RepeatAtSubrStartLegal ::= “repeatatsubrstartlegal” “:=” Boolean

RepeatAtSubrEndLegal ::= “repeatatsubrendlegal” “:=” Boolean

RepeatInSubrLegal ::= “repeatinsubrlegal” “:=” Boolean

RepeatInLoopLegal ::= “repeatinlooplegal” “:=” Boolean

RepeatCountMinimum ::= “repeatcountmin” “:=” <repeatCountMinimum>

RepeatCountMaximum ::= “repeatcountmax” “:=” <repeatCountMaximum>

If single-row loop compression is not legal on the tester (or not desired), the **RepeatCompression** parameter is set to `false`.

The **RepeatCountMin** and **RepeatCountMax** parameters describe limitations on the iteration count.

The **RepeatInSubrLegal** and **RepeatInLoopLegal** parameters describe the availability of the single-row repeat capability within subroutines and multiple-row loops.

The **RepeatAtStartLegal** and **RepeatAtEndLegal** parameters describe whether it is legal to have a single-row repeat on the first or last row of a test program. The **RepeatAtSubrStartLegal** and **RepeatAtSubrEndLegal** parameters describe the same constraint for the first and last rows of a subroutine definition.

3.6.18 Scan Controls

The Scan Controls parameters let you define the scan cycle length, the resolution of the scan cycle boundaries, the number of scan states, the resolution of the serial pattern memory block, and the scan mode type.

Table 3-18 lists Scan Controls.

Table 3-18. Scan Controls

TCL Parameter	*Data Type*	*Description*
ScanCycleMax	integer	Maximum scan cycle length for a scan row
ScanCycleMin	time	Minimum scan cycle length for a scan row
ScanCycleResolution	time	Scan cycle length timing resolution
ScanPatternMin	integer	Minimum number of scan states per scan run (vector length)
ScanPatternMax	integer	Maximum number of scan states per scan run (vector length)
ScanPinDirection	<type>	Specifies theI/O direction on a scan pin
ScanChannelMax	integer	Maximum number of parallel scan chains available on the tester
ScanPatternResolution	integer	Block size of serial pattern memory for scan data
ScanType	<type>	Scan hardware modes
ScanMode	<type>	Scan vector modes, serial or parallel

A complete BNF syntactical representation of the Scan Controls parameters follows:

Scan ::= (ScanTimes | ScanPatterns | ScanType | ScanMode)

ScanTimes ::= (ScanCycleMax | ScanCycleMin | ScanCycleRes)

ScanCycleMax ::= “scancyclemax” “:=” Time

ScanCycleMin ::= “scancyclemin” “:=” Time

ScanCycleRes ::= “scancycleresolution” “:=” Time

Time ::= (<intTime> | <floatTime>) [TimeUnit]

TimeUnit ::= (“ps” | “ns” | “us” | “ms” | “s”)

ScanPatterns ::= (ScanPatternMin | ScanPatternMax | ScanPatternRes | ScanChannelMax)

ScanPatternMin ::= “scanpatternmin” “:=” <scanPatternMin>

ScanPatternMax ::= “scanpatternmax” “:=” <scanPatternMax>

ScanPatternRes ::= “scanpatternresolution” “:=” <scanResolution>

ScanPinDirection ::= “scanpindirection” “:=” “[“ <dir_list> “];”

<dir list> ::= <direction> { “,” <direction> }

<direction> ::= (“input” | “output” | “bidir”)

ScanChannelMax ::= “scanchannelmax” “:=” <scanChannelMax>

ScanRegistersOnly ::=”scanregistersonly” (“true” | “false”) “;”

ScanType ::= “scantype” “:=” “[” ScanTypeName { “,” ScanTypeName } “]”

ScanTypeName ::= (“in” | “out” | “inout” | “mask” | “inmask” | “feedback”)

ScanMode ::= “scanmode” “:=” (“serial” | “parallel”)

The **ScanPatternMin** parameter specifies the minimum number of states the tester can have on any scan chain.

The **ScanPatternMax** parameter specifies the maximum number of states the tester can have on any scan chain.

The **ScanChannelMax** parameter describes the number of parallel scan chains that are available on the tester-specific scan hardware.

The **ScanPinDirection** parameter specifies the valid I/O direction for a pin on the tester-specific scan hardware. More than one direction can be specified; for example:

```
ScanPinDirection := [ input ];

ScanPinDirection := [ input, bidir ];
```

The **ScanPatternResolution** parameter specifies the block size of the tester’s serial pattern memory. The **ScanPatternResolution** parameter determines how scan states in a scan chain are partitioned in the tester’s memory.

The **ScanCycleMax**, **ScanCycleMin**, and **ScanCycleResolution** parameters apply to the period of the Scan TimePlate that is used with a ScanRow, which is a WDB pattern row that references a scan TimePlate and one or more scan runs.

The **ScanType** parameter specifies the type(s) of scan operations that the tester hardware is capable of performing. Options are: `in, out`, `inout`, `mask, inmask,` and

`feedback`. The **ScanType** parameter is a global setting for all the scan channels. The ScanType options are described as follows:

`inout` means that the scan run is simultaneously shifting serial pattern data into and out of the specified scan channels. The data shifted out is compared against expect data in accordance with the masking data.

`inmask` is the same as `inout` except that the output data is not compared it is all masked, no matter how the masking data is set). This is useful for initializing the scan run (it is equivalent to `inout` with the mask data all on).

`feedback` is the same as `inout` except that the output data is re-shifted into the input channel.

`in` shifts data in, while shifting no data out.

`out` shifts data out and compares it against expect, while shifting no data in.

For example, the ScanType statement:

```
ScanType := [inout, inmask, feedback];
```

specifies that a scan channel can perform three types of operations: `inout`, `imask,` and `feedback`.

The **ScanMode** parameter can be `SERIAL`, or `PARALLEL`. The parameter indicates whether the WaveBridge uses the special-purpose scan hardware on the tester (for `SERIAL` mode), or converts the scan runs into normal parallel vectors (for `PARALLEL` mode).

The **ScanRegistersOnly** parameter (Teradyne Catalyst and A580 only) controls the use of original scan cell names in the output test program as opposed to having the WaveBridge synthesize scan registers. The use of this parameter saves both time and decreases the size of the output test program. **ScanRegistersOnly** directs the WaveBridge to replace scan cell names in the input WDB with synthesized scan registers. Valid values for the **ScanRegistersOnly** parameter are `TRUE` (default) or `FALSE`.

If set to `FALSE` the output test program will use all the original scan cell names found in the input WDB. If set to `TRUE` (or not specified at all) all scan chains will be represented by one (or more) maximum width scan registers and no original scan cell names will be used. Enabling this parameter can typically result in twenty times faster WaveBridge execution and fifty times smaller test programs.

If this feature is not desired as the default behavior the string "`ScanRegistersOnly := FALSE;`" should be added to the stock tester TCL files (or `TRUE` changed to `FALSE` if this value already exists). The following is an example of an entry to an TCL override file that would disable this feature:

Start Example

```
testcontrol scan_registers
ate
   ScanRegistersOnly := FALSE;
end
end
```

End Example

3.6.19 Subroutine Constraints

The Subroutine Constraints parameter controls the utilization of subroutines in your test program, depending on your tester's capabilities. A subroutine is a named collection of pattern rows that can be called from other locations in the test program. There are restrictions on the number of pattern rows contained in a subroutine as well as its placement in the test program.

Below is an example of a subroutine in a WDB, as viewed as a Waveform Generation Language (WGL) program file. For more information on WGL, see Chapter 2: *Waveform Generation Language* in this guide.

Example WGL subroutine:

Start Example

```
waveform subr1
        signal a [0..31]: input hex; end

        pattern load (a)
                call s1();
                call s2();
                call s3();
        end

        subroutine s1 ()
                vector (+) := [01AF00ZZ];
        end
```

```
        subroutine s2 ()
                vector (+) := [01010101];
                vector (+) := [0100Z0Z1];
                vector (+) := [01110101];
                vector (+) := [0111Z0Z0];
        end

        subroutine s3 ()
                vector (+) := [0101Z101];
                vector (+) := [1101Z10Z];
                vector (+) := [0011Z01Z];
                vector (+) := [0111Z1Z0];
                vector (+) := [0101ZZZ0];
        end
end
```

End Example

Table 3-19 lists Subroutine Constraints.

Table 3-19. Subroutine Constraints

TCL Parameter	*Data Type*	*Description*
SubroutineCompression	Boolean	Subroutine compression legal
SubrDefnMaximum	Integer	Maximum number of subroutine definitions allowed
SubroutineNestMax	integer	Maximum subroutine call nesting
SubroutineRowMax	integer	Maximum pattern rows in a subroutine definition
SubroutineRowMin	integer	Minimum pattern rows in a subroutine definition
SubroutineAtStartLegal	Boolean	TRUE if call is legal at start of test program
SubroutineAtEndLegal	Boolean	TRUE if call is legal at end of test program
SubroutineAfterRepeatLegal	Boolean	TRUE if call is legal on single-row repeat
SubrMemoryMax	Integer	Maximum memory allocated for subroutine

Table 3-19. Subroutine Constraints (continued)

TCL Parameter	*Data Type*	*Description*
SubroutineRepeatCountMax	integer	Maximum iteration of single-row repeat with call
SubroutineSpacingMin	integer	Minimum number of rows between call
ScanInSubrLegal	Boolean	TRUE if scan rows in subroutines are allowed
SubroutineInLoopLegal	Boolean	TRUE if it is legal to have subroutine calls inside of loops

A complete BNF syntactical representation of the Subroutine Constraints parameter follows:

Subroutine ::= (SubrCompression | SubrDefnMaximum | SubrNestMaximum | SubrRowMaximum | SubrRowMinimum | SubrRepeatMaximum | SubrMemoryMax | SubrCallConstraint)

SubrCompression ::= “subroutinecompression” “:=” Boolean

SubrDefnMaximum ::=”subroutinedefnmax” “:=” <suberDeferMaxer>

<suberDeferMaxer> :=numeric

SubrNestMaximum ::= “subroutinenestmax” “:=” <subrNestMaximum>

SubrRowMaximum ::= “subroutinerowmax” “:=” <subrRowMaximum>

SubrRowMinimum ::= “subroutinerowmin” “:=” <subrRowMinimum>

SubrRepeatMaximum ::= “subroutinerepeatcountmax” “:=” <subrRepeatMaximum>

SubrCallConstraint ::= (SubrAtStartLegal | SubrAtEndLegal | SubrAfterRepeatLegal | SubrSpacingMinimum)

SubrAtStartLegal ::= “subroutineatstartlegal” “:=” Boolean

SubrAtEndLegal ::= “subroutineatendlegal” “:=” Boolean

SubrAfterRepeatLegal ::= “subroutineafterrepeatlegal” “:=” Boolean

SubrMemoryMax ::= “.subrmemorymax” “:=” numneric

SubrSpacingMinimum ::= “subroutinespacingmin” “:=” <subrSpaceMin>

If subroutine compression is not legal on the tester (or not desired), the **SubroutineCompression** parameter is set to `false`.

The maximum number of subroutines allowed can be defined using the **SubroutineDefnMaximum** parameter.

The minimum and maximum restrictions on the number of pattern rows in a subroutine are described by the **SubroutineRowMin** and **SubroutineRowMax** parameters.

The **SubroutineNestMax** parameter describes to what level internal subroutine calls may be nested during execution of the test program. The default for this parameter is zero (`0`), no nesting allowed.

The **SubroutineAtEndLegal** and **SubroutineAtStartLegal** parameters describe the constraints on a subroutine call being located at the beginning or at the end of the test program's pattern rows.

The **SubroutineAfterRepeatLegal** parameter describes whether it is legal to have a subroutine call immediately following a pattern row that is also a single-row repeat.

The **SubroutineMemoryMax** parameter specifies the maximum amount of memory allocated for a subroutine. This parameter is used in cases where a specific ATE has sunroutine memory restrictions.

The **SubroutineRepeatCountMax** parameter describes the maximum count allowed on a single-row repeat that is also a subroutine invocation.

The **SubroutineSpacingMin** parameter describes the minimum number of rows that are necessary between adjacent subroutine calls. If the value for this parameter is zero (`0`), adjacent subroutine calls are allowed.

The **ScanInSubrLegal** parameter indicates whether it is legal to have scan rows in subroutines.

The **SubroutineInLoopLegal** parameter indicates whether if it is legal to have subroutine calls inside of loops.

3.6.20 TimePlate Matching Preference Control

NOTE

This TCL parameter applies only to the TimePlate Match Conditioner and to WaveBridges that provide TimePlate Match capabilities. It does not apply to the SequenceMatch Conditioner.

A single TCL parameter allows you to establish an order of preference when more than one TimePlate matches a given segment of events. This TCL parameter is named:

TilerTimePlateOrderCost

The default TimePlate matching strategy uses a goodness-of-fit approach. In this appoach, the TimePlates are evaluated against the current segment of the event stream. Based on the evaluation, a weight is assigned to each matching TimePlate that reflects how closely it matches the events. Then, the assigned weights are reviewed and the TimePlate with the highest weighting is matched with the events. For more information about the TimePlate Match Conditioner's internal process, see Section 5.5.1 of the *WDB Conditioners Guide*.

In the case where two or more TimePlates have the same weight, the default is to match the TimePlate that occurs first in the WDB. (The TimePlate order in the WDB is determined by the source; either a WGL file or the WaveMaker Timing Editor.) You can use the **TilerTimePlateOrderCost** parameter to establish a different order.

When using this parameter, TimePlates are assigned an integer that is determined by the TimePlate order in the WDB. The first entry is assigned 0, the next is 1, then 2, and so on. The **TilerTimePlateOrderCost** parameter is used as a multiplier to create a weighting. The new weighting for each TimePlate is created by subtracting the TimePlate order value from the total number of TimePlates and then multiplying the difference by the **TilerTimePlateOrderCost** parameter value.

The default value of 0 for this parameter results in the default goodness-of-fit matching strategy.

Because the goodness-of-fit rankings are not user-visible, a thorough understanding of the TimePlates is necessary to make decisions about the TimePlate entry order when using the **TilerTimePlateOrderCost** parameter.

The following example shows the **TilerTimePlateOrderCost** value set to ten:

Start Example

```
testcontrol example7
   ate
      ForceMatchResolution := 0ns;
      CompareMatchResolution := 0ns;
      DriveMatchResolution := 0ns;
      CycleMatchResolution := 0ns;
      TilerTimePlateOrderCost := 10;
   end ate
```

End Example

3.6.21 Timeset Controls

The Timeset Controls parameters let you control the timesets created in the output test program.

A complete BNF syntactical representation of the Timeset Controls parameter follows:

TimeSet ::= “timesetmax” “:=” <timesetMaximum>

TimeSetType ::= “timesettype” “:=” (“allow_both” | “force_single” | “force_dual”)

LocalTimeSet ::= “localtimesetmax” “:=” <localtimesetMaximum>

The time values for the timing generators are organized into timing sets for each WDB TimePlate. The number of timing sets available is controlled by the **TimeSetMax** TCL parameter. See the WaveBridge chapter for your tester for the value for this parameter.

The **TimeSetType** parameter allows you to override tester-specific defaults that specify that more than one timeset is used to create certain types of strobes. See the chapter for your WaveBridge in the appropriate TDS tester guide to see if this feature is supported for your tester.

If your tester supports the definition of local timesets, the **LocalTimeSetMax** parameter lets you control the number of these timesets locally, in the same manner that the **TimeSetMax** parameter controls the number of timesets globally.

3.6.22 Timing Expressions

Whenever a time value or range is used on the right-hand side of a tester parameter assignment, a TCL Timing Expression can be written. In most cases, the tester parameters

are assigned very simple expressions with time values (like 200nS). When it is necessary to derive a tester parameter setting from some other tester parameter, the tester parameter can be used in the timing expression. Table 3-20 shows Timing Expression operator precedence.

Table 3-20. Timing Expression Operator precedence

Precedence	*Operator*
1 (highest)	= max min select
2	* / % .. and
3	binary + - or
4	unary -
5 (lowest)	= < > > = < = > <

Sub-expressions can be enclosed in parentheses to override the natural precedence of a time expression.

The relation operators compare time values and return a Boolean result. Table 3-21 shows timing relation operators and their results.

Table 3-21. Timing relation operators and results

Operation	*Result*	*Operand1*	*Operand2*
RelOp	Boolean	time	time
AddOp	time	time	time
MulOp	time	time	time
	time	integer	time
	time	time	integer
	integer	integer	integer
Max	time	time	time
Min	time	time	time

A complete BNF representation of the Timing Expressions parameter follows:

TimeExpr ::= SimpleExpr [Relop SimpleExpr]

Relop ::= (“=” | “<>” | “>=” | “<=” | “>” | “<”)

SimpleExpr ::= [“-”] Term { AddOp Term }

AddOp ::= (“+” | “-” | “or”)

Term ::= Factor { MulOp Factor }

MulOp ::= (“*” | “/” | “%” | “..” | “and”)

Factor ::= (“(” TimeExpr “)” | SubExprList | BuiltIn | Time
| SimpleOperand)

SubExprList ::= “[” TimeExpr { “,” TimeExpr } “]”

BuiltIn ::= (“max” | “min” | “select”) “(” [TimeExpr { “,” TimeExpr }] “)”

SimpleOperand := <ATEparamOrTedge> [“(” <cycleNumber> “)”]
[“’” “format”]

Time ::= (<intTime> | <floatTime>) [TimeUnit]

TimeUnit ::= (“ps” | “ns” | “us” | “ms” | “s”)

The multiplication operators take as operands any combination of integer or time values and return an integer value only if both operands are of integer type. In all other cases, a time value is returned.

The `%` operator performs a modulus operation on time values.

The `..` operator performs a range operation on time values that is defined by the absolute difference between the two operands that you use. Thus, the order in which you enter the operands is not important, since the resulting time value is the same in either case. Both operands must contain a time value for the range operation.

A time value is formed from an integer or floating point number followed by a unit of time measurement. An <ATEparamOrTedge> is the name of any tester parameter that is a time value, or an integer, or a timing edge.

A timing edge is denoted as `T0`, `T1`, and so on, with an optional subscript containing a cycle number. `T0` refers to the start of a cycle. Any edges that are relative to `T0` are numbered sequentially `T1`, `T2`, etc. If subsequent cycles are referred to, a cycle subscript is added. `T0(1)` refers to the end of cycle `T0` (and is the start of the next cycle). `T1(1)` refers to the next cycle’s `T1` edge.

An optional attribute can be specified following a timing edge. The format attribute of the timing edge is the tester format associated with the edge in the TimePlate. (A TimePlate is

a data storage construct contained within a WDB. The TimePlate carries data that determines the shape and timing of each signal within the cycle.) Thus, `T0'` format and `T1'` format mean the same thing. Attributes are used to "guard" timing constraint expressions.

The following example shows guarded timing expressions, as they would appear in a Tester file or the ATE Constraints block of a TCL file.

```
ForceConstraint[1] := T1..T1(1) >= 32ns;
ForceConstraint[2] := T1..T2 >= 20ns;
ForceConstraint[3] := (T1'format = NR) and (T1..T2 >=
24ns);
```

The first two constraints are always in effect. The third constraint is in effect only when the edges belong to a non-return format (`NR` being the tester-specific name of the format, in this case).

Timing expressions that cannot be evaluated because of having false guards or having undefined edges are ignored. TDS WaveBridge modules do not allocate tester resources for them, nor does the WaveBridge's Tester Rules Checker check them.

3.6.23 Transform

If the WaveBridge for your tester does not support timing waveforms that contain tri-state logic (represented by the Z TDS state character) in the program source WDB, or if the WaveBridge does not support tri-state pattern data on input pins, the Transform parameter lets you specify automatic timing and pattern transformation.

A complete BNF syntactical representation of the Transform parameter follows:

Transform ::= (TransformPattern | TransformTiming)

TransformPattern ::= "transformpattern" ":=" Boolean

TransformTiming ::= "transformtiming" ":=" Boolean

Example of the **Transform** parameters in TCL file:

Start Example

```
ate
   .   .   .
   TransformTiming := TRUE;
   TransformPattern := TRUE;
   .   .   .
end ate
```

End Example

3.6.23.1 Timing Transformations

If the **TransformTiming** TCL parameter is set to TRUE and the format of an input timing waveform is not supported by WaveBridge, automatic timing transformation is applied. The WaveBridge performs one of several state transformations to find a supported shape, and it retains the resulting transformation only if the shape is supported. All transformations are noted in the TRC Report file and the WDB is annotated to show that the timing waveform has been modified using the **TransformTiming** parameter.

If the **TransformTiming** TCL parameter is set to FALSE, no automatic timing transformation is done, but the timing waveforms are analyzed and if the formats are not supported, the unsupported waveforms are noted in the TRC Report file. If the TCL file does not include the **TransformTiming** TCL parameter, the waveform analysis is suppressed.

The WaveBridge performs the transformations in the following order. When the WaveBridge finds a match, the transformation is noted in the TRC report and the process is stopped.

1. **Z to X**: The WaveBridge replaces forcing Z states with expect X states. This might change the signal direction from input to bidirectional.
2. **X to Z**: If a signal is *not* bidirectional, the WaveBridge transforms expect X states to drive Z.
3. **X to XQX**: If a signal has bidirectional tracks, the WaveBridge replaces X states with XQX.
4. **C to S**: The WaveBridge replaces C states with S states (causing a pattern complement). The WaveBridge never transforms the surround-by-complement shapes CSC, CS, SC, and SCS.

5. **S to C**: The WaveBridge replaces S states with C states (causing a pattern complement). The WaveBridge never transforms the surround-by-complement shapes CSC, CS, SC, and SCS.

3.6.23.2 Pattern Transformations

If the **TransformPattern** TCL parameter is set to TRUE, automatic pattern transformation is applied. This is accomplished by turning off the driver for the entire cycle in which a tri-state pattern (Z pattern character) is present. All transformations are noted in the TRC Report file.

If the **TransformPattern** TCL parameter is set to FALSE, Z pattern bits are retained. This assumes that the tester is capable of using a Z pattern bit on an input pin to inhibit the driver for the cycle in which the Z pattern bit occurs.

If the TCL file does not contain the **TransformPattern** TCL parameter, no automatic pattern transformation or analysis is done. The Tester Rules Checker (TRC) and code generation phases of the WaveBridge run detect the unsupported pattern bit, and the occurrence is noted in the TRC Report file.

No transformation is performed on any patterns associated with a signal defined as REFERENCE in the TDS Pin Assignment file. For details about REFERENCE signals, see Chapter 4: *User-Defined Files* in this guide.

If the signal is an input signal and has a Z or X pattern bit, the signal is changed to a bidirectional signal, with the output track in the TimePlate containing only an X state character. The pattern bits associated with this output track in the TimePlate are changed to X pattern bits.

If the signal is a bidirectional signal and the input part of the bidirectional track has a Z or X pattern bit associated with it, a new output track in the TimePlate is created that is associated with the pattern bits for the bidirectional signal. If possible, the output part of the bidirectional track is copied to the newly created output track.

NOTE

If the timing waveform created by the transform operation would also be unsupported, no transform operation is performed for that waveform.

A message noting each transformation is retained in the TRC Report file. Below is an example of these messages.

Example Fragment of TRC Report file containing Transform messages:

Start Example

```
================== Waveform Transformations ==================
Database: dest.wdb
Inform: Signal sig1 was transformed as follows:
        Transforming Z to X in pattern for sig1 (1 times).
        Changing signal sig1 to bidirectional.
        Adding X pattern for signal sig1 (5 times).
        Transforming Z to X in timing on sig1.
        Created          sig1 := output[X]     in timeplate tp1

Inform: Signal sig2 was transformed as follows:
        Transforming Z to X in timing on sig2.
        Changed to bidir sig2 := bidir[Z => X] in timeplate tp1

Inform: Transforming Z to X on signal sig1 (1 time).
        Created      sig1 := output[X    ] in timeplate tp1

Inform: Transforming Z to X on signal sig2 (10 times).
        Created      sig2 := output[X    ] in timeplate tp1
        Copied bidir sig2 := output[X Q X] in timeplate tp3
        Copied bidir sig2 := output[X Q X] in timeplate tp4
```

End Example

3.7 Pin Groups

The Pin Groups block allows the configuration of test programs based upon special tester hardware configurations. If a tester is configured such that groups of pins have different capabilities, sets of functionally equivalent pins can be combined into a “pin group.”

The syntax of the Pin Groups block is:

pingroup *<PinGroupName>*
PinGroupBody
end pingroup *[<PinGroupName>]*

Example of a Pin Groups block, in the context of a partial TCL file:

Start Example

```
testcontrol pingroup_example

   ate
      CycleMin := 50ns;
      CycleMax := 5us;
      ForceConstraint[1] := T1..T2 >= 50ns;
      ForceConstraint[2] := T2..T3 >= 50ns;
   end ate

   pingroup a
      pins := [1,2,4..10,12];
      CycleMin := 100ns;
      CycleMax := 1us;
      ForceConstraint[1] := T1..T1(1) >= 100ns; # these replace
                            # those above
      ForceConstraint[2] := T2..T2(1) >= 100ns;
   end pingroup a

   pingroup b
      cardtype := D1;
      cardslots:= [1,4..10];
   end pingroup b

   pingroup c
      pins := [1..10];
      CycleMax := 2us;
   end pingroup c

   pingroup d
      cardtype := D1;
      cardslots:= [2];
      pins := [100,101];
      CycleMax := 3us;
   end pingroup d
   .  .  .
   .  .  .
   .  .  .

end testcontrol
```

End Example

A complete BNF syntactical representation of the Pin Groups block follows:

```
PinGroups ::= { “pingroup” <PingroupName>
                PinGroupBody
        “end” “pingroup” [ <PingroupName> ] }

PinGroupName ::= ( <name> | <nameString> )

PinGroupBody ::= CardInfo Pins { AteConstraint “;” }

CardInfo ::= [ CardType] [ CardSlots ]

CardType ::= “cardtype” “:=” <cardTypeName> “;”

CardSlots ::= “cardslots” “:=” “[” Slot { ”,” Slot } “]” “;”

Slot ::= <slotNumber> [ “..” <slotNumber> ]

Pins ::= { “pins” “:=” PinList }

PinList ::= “[” PinNum { “,” PinNum } “]” “;”

PinNum ::= <pinNumber> [ “..” <pinNumber> ]
```

Some testers allow you to have several different pin cards installed in different slots in the tester. These different cards can each have different ATE constraints. The Pin Groups block allows you to specify different ATE constraints on a per-card basis.

The <PinGroupName> is assigned to the channel card hardware on the tester. Each pin group specifies zero or more <pinNumber>s that are assigned to the pin group tester resource. Tester capabilities that differ from those already in effect by default for all pins are specified in **AteConstraint** syntax.

<name> and <nameString> are user-defined identifiers or strings used to identify an individual Pin Groups block.

The **cardType** parameter lets you specify the type of pin card using the <cardTypeName> identifier.

The **cardSlots** parameter lets you use the <slotNumber> identifier to specify which slots on the test equipment contain the pin cards that you specify with the **cardType** parameter. WaveBridge uses this information during the tester rules check and resource allocation phases of the WaveBridge run.

The pin group inherits the tester constraints in the default Tester file (*<your_tester_name>*`.tcl`) for the tester, which can be found in the directory set by the TDSDIR environment variable. Any tester constraints specified in your TCL file in the

ATE Constraints block override these defaults, and in turn become inherited by the pin groups. Any tester constraints specified in your Pin Groups block override those inherited by the same name.

3.8 Message Overrides

The Message Overrides block lets you customize the TRC messages that WaveBridge generates during a TRC run. Each message override has a name, a parameter list, and a message body.

The syntax of the TCL Message Overrides block is:

message *<messageName> MessageParams*
MessageBody
end message *[<messageName>]*

Example of a TCL Message Overrides block:

Start Example

```
message ForceRes
   (tp, chan, sig:integer; edge, res, lowGood, highGood:time;
   tester : string)

   severity fatal;
   terse “Bad force edge resolution”;
   cause “A force edge at time @edge does not”
         +“reside at the required resolution of”
         +“@res for the @tester tester.”;
   location “Problem occurred in timeplate @tp,”
         +“channel @chan, signal @sig, at time”
         +“@edge.”;
   remedy   “Snap the edge at @edge to a time that”
         +“resides on the tester’s force resolution”
         +“of @res. The surrounding legal edge times”
         +“are @lowGood < @edge < @highGood.”;
end message
```

End Example

A complete BNF syntactical representation of the Message Overrides block follows:

MessageOverrides ::= { “message” <messageName> MessageParams
MessageBody
“end” “message” [<messageName>] }

MessageParams ::= [“(” <parameterName> { “,” <parameterName> } “:”
ParamType “)”]

ParamType ::= (“signal” | “TimePlate” | “list” | “time” | “integer”
| “string”)

MessageBody ::= { (Severity | Repetition | MessageText) “;” }

Severity ::= “severity” (“informative” | “warning” | “fatal”)

Repetition ::= “repetition” “:=” <repetitionCount>

MessageText ::= (“terse” | “cause” | “location” | “remedy” | “generic”) “:=”
MessageString

MessageString ::= ‘ “ ’ <string> ‘ ” ’ { “+” ‘ “ ’ <string> ‘ ” ’ }

A <messageName> must refer to a TRC function. The TRC messages are described in the TRC Directives block of the TCL file.

The <parameterName> describes information that is passed to the message when TRC is generating its report. The actual name serving as a parameter name can be anything; however, the actual information provided is associated with the parameter’s position in the parameter list.

A message string may contain <parameterName> references from its parameter list. The actual values of these parameters are substituted into the message when the message is formatted. The **MessageParams** parameter may be absent if only the **repetition** or **severity** reserved words are contained in the message block.

Each message has an associated level of severity, represented by the **severity** reserved word.

Each message has an associated <repetitionCount> that describes the number of times the message is issued during a TRC run. If it is absent, the default is infinite. Messages can be inhibited by setting the repetition count to zero (`0`).

The **terse** reserved word specifies the message text used in summary reports. It must not contain any parameter references. It must be 25 characters or less in length.

The **cause** reserved word specifies the message text used to describe the reason behind the tester incompatibility that TRC discovered.

The **location** reserved word specifies the message text used to describe the location of the tester incompatibility. There are many kinds of information stored in WDB. TRC may find tester incompatibilities in a number of places, and the location message pinpoints the problem.

The **remedy** reserved word specifies message text that is a helpful suggestion on how to repair the tester incompatibility that TRC discovered. Since the process of creating a test program from design or simulation data is complex, many avenues exist for repair. The **remedy** message attempts to enumerate the possible actions and TDS tools that are useful to effect repair.

The **generic** reserved word specifies message text used to summarize any of the above message components without referring to any of the formal parameters. It is used to explain in one place each type of TRC problem reported, but it is issued only once.

NOTE

The default WaveBridge TRC messages are described in Chapter 9 of the Tester Bridges Overview Guide.

TCL produces the following messages identifying syntactic errors associated with this block:

```
Severity already set in this message override.

Repetition already set in this message override.

Repetition count must be >= 0.

Terse string already defined for this message override.

Cause string already defined for this message override.

Location string already defined for this message override.

Remedy string already defined for this message override.

Generic string already defined for this message override.

Terse string must be 25 characters or less.
```

NOTE

Not all TRC messages can be edited; you can edit only those TRC messages listed in Chapter 9 of the Tester Bridges Overview Guide.

3.9 TRC Directives

The TRC Directives block controls how the WaveBridge performs tester rule checking. This block also controls the format of the TRC reports.

The syntax of the TCL TRC Directives block is:

```
trc
{TrcSpec}
end trc
```

Example of a TRC Directives block:

Start Example

```
trc
   format := linear;
   formwidth := 80;
   repetition := 5;
   ignore := informative;
   quiton := TimeSetLimit;
end trc
```

End Example

The TRC setting produces a linear description of tester incompatibilities without the informative messages. If a force recovery timing set limit violation is found, the checking is aborted. Checking occurs for only the WDB pattern violations.

A complete BNF syntactical representation of the TRC Directives block follows:

TrcDirectives ::= [“trc”
{ TrcSpec “;” }
“end” “trc”]

TrcSpec ::= (ReportFormat | ReportWidth | GlobalRepetition
| IgnoreWhen | QuitOn)

ReportFormat ::= “format” “:=” (“tabular” | “linear” | “both”)

ReportWidth ::= “formwidth” “:=” (“80” | “132”)

GlobalRepetition ::= “repetition” “:=” <integer>

IgnoreWhen ::= “ignore” “:=” (“informative” | “warning”)

QuitOn ::= “quiton” “:=” <messageName> [“after” <integer>]

The **format** reserved word allows the selection of two kinds of report styles: tabular and linear.

The **tabular** reserved word specifies a format that is a concise table that summarizes all of the TRC messages. Following the table is a generic description of each kind of error referenced in the report.

The **linear** reserved word specifies a format that contains a description of each TRC message as it occurs during the checking. Embedded in the description is WDB information that is specific to the check (signal names, cycle numbers, edge times, and so on). Since this style of report may be quite long, each message can have its repetition controlled. The repetition of messages is described in*Message Overrides* on page 3-77.

The **repetition** reserved word allows setting a global repetition count for all TRC functions. The message override settings for a check take precedence over this global setting. The default setting is 5.

The **ignore** reserved word allows suppression of informative messages or both informative messages and warnings.

The **quiton** reserved word specifies a setting that allows the setting of triggers that abort the checking. A <messageName> must be the name of a TRC function and, if the check fails, TRC terminates. For a list of the function names that you can specify, refer to Section 7.4.2.4 in the *Utilities and Tools Guide*.

The optional **after** reserved word allows termination control after the specified number of occurrences of the check reporting a failure. Multiple occurrences of **quiton** can be set.

A database can be thought of as having two parts for design waveforms and tester resource allocation. The optional **design** or **test** reserved words allow selection of what part of the database to check. If omitted, both sections are checked. The remaining reserved words describe the individual sections in the database that are checked. By default, TRC checks both sections.

3.10 Match Directives

The Match Directives block controls TimePlate and sequence matching operations, which are available through the SequenceMatch and TimePlate Match Conditioners, as well as some WaveBridge modules.

Note that, to achieve optimal results, it is recommended that you perform TimePlate match operations outside of your WaveBridge runs, using either the SequenceMatch or TimePlate Match Conditioners.

NOTE

The SequenceMatch and TimePlate Match Conditioners respond to a slightly different set of TCL parameters. The SequenceMatch Conditioner responds to all Match Directives block parameters, but the TimePlate Match Conditioner does not. The differences are pointed out where they occur. For general information about these conditioners, refer to Chapter 2 and Chapter 5 in the WDB Conditioners Guide.

The process of TimePlate matching consists of matching segments of a stream of waveform data (events) with the timing structure provided by TimePlates. As matches occur, the segment of the event stream and the matching TimePlate are combined to form structured waveform data in WDB format. In the SequenceMatch Conditioner, you can also specify that the event stream match a sequence of TimePlates.

The source of the input events (called the wave source) can be an SEF or WDB. The source of the TimePlates (called the timing source) is a WDB.

You can use the Match Directives block (or Match block) to name the wave source, timing source, and destination WDB. You can also limit the matching to a specific region in the wave source or a specific set of TimePlates in the timing source. In the SequenceMatch Conditioner, you can specify match weights for TimePlates and sequences to effect which one gets ultimately chosen if more than one matches. You can use as many Match Directives blocks as needed, thus allowing a multitude of source data matches and output destinations.

If the structured waveform data already exists (if it was created from the WaveMaker package or WGL), the process of TimePlate matching is not required, and the Match Directives block should not be used in your TCL file.

For general information about the SequenceMatch Conditioner’s matching process, see Section 2.5 of the *WDB Conditioners Guide*.

For general information about the TimePlate Match Conditioner's matching process, see Section 5.5.1 of the *WDB Conditioners Guide*.

The syntax of the TCL Match Directives block is:

match
MatchBody
end match

Example of a TCL Match Directives block:

Start Example

```
match
   events
      directory := wave_source;
      start := begin;
      stop := end;
   end events
   timing
      directory := timing_source;
      timeplates := ts1, ts2, ts3;
      persistence := 1;
   end timing
   destination
      directory := wave_dest;
      start := prevstop;
   end destination
end match
```

End Example

A complete BNF syntactical representation of the Match Directives block follows. Note that underlined items apply only to the SequenceMatch Conditioner:

MatchDirectives ::= { "match"
MatchBody
"end" "match" }

MatchBody ::= { MatchSpecs ";" }

MatchSpecs ::= (EventBlock | TimingBlock | DestinationBlock |
SequencesBlock)

EventBlock := “events”
{ EventSpec }
“end” “events”

EventSpec ::= (WaveSource | StartTime | StopTime)

WaveSource ::= “directory” “:=” <directoryname> “;”

StartTime ::= “start” “:=” TimeSpec “;”

StopTime ::= “stop” “:=” TimeSpec “;”

TimingBlock ::= “timing”
{ TimingSpec }
“end” “timing”

TimingSpec ::= (TimingSource | Timeplates | Persistence | MatchType)

TimingSource ::= “directory” “:=” <WDBname> “;”

Timeplates ::= “timeplates” “:=” Timeplate { “,” Timeplate } “;”

Timeplate ::= TimeplateName [“ $” <weight>] [“<=” TimeplateName]

TimeplateName ::= (<timeplateName> | <timeplateString>)

Persistence ::= “persistence” “:=” <persistenceValue> “;”

MatchType ::= “matchtype” “:=” MatchTypeValue “;”

MatchTypeValue ::= “match_normal” | “match_exact” | “match _qnox”
“match_stableq” | “match_nomiss”

DestinationBlock ::= “destination”
{ DestSpec }
“end” “destination”

DestSpec ::= (WaveDestination | DestTime)

WaveDestination ::= “directory” “:=” <WDBname> “;”

DestTime ::= (“start” | “stop”) “:=” TimeSpec “;”

TimeSpec ::= (“begin” | “end” | “prevstop” | Time)

Time ::= (<intTime> | <floatTime>) [TimeUnit]

TimeUnit ::= (“ps” | “ns” | “us” | “ms” | “s”)

SequencesBlock ::= “sequences”
{ SequenceDef }
“end” “sequences”

SequenceDef ::= SeqType <SequenceName> “:=” SeqBody “;”

SeqType ::= “seq” | “subseq”

SeqBody ::= SeqItem | “(“ SeqBody “)” |
SeqBody “+” SeqBody | SeqBody “|” SeqBody

SeqItem ::= SequenceName [SeqMinMax] [SeqCost] [SeqSubstitution]

SeqMinMax ::= “(“ <min> [“...” <max>] “)”

SeqCost ::= “ $” <weight>

SeqSubstitution ::= “<=” TimeplateName

SequenceName ::= (<sequenceName> | <sequenceString>)

Remember that the underlined elements in this BNF description apply only to the SequenceMatch Conditioner and not to the TimePlate Match Conditioner or WaveBridges that allow TimePlate matching.

NOTE

*A **TimeplateName** or **SequenceName** that is the same as a TCL reserved word can be specified by enclosing the TimeplateName in double quotation marks (“ ”). It is good practice to enclse all TimePlate and sequence names in quotes.*

Each Match block is specified by the **match** and **end match** reserved words. Your TCL file can contain multiple Match blocks. Each Match block contains one Events block, which defines the wave source, one Timing block, which defines the timing source, and one Destination block, which defines the destination WDB. A SequencesBlock can be specified for SequenceMatch Conditioner operations, but it is optional.

An Events block is specified by the **events** and **end events** reserved words. The Events block must contain one **directory** statement, which identifies the SEF database or WDB that is to provide the input events:

```
directory := <wave source directory>;
```

Optionally, you can include **start** and **stop** statements to specify the times where the wave source matching process is to occur. If **start** and **stop** are absent, they default to `begin` and `end`, respectively. If you are using multiple Match blocks, you can set **start** to `prevstop` to continue the matching process with the next sequential segment of the wave source. It is legal for the start/stop times to overlap, but be aware that the last Match block determines the match behavior of the overlapping area.

A TimingBlock is specified by the **timing** and **end timing** reserved words. The TimingBlock must contain one **directory** statement, which identifies the WDB that is to provide the input TimePlates:

```
directory := <timing source WDB>;
```

Optionally, you can include a **timeplates** statement in the TimingBlock; however, a **timeplates** statement cannot be included if you are using a SequencesBlock. The **timeplates** statement has two forms. One form establishes the set and order of TimePlates to use for matching, as shown in the following example:

Start Example

```
timing
   directory := timing_source;
   timeplates := ts1, ts2, ts3;
   persistence := 1;
end timing
```

End Example

This example specifies that only the TimePlates named `ts1`, `ts2`, and `ts3` will be used for matching. When more than one TimePlate matches the input wave source with equal weight, the TimePlate that appears first in the list is chosen. In this example, the preference is to choose `ts1` over `ts2` and `ts3`, and to choose `ts2` over `ts3`.

When using the SequenceMatch Conditioner, you can optionally specify TimePlate weights. The specified weight is added to the weight that is computed by the SequenceMatch Conditioner. (For information about how SequenceMatch computes weights, refer to Section 2.5.2 in the *WDB Conditioners Guide*.) In the following example, TimePlate `ts2` has a value of 10 added to its computed weight, while `ts3` has a value of 20 added to its computed weight.

Start Example

```
timing
   directory := timing_source;
   timeplates := ts1, ts2 '$10, ts3 '$20;
   persistence := 1;
end timing
```

End Example

This technique impacts the SequenceMatch Conditioner to choose `ts3` over `ts2` and `ts2` over `ts1`, unless the other TimePlate matches significantly better, as decided by the computed weight.

The other form of the **timeplates** statement associates MatchPlates with SpecPlates, which allows you to use one TimePlate (the MatchPlate) for the matching process, and when a match occurs, to substitute a different TimePlate (the SpecPlate) to be written to the destination WDB. The following example shows the use of this syntax to associate `tp1` with `devcycle1` and `tp2` with `devcycle2`.

Start Example

```
timing
   directory := "unit.wdb";
   persistence := 1;
   timeplates := tp1 <= devcycle1, tp2 <= devcycle2;
end timing
```

End Example

In this example, `tp1` and `tp2` are MatchPlates and `devcycle1` and `devcycle2` are SpecPlates. When `tp1` or `tp2` matches a segment of the event stream, `devcycle1` or `devcycle2` are written to the destination WDB. The SpecPlate does not need to be the same length as the MatchPlate. The SpecPlate substitution does not occur until after a TimePlate has been chosen for output. Note that a time comment is added at the end of each vector to reflect the traversal of the wave source. To see the time comment, you must convert the WDB to a WGL file.

The MatchPlate/SpecPlate capability allows you to use a general TimePlate for matching and then substitute a more exact TimePlate when a match occurs, which is useful if the original SEF database does not contain accurate timing (from a zero or unit delay simulation) and real device timing is known. It is also useful for matching events according to one cycle period and then switching to a different cycle period.

Optionally, you can include a **persistence** statement to specify how many match failures are allowed before the matching operation halts. The default value is one (`1`), which causes the matching to halt after one match failure. If you set it to a higher value and a match failure occurs, the behavior is to skip ahead until a match is found. The maximum amount of time skipped is (persistence - 1) times the maximum TimePlate period. If a match is subsequently found, an empty cycle is written to the destination WDB. Empty cycles must be replaced by valid data before the destination WDB can be used as input to a WaveBridge.

For the SequenceMatch Conditioner, you can optionally include a **matchtype** statement to specify the matching algorithm. The algorithms are described on page 2-17 of the *WDB Conditioners Guide*.

A Destination block is specified by the **destination** and **end destination** reserved words. The Destination block must contain one **directory** statement that identifies the destination WDB:

```
directory := <destination WDB>;
```

The destination WDB can be the same as the timing source WDB. When this is the case, the structured waveform data is stored in a temporary location until the full match process completes error-free, and then it is written to the specified destination WDB. The timing source WDB is overwritten only when the match process is successful.

Optionally, you can include a **start** statement to position where in the destination WDB the matched data is written. You can specify two kinds of start values: either the time in the destination WDB at which you want to begin writing, or the string `prevstop`, which means that you want to append data to the destination WDB. If you specify a time value, it must be less than or equal to the end of the destination WDB.

If you want to create a single WDB from multiple wave sources, you can set the **start** value to `prevstop`. In this case, the Match blocks in the TCL file are processed sequentially, and each match is appended to the end of destination.

The user-defined identifiers `<directoryname>`, `<WDBname>`, and `<timeplateName>` must be enclosed in double quotation marks if they contain TCL reserved symbols, embedded blanks, or collide with TCL reserved words.

For the SequenceMatch Conditioner, the optional Sequences block can define sequences of TimePlates that are to be matched. This block is specified by the **sequences** and **end sequences** reserved words. Sequences defined with the **seq** keyword are used for matching. Subsequences, which are sequences defined with the **subseq** keyword, can only be used in other sequence definitions; they are not directly matched. You can use any

SequenceNames you wish for your sequences, although if the name is a reserved word, it must appear in quotes. The sequence names allow you to use sequences inside other sequences. The sequence names are also used to identify the sequences in the report file.

The items in the sequences can be TimePlates, other sequences, or subsequences. Each item can also include an optional minimum and maximum repetition count. The minimum repetition count is allowed to be 0, which means the item is optional. Each item can also include a weight, similar to the weights in the **timeplates** statement, that causes the Conditioner to choose sequences with higher weights. TimePlates appearing in sequences can also include a SpecPlate, which is a substitution TimePlate that will be output in the destination WDB when that TimePlate is matched, in a fashion similar to the SpecPlates in the **timeplates** statement.

The items may be separated by "+" to indicate that they must appear sequentially, or by "|" (a vertical bar symbol) to indicate that either of the items may appear at this point. If both of the symbols "+" and "|" are used in the same sequence definition, parentheses should be used to clarify the intended grouping of the items.

The following example defines two subsequences and one sequence:

```
sequences
subseq "read" := tp1 '$10 + tpidle(1..3);
subseq "write" := tp2 + tpidle(1..3) <= tpout;
seq "memcycle" := "read" | "write";
end sequences
```

In this example, the Conditioner will match either the "read" or "write" subsequences. The "read" subsequence consists of TimePlate `tp1` followed by 1 to 3 occurences of TimePlate `tpidle`, and the "write" subsequence consists of TimePlate `tp2` followed by 1 to 3 occurrences of TimePlate `tpidle`. Additionally, TimePlate `tp1` has a user specified weight of 10 added to its computed weight, so if `tp1` and `tp2` match the input events with an equal computed weight (or a weight that differs by less than 10) then `tp1` will be chosen. Additionally, the `tpidle` timeplates are replaced in the output WDB with the timeplate `tpout`, but only when appearing after timeplate `tp2` in the "write" sequence.

If you use a SequencesBlock, you cannot also include a **timeplates** statement in the Timing block. To specify a TimePlate that you wish to match by itself, you must specify it in a sequence definition. For example, suppose you wish to match `tp1`, `tp2` and `tp3` individually. You could achieve this with the OR syntax, as shown in the following Sequences block:

Start Example

```
sequences
   seq “single_timeplates” := tp1 | tp2 | tp3;
end sequences
```

End Example

The above example specifies that the sequence “single_timeplates” consists of TimePlate `tp1`, or TimePlate `tp2` or TimePlate `tp3`. In other words, it is a sequence that is only one TimePlate long, and consists of any one occurrence of the specified TimePlates.

TCL produces the following messages identifying syntactic errors associated with this block:

`Missing wavesource <directoryname>.`

`Missing timingsource <WDBname>.`

`Wave source <directoryname> cannot be opened.`

`Timing source <WDBname> cannot be opened.`

`Duplicate timeplate name.`

`Invalid timeplate name.`

`Timeplate name does not exist in timingsource.`

`Start time cannot be set to end.`

`Start time must be less than or equal to stop time.`

`Start cycle must be less than or equal to stop cycle.`

`Cannot use prevstop on first match specification.`

`Stop time cannot be set to begin.`

`Stop time cannot be set to prevstop.`

`Persistence must be an integer >= 1.`

3.11 Program Control Directives

The Program Control Directives block controls the formatting of the test programs that WaveBridge produces. The control settings are applied globally to all test program files created by the WaveBridge run.

The syntax of the TCL Program Control Directives block is:

programcontrol
{ControlSpec}
end programcontrol

The following is an example of a TCL Program Control Directives block.

Start Example

```
programcontrol
   formwidth := 72;
   columnformat := every 10;
   comment := every 10 simtime,testtime,cyclenumber,
    memoryaddress,simcomment;
   structure
    main := “file1”;
    loadz;
    loads := “file1” replace;
    calprocedure := “file2” include;
   end structure
   mergecommonsignals := TRUE;
end programcontrol
```

End Example

A complete BNF syntactical representation of the Program Control Directives block follows:

ProgControlDirectives ::= { “programcontrol”
{ Control Specs }
“end” “programcontrol” }

ControlSpecs ::= (FormWidth | Columns | Comments
| ControlAndTiming | ProgramStructure | DisableUnused
| TimeClockFormat | ShowFormatBlock | TimePeriodFormat
| TimeEdgeFormat | TimeSetMerging | MergeCommonSignals | MapInitialXtoZ)

FormWidth ::= “formwidth” “:=” <colNum> “;”

Columns ::= “columnformat” “:=” (SignalGroups | EveryColumn
| ColumnList) “;”

SignalGroups ::= “signals” SignalName { “,” <signalName> }

SignalName ::= [‘ “ ’] <signalName> [‘ ” ’]

EveryColumn ::= “every” <colNum>

ColumnList ::= <colNum> { “,” <colNum> }

Comments ::= “comment” “:=” [“every” <rowNum>] CommentType { “,”
CommentType } “;”

CommentType ::= (“simtime” | “testtime” | “cyclenumber”
| “memoryaddress” | “simcomment” | “timeplatename”)

ControlAndTiming ::= { ControlPairs }

ControlPairs ::= (ControlSheetSets | TimingSheetSets
| ControlSecondarySheetSets | TimingSecondarySheetSets)

ControlSheetSets ::= “ControlSheetSets” “:=” EquationSheetSetName { “,”
EquationSheetSetName } “;”

ControlSecondarySheetSets ::= “ControlSecondarySheetSets” “:=”
EquationSheetSetName { “,” EquationSheetSetName } “;”

TimingSheetSets ::= “TimingSheetSets” “:=” EquationSheetSetName { “,”
EquationSheetSetName } “;”

TimingSecondarySheetSets ::= “TimingSecondarySheetSets” “:=”
EquationSheetSetName { “,” EquationSheetSetName } “;”

EquationSheetSetName ::= <EquationSheetName> “:” <ExpressionSetName>

EquationFlags := (ControlSheetSets | TimingSheetSets |TimingSecondarySheetSets |
LevelSheetSets | LevelSecondarySheetSets

LevelSheetSets := "levelsheetsets" ":=" LevelSheetSetName { ","
LevelSheetSetName } ";"

LevelSecondarySheetSets := "levelsecondarysheetsets" ":="
LevelSecondarySheetSetName { "," LevelSecondarySheetSetName } ";"

ProgramStructure ::= “structure”
{ StructureBody }
“end” “structure”

StructureBody ::= Partition [“:=” ‘ “ ’ <fileName> ‘ ” ’
[PartitionReplace]] “;”

Partition ::= <partitionName> [“radix” MemoryRadix]

PartitionReplace ::= (“include” | “replace”)

MemoryRadix ::= (“binary” | “octal” | “hexadecimal”
| “bin” | “oct” | “hex”)

DisableUnused ::= “DisableUnusedChannels” “:=” Boolean “;”

TimeClockFormat ::= (“freq” | “time”)

ShowFormatBlock ::= “showformatblock” “:=” Boolean “;”

TimePeriodFormat ::= (“freq” | “time”)

TimeEdgeFormat ::= (“freq” | “time”)

TimeSetMerging ::= (“true” | “false”)

MergeCommonSignals ::= (“true” | “false” | “new” | “new_flat”)

MapInitialXtoZ ::= “MapInitialXtoZ” “:=” Boolean “;”

TimeClockFormat, **TimePeriodFormat**, **TimeEdgeFormat** parameters control the master clock, period, or edge setting formats. The choices are either `time` (default) or `freq` to indicate that the values must be expressed in units of time or units of frequency.

The **ShowFormatBlock** parameter lets you disable or enable program generation that includes format and pattern characters in a single output file. See the chapter for your WaveBridge in the appropriate TDS tester guide to see if this feature is supported for your tester.

The **formwidth** parameter controls the number of the pattern characters output per line. The default maximum number of pattern characters is `80`.

NOTE

It is possible to set values with the FormWidth parameter that may not be valid for the target tester. This is to allow for simplified test program display during debugging.

The **columnformat** parameter provides three different options for controlling the column format. These options allow you to: define absolute column numbers after which spaces are inserted in the output file; define the order in which signals are output; and define where spaces are to be inserted in this ordered list.

`columnformat := every <integer>;`

<integer> is a positive integer that overrides the default of `10`, and it generates spaces repeatedly, based on the integer value specified. If `0` is specified, there are no column spaces.

`columnformat := <list of columns>;`

<list of columns> is one or more positive integers (separated by commas), which insert a space after each specified column. No spaces are added to the end of the column list.

`columnformat := signals <list of signals & groups & dashes>;`

<list of signals & groups & dashes> is one or more valid signal or group names or dashes (separated by commas). The `signals` parameter controls two elements of formatting: 1) where spaces are inserted (one space after each signal or group name), and 2) the signal to column mapping. If a dash (`-`) appears in the list, a space is inserted at the current position in the output listing. You can use multiple dashes to insert multiple spaces, as shown in *Example 4* on page 3-96. Signals and groups that are not included in the `columnformat` list appear after the specified signals and groups, and are listed in the order they occur in the database.

The default for signal/group-to-column-order is controlled by the Pin Assignment file. If a TCL file is used for a WaveBridge run, and if the TCL file contains a **columnformat** statement that specifies signal groups, the order specified in the TCL file is used in the output test program. For general information on the Pin Assignment file, see Chapter 4 in the *Getting Started Guide*. For detailed information, see the WaveBridge chapter specific to your tester.

NOTE

For some testers, the column order must match the tester pin order. For those testers, features that change the tester pin/column relationship are not applicable.

If the TCL file does not set the ordering, the order of the groups in the Pin Assignment file is the order in which the output columns are listed.

If no integer value is specified by the **columnformat** reserved word, a space is automatically placed between groups every ten columns.

The following examples assume that the tester has one large default group. Otherwise, spaces are automatically inserted between groups.

Example 1

```
columnformat := every 4;
```

results in pattern output:

```
1101 1111 0AF9 B26
0011 0000 02F9 223
1110 1110 0143 223
```

Example 2

```
columnformat := 2,5;
```

results in pattern output:

```
11 011 1110AF9B26
00 110 00002F9223
11 101 1100143223
```

Example 3

```
columnformat := every 0;
```

gives a pattern format:

```
110111110AF9B26
0011000002F9223
111011100143223
```

Example 4

```
columnformat := signals x, y, -, -, z;
```

gives a pattern format (a, b, and c are valid signals/groups in the WDB, but appear last because they are not specified in the statement):

```
x  y   z a b c
11 1   a 1 1 1
01 0   9 1 0 0
10 1   4 0 1 1
```

Radix is associated with each signal, group, or bus defined in the database. Since the radix information is in the database, it must be entered by editing the WGL version of the WDB and converting the WGL back to a WDB, using the WGL In Converter module. You can also use WaveMaker's Signal Definition Editor. The database radix (group radix) controls the radix relation in an output pattern.

You can select the type and placement of pattern row comments that appear in the test program using the **Comment** statement. Pattern row comment fields appear in the order given in the list specified by the **Comment** statement. If this statement is absent, there are no comments on the pattern rows. By default (if you do not provide a TCL file), all pattern row comments are used.

NOTE

The comments from the simulation file are always inserted into the test program.

The **Comments** directive allows you to specify which comments will appear in the output pattern. The rows in which the comments appear are determined by the **rowNum** parameter. If you specify **simcomment** as one of the **CommentType** parameters, then all simulation comments are inserted into the output pattern.

Data that you use repeatedly, for many different test programs, can be stored in separate TCL files and brought into the test program. This lets you create a library of such data files, with each file containing specific types of data in proper syntax. The TCL **structure** sub-block provides a method of inserting user-defined files containing this repetitive data (called include files) into the final test program.

Each **structure** sub-block has a <partitionName> that defines (in a tester-specific way) where the include file is to go. It is optionally followed by the **radix** reserved word. Using **radix** permits selection of `binary` (or `bin`), `octal` (`oct`), or `hexadecimal` (`hex`)

radices. Optionally, you may specify a source <fileName> to use for the inclusion (or replacement). If the file name is not present, no action is taken. The file name must be enclosed in double quotation marks (“ ”). The file name can be followed by a reserved word, either **include** or **replace**. If neither reserved word is specified, an inclusion is performed by default at the appropriate location.

If the **replace** reserved word is used, your include file replaces the section of the test program that it goes before.

Example of a structure sub-block:

Start Example

```
structure
   start “file1”; # put file1 section start of test program
   start “file2”; # put file2 section start of test program after file1
    main;          # no action
   loads := “file3” replace; # replace section “loads” with file3.
end structure
```

End Example

TCL produces the following messages identifying syntactic errors associated with this block:

```
Form width must be > 0 and <= 4096.

Only one of the 3 column format options may be chosen.

Column must be >= 0.

Column numbers must be unique in column list.

Column numbers must increase in column list.

Comment type has already been defined in comment list.

Comment row spacing must be > 0.
```

The **ControlSheetSets**, **TimingSheetSets**, **LevelSheetSets, LevelSecondarySheetSets, ControlSecondarySheetSets**, and **TimingSecondarySheetSets** parameters are used only if your tester supports test programs that use equations and you have purchased and installed a TDS WaveBridge with the optional equation support module. These parameters control certain aspects of equation usage in the output test program:

- Overriding the default expression sets used for each equation sheet in the source WDB
- Specifying more than one expression set for an equation sheet from the source WDB to be output in the test program
- Specifying the location of program segments containing equations in the output test program

In the WDB, there can be many expression sets for each equation sheet, but only one of these expression sets can be active for the equation sheet at any given time. There are various mechanisms in WGL for specifying which expression set is active in the source WDB. Some of these settings in the source WDB can be overridden using the **ControlSheetSets**, **TimingSheetSets**, **LevelSheetSets, LevelSecondarySheetSets**, **ControlSecondarySheetSets**, and **TimingSecondarySheetSets** parameters in the TCL file. For information on the WGL mechanisms for controlling active (or default) expression sets, see *EquationSheet* on page 2-54 and *EquationDefaults* on page 2-66 of this guide.

You can use the **ControlSheetSets, LevelSheetSets**, and **TimingSheetSets** parameters to override the active expression set for an equation sheet in the source WDB. For example, the WGL file for your source WDB specifies the following settings:

Start Example

```
equationsheet Sheet_1
   exprset worst
      Vcc1:= 4.5V;
      TempDegC1 := 70;
      Textern1 := 10nS;
   end
   exprset best
      Vcc1 := 5.75V;
      TempDegC1 := 0;
      Textern1 := 0nS;
   end
end

equationsheet Sheet_2
   exprset worst
      Vcc2:= 4.5V;
      TempDegC2 := 70;
      Textern2 := 10nS;
   end
   exprset normal
```

```
      Vcc2 := 5V;
      TemDegC2 := 30;
      Textern2 := 5nS;
   end
   exprset best
      Vcc2 := 5.75V;
      TemDegC2 := 0;
      Textern2 := 0nS;
   end
end

equationdefaults
   Sheet_1:worst;
   Sheet_2:best;
end
```

End Example

The WGL fragment above specifies that the active (or default) expression set for the equation sheet named `Sheet_1` is `worst`, and the active expression set for `Sheet_2` is `best`.

To change the active expression sets in the WDB by using TCL parameters, make an entry in your TCL file like the one shown in the following example:

Start Example

```
programcontrol
   . . .
   TimingSheetSets := Sheet_1:best, Sheet_2:normal;
   . . .
end
```

End Example

In your output test program, the active expression set for `Sheet_1` will be `best` and the active expression set for `Sheet_2` will be `normal`.

You may want to create an output test program that includes several (or all) of the expression sets for each equation sheet. This is possible by using the **ControlSecondarySheetSets**, **LevelSecondarySheetSets**, and **TimingSecondarySheetSets** parameters. When you use these parameters, all of the expression sets that you specify appear in the output program. All expression sets except for the active expression set are commented out, or otherwise disabled. For subsequent test runs, comment out the active expression set and remove the comment characters for the

expression set you want to activate. The following example shows how to use the **ControlSecondarySheetSets**, **LevelSecondarySheetSets**, and **TimingSecondarySheetSets** parameters.

Start Example

```
programcontrol
   . . .
   TimingSheetSets := Sheet_1:worst, Sheet_2:normal;
   TimingSecondarySheetSets := Sheet_1:best, Sheet_2:best, Sheet_2:worst;
   . . .
end
```

End Example

The parameters shown in the previous example cause all expression sets for all of the equations sheets defined in the WGL example on page 3-98 to be included in the output test program, with the default expression sets active.

Depending on your tester, the **ControlSheetSets**, **LevelSecondarySheetSets**, and **TimingSheetSets** parameters in the TCL file determine where in the output test program the equations are placed. If you specify these parameters in the TCL file, your WaveBridge automatically places these equations in the correct location in the test program; if not specified in the TCL file, all equations automatically are placed in a single location in the test program, based on your tester's requirements. Since each tester has different requirements for where the equations can be located, see the TDS chapter for your specific WaveBridge for details.

Depending on your tester and the corresponding WaveBridge with equation support, you may not need to use the **ControlSheetSets**, **LevelSheetSets**, **TimingSheetSets**, **ControlSecondarySheetSets**, **LevelSecondarySheetSets**, and **TimingSecondarySheetSets** parameters in the Program Control Directives block. For WaveBridges such as these, all default expression set assignments in the source WDB are used in the output test program. See the TDS chapter for your specific WaveBridge to determine if you need to use TCL parameters to generate a test program that includes equations.

CAUTION

Improper use of ***ControlSheetSets****,* ***LevelSheetSets****,* ***TimingSheetSets****,* ***ControlSecondarySheetSets****,* ***LevelSecondarySheetSets****, or* ***TimingSecondarySheetSets*** *parameters can result in unexpected behavior in equation output. Failure to use these parameters correctly can result in the inaccurate equations being used in the test program, since variables and equations may be output in an unexpected order. Adherence to the following rules will insure correct output for your equations:*

• When overriding ***TimingSheetSets*** *or* ***LevelSheetSets****, also override* ***ControlSheetSets****. If not, constants that are declared in the Control Sheet could be output after the TimingSheet causing erroneous results*

• When specifying multiple sheets for any of the four parameters, be sure to specify them in the desired output order.

For more information on Equation Sheets, please refer to the EquationSheet topic on page 2-54.

The **TimeSetMerging** parameter specifies whether the WaveBridge should try to merge identical timesets or not. The default setting is `true`, indicating that the WaveBridge will try to merge timesets. To turn merging off, set **TimeSetMerging** to `false`.

The goal of **TimeSetMerging** is to reduce the number of tester resources required for the output test program. The simple example below shows the effect of **TimeSetMerging**. Before merging, four timesets are used, whereas afterwards only two are used; TS1, TS3, and TS4 are combined.

Before			*After*		
TimePlate Name	TimeSet Name	Values	TimePlate Name	TimeSet Name	Values
read	TS1	10ns, 20ns	read, idle, add	TS1	10ns, 20ns
write	TS2	20ns, 20ns	write	TS2	20ns, 20ns
idle	TS3	10ns, 20ns			
add	TS4	10ns, 20ns			

There is one situation when you would want to set **TimeSetMerging** to `false`. Some WaveBridges (such as Teradyne A580, Schlumberger ITS 9000, and LTX/Trillium) generate timeset names from the TimePlate names; this occurs only when **TimeSetMerging** is `false`. (For these testers, when **TimeSetMerging** is set to `true`, the WaveBridge generates default timeset names.)

The **MergeCommonSignals** parameter specifies whether all signals having the identical DC, formats, and timing across all TimePlates are merged into a single pin group in the tester program. **MergeCommonSignals** options are (the WaveBridge for your tester may not support all of these options):

- `true`

  The WaveBridge merges common signals into groups. This is desirable for final test program generation when you want the most compact test program. The group names are a concatenation of the signal names, and are used in the output test program. The WaveBridge does not use the groups defined in the input WDB and Pin Assignment File.

- `false`

  The WaveBridge outputs timing for each individual signal.

- `new`

  Whenever possible, the WaveBridge uses input WDB and Pin Assignment File groups in the output test program. Additionally, the WaveBridge forms groups for signals not included in WDB or Pin Assignment File groups. These groups are named PINGRP<n>, are output to the destination WDB, and are used in the output test program.

- `new_flat`

  As with the `new` option, whenever possible the WaveBridge uses input WDB and Pin Assignment File groups in the output test program. The WaveBridge forms groups for signals not included in WDB or Pin Assignment File groups. These groups are named PINGRP<n> and are output to the destination WDB. In contrast to the `new` option, these groups are not used in the output test program.

The **MapInitialXtoZ** parameter specifies whether X states on bidirectional signals are conditionally mapped to Z states, which ensures that the driver is disabled for a compare event. The default value is `false`. If **MapInitialXtoZ** is set to `true`, it affects only

bidirectional signals. An X state on an output track (of a bidirectional signal) or a bidirectional track is mapped to a Z state under the following conditions:

1. The X state is the first event in the track, or

2. The event preceding the X state is an input event

If the mapping occurs for an output track, the specification of the track's direction is changed to B (bidirectional) in the destination WDB. (The source WDB is not changed.) The WaveBridge makes this change because the track has the original output events plus the mapped Z state, which is treated as an input event. Note that a TimePlate is not allowed to have more than one bidirectional track. If this condition is violated, a fatal TRC message is generated. You will have to remove the bidirectional track from the TimePlate or disable the mapping.

The **MapInputXto0** parameter specifies whether the X states on Input signals should be mapped to zero. This command is available only on the Teradyne J750 IBridge and WaveBridge when specified in the Program Control block of the user generated TCL file.

MapInputXto0 := True enables this command.

MapInputXto0 := False is the default and no mapping takes place.

3.12 Pattern Load Directives

The Pattern Load Directives block (sometimes referred to as the Burst block) controls the transformation of complete WDBs (after timing has been assigned by TimePlate matching) into test program files and pattern bursts.

The syntax of the TCL Pattern Load Directives block is:

```
burst [ <burstName> ]
BurstBody
end burst [ <burstName> ]
```

Example of a TCL Pattern Load Directives block:

Start Example

```
burst test1
   source
      directory   := goodwdb;
      start       := begin;
      stop        := 999ms;
      compress    := true;
   end source

   destination
      program    := a.prog,
      pinmap     := a.pin,
      channelmap := a.ch,
      socket     := a.soc,
      pattern    := a.pat;
   end destination
end burst test1
```

End Example

A complete BNF syntactical representation of the Pattern Load Directives block follows:

PatternLoadDirectives ::= { “burst” [<burstName>]
BurstBody
“end” “burst” BlockName }

BurstBody ::= { BurstItems }

Destination ::= “destination” “:=” DestName { “,” DestName } “;”

LabelStruct ::= “labelprefix” “:=” LabelName { “,” LabelName } “;”

BurstItems ::= (SourceBlock | DestBlock | LabelBlock)

SourceBlock ::= “source”
{ SourceSpec }
“end” “source”

SourceSpec ::= (SourceDB | StartTime | StopTime | Pattern | Compress | IncrResAssign |SourceSets)

SourceDB ::= (“directory” | “database”) [“modules”] “:=” (<dirNameAN> | <dirNameStr>) “;”

StartTime ::= “start” “:=” TimeSpec “;”

StopTime ::= “stop” “:=” TimeSpec “;”

TimeSpec ::= (“begin” | “end” | “prevstop” | Time)

Time ::= (<intTime> | <floatTime>) [TimeUnit]

TimeUnit ::= (“ps” | “ns” | “us” | “ms” | “s”)

Pattern ::= “pattern” “:=” <patburstname> { ",“ <patburstname>} “;”

Compress ::= “compress” “:=” Boolean “;”

Boolean ::= (“true” | “false”)

SourceSets ::= (GreedyTGs | FormatUsageSet | TgUsageSet
| StrobeUsageSet)

GreedyTGs ::= “notgsharing” “:=” “[” SigList “]” “;”

FormatUsageSet ::= “formatusage” “:=” “[” SigList “]” “[” NameList “]” “;”

TgUsageSet ::= “tgusage” “:=” “[” SigList “]” “[” NameList “]” “;”

StrobeUsageSet ::= “strobeusage” “:=” “[” SigList “]” “[” NameList “]” “;”

SigList ::= SigName { “,” SigName }

SigName ::= (<signalname> | <signalnameStr>)

NameList ::= NameOrString { “,” NameOrString }

NameOrString ::= (<name> | <nameString> | “edge” | “window”)

DestBlock ::= “destination”
{ BurstDestSpec “;” }
“end” “destination”

BurstDestSpec ::= (DestName | LoadAddr | FileFormat)

LoadAddr ::= “loadaddress” “:=” <testerMemoryAddr>

FileFormat ::= “fileformat” “:=” (“binary” | “ascii”)

DestName ::= (“program” | “pinmap” | “channelmap” | “socket”
| “pattern” | “timeset” | “relevance” | “acspec” | “dcspec”

```
| “directory” | “netlist” | “prefix” | “extension” | “format”
| “setup” | “database”) “:= ” [ ‘ “ ’ ] <Name> [ ‘ ” ’ ] “ ; ”

LabelBlock ::= “labelprefix”
                    { LabelName “;” }
               “end” “labelprefix”

LabelName ::= ( “format” | “timing” | “control” | “source” | “pattern”) “:=”
                  [ ‘ “ ’ ] <labelName> [ ‘ ” ’ ] “;”
```

In the simplest case, only one Pattern Load Directives block is required. The WDB specified in the **source** definition of the Pattern Load Directives block is required and is the same WDB specified by the **destination** definition in the appropriate Match Directives block. (Match Directives blocks create WDBs; Pattern Load Directives blocks consume them to create test programs.) Alternatively, the WDB specified by the **sourceDB** reserved word could have been created with WGL source or via WaveMaker editors.

By default, the Pattern Load Directives block utilizes all the pattern rows and subroutines in the WDB. If **start** and **stop** times appear in the Pattern Load Directives block, only that section of the associated source WDB is transformed into a test program.

The **pattern** parameter in the source definition specifies the name of the Pattern block in the WDB to use as the source of the burst.

The **compress** reserved word controls whether compression is done on the corresponding burst.

The **destination** reserved word can be used to specify the name(s) and type(s) of output file(s) in which to put the test program. Some testers can accept a single test program as input. You can put the test program in a single file by specifying only the **program** reserved word. The other reserved words (**pinmap**, **channelmap**, **socket**, **pattern**, **timeset**, **relevance**, **acspec**, **dcspec**, **directory**, **netlist**, **prefix**, **extension**, **format**, and **setup)** allow the test program generated by WaveBridge to be output to various other files, as may be appropriate to your tester. For specific information, refer to the WaveBridge chapter specific to your tester.

The **labelprefix** reserved word can tailor the creation of labels in the test program source. The actual labels that are created usually have a numeric suffix appended to the <labelName> specified by these settings. By optionally enclosing the <labelName> in double quotation marks (“ ”), you can use any character string (including white space) as a valid prefix. For detailed information, see the WaveBridge chapter specific to your tester.

The **formatusage** reserved word describes the set of available formats that WaveBridge uses in resource allocation. If this reserved word is not used, WaveBridge defaults to the full set (all formats are available for all signals). If this reserved word is used, only the formats in the set are used during format allocation for the specified signals.

An example of a **formatusage** clause is:

```
FormatUsage := [ALL][RZO, XOR, FCRA, WINDOW, EDGE, MCLK,
       DCLK, DSTB, IO, RI, MUXPIN];
FormatUsage := [CLK] [MCLK]
```

The use and effects of the other options vary from tester to tester.

The **tgusage** reserved word describes the set of available timing generators that WaveBridge uses in resource allocation. If this reserved word is not used, WaveBridge defaults to the full set (all timing generators are available for all signals). If this reserved word is used, only the timing generators in the set are used during allocation for the specified signals.

An example of a **tgusage** clause is:

```
TGusage := [clk, clk2][tg8];
```

This example forces the signals `clk` and `clk2` to share the `tg8` timing generator edges. WaveBridge validates and uses the signal-to-TG mapping specified from the settings specified by the **tgusage** reserved word. If an illegal timing generator is referenced or it is not compatible with the actual waveform timing, WaveBridge discards its use and issues a warning message.

By default, WaveBridge seeks to share timing generators whenever possible, using like timing values described in the individual timing tracks of a WDB TimePlate. As an example, this means that one signal with a return-to-zero format with edge times at 100ns and 200ns (defined in one timing track of a TimePlate) and another signal with a return-to-one format with edge times of 100ns and 250ns (defined in another timing track of the same TimePlate) can share a timing generator. (Note that the direction of the signals must be identical, but the state of the signals at the shared edge is unimportant.) Both signals have an edge time (100ns) that can use the same timing generator, as shown in the WGL example of a TimePlate below.

Example of shared timing generator opportunity:

Start Example

```
timeplate read1 period 300ns
   clock1 := input [0ps:D, 100ns:U, 200ns:D];
   in := input [0ps:U, 100ns:D, 250ns:U];
   .   .   .
end
```

End Example

By default, signals `clock1` and `in` share a timing generator for the edge specified at 100ns. To suppress this default WaveBridge behavior, you can use the **NoTGSharing** TCL statement. The WaveBridge timing generator sharing algorithm is inhibited for all the signals mentioned in the signal list, as in:

```
NoTGSharing := [clock1, in];
```

The signals `clock1` and `in`, because they are listed in the **NoTGSharing** statement, are exempted from sharing timing generators, even if they have edge times that would otherwise be conducive to sharing.

The **strobeusage** clause describes the type of strobe for the specified signals that WaveBridge uses in resource allocation. If this parameter is absent, it defaults to the setting of **CompareType**, which can be `edge`, `window`, or `both`. If this parameter is defined, the specified signals are allocated the strobe type.

Examples of **strobeusage** clauses are:

```
StrobeUsage := [CMP1, CMP2][Edge];
StrobeUsage := [CMP3] [Window];
```

This example sets up cause signals `CMP1` and `CMP2` to utilize edge strobing, and signal `CMP3` to utilize a window strobe. If the suggested strobe type is incompatible with the signal or not enough strobe resources are available, WaveBridge discards the suggestion and issues an appropriate warning.

NOTE

A warning message is issued when "`CompareType := both;`*" is specified in either a Main or Override TCL files and the Window Strobe in the WDB is too narrow. The message is as follows:*

```
Warning: Found in Resource Assignment
Strobe type has been changed from window to edge. Because
the window strobe is too narrow for signal sig2 in
timeplate tp1, the output test program will use an edge
strobe.
```

The **loadaddress** clause describes where in tester memory the pattern is loaded. The default value of <testerMemoryAddr> is zero (`0`). This provides a mechanism for loading reusable test program parts in one portion of tester memory (starting at address 0), and each test program burst can be repeatedly loaded beyond the shared portion.

The **FileFormat** clause describes whether a binary or ASCII test program is to be generated for the burst. The default is ASCII. Any tester that does not support `BINARY` reports an error when WaveBridge setting is not `ASCII`.

3.13 TCL Quick Reference

The following table alphabetically lists all BNF entries that define Test Control Language syntax. Use this table directly to resolve questions about TCL syntax, or go to the indicated page(s) for more information.

Table 3-22. BNF Entries for Test Control Language (TCL).

BNF Entry	*Page*
AteConstraints ::= ["ate" { AteConstraint ";" } "end" "ate"]	3-10
ATEVersion ::= "AteVersion" ":=" ' " ' <versionString> ' " ' ";"	3-18
Boolean ::= ("true" | "false")	3-18
BurstBody ::= { BurstItems }	3-104
BurstDestSpec ::= (DestName | LoadAddr | FileFormat)	3-105

Table 3-22. BNF Entries for Test Control Language (TCL).

BNF Entry	*Page*
BurstItems ::= (SourceBlock | DestBlock | LabelBlock)	3-104
CardInfo ::= [CardType] [CardSlots]	3-76
CardSlots ::= “cardslots” “:=” “[” Slot { ”,” Slot } “]” “;”	3-76
CardType ::= “cardtype” “:=” <cardTypeName> “;”	3-76
CmpConstraints ::= “compareconstraint” Subscript “:=” TimeExpr	3-33
CmpControl ::= (CmpEdgeRange | CmpWindowRange | CmpStrobeType)	3-33
CmpEdgeRange ::= “compareedgerange” Subscript “:=” Range	3-33
CmpMatchResolution ::= “comparematchresolution” “:=” Time	3-33
CmpResolution ::= “compareresolution” “:=” Time	3-33
CmpStrobeType ::= “comparetype” “:=” (“edge” | “window” | “both”)	3-33
CmpWindowMinimum ::= “comparewindowmin” “:=” Time	3-33
CmpWindowRange ::= “comparewindowrange” Subscript “:=” Range	3-33
ColumnList ::= <colNum> { “,” <colNum> }	3-92
Columns ::= “columnformat” “:=” (SignalGroups | EveryColumn | ColumnList) “;”	3-92
Comments ::= “comment” “:=” [“every” <rowNum>] CommentType { “,” CommentType } “;”	3-92
CommentType ::= (“simtime” | “testtime” | “cyclenumber” | “memoryaddress” | “simcomment” | “timeplatename”)	3-92
Compare ::= (CmpControl | CmpConstraints | CmpMatchResolution |CmpResolution | CmpWindowMinimum)	3-33
Compress ::= “compress” “:=” Boolean “;”	3-105
CompressionAdjacency ::= CompressionSpacing	3-17
CompressionItem ::= (“burstbegin” | “burstend” | “vector” | “repeatedvector” | “loopbegin” | “loopend” | MoreCompressionItems)	3-17

Table 3-22. BNF Entries for Test Control Language (TCL).

BNF Entry	*Page*
CompressionMemRowMax ::= “compressionmemrowmax” “:=” RowMax	3-49
CompressionSpacing ::= “compressionspacing” CompressionItem “..” CompressionItem “:=” <spacing>	3-17
ControlAndTiming ::= { ControlPairs }	3-92
ControlPairs ::= (ControlSheetSets | TimingSheetSets | ControlSecondarySheetSets | TimingSecondarySheetSets)	3-92
ControlSecondarySheetSets ::= “ControlSecondarySheetSets” “:=” EquationSheetSetName { “,” EquationSheetSetName } “;”	3-92
ControlSheetSets ::= “ControlSheetSets” “:=” EquationSheetSetName { “,” EquationSheetSetName } “;”	3-92
ControlSpecs ::= (FormWidth | Columns | Comments | ControlAndTiming | ProgramStructure | DisableUnused | TimeClockFormat | ShowFormatBlock | TimePeriodFormat | TimeEdgeFormat)	3-91
Cycle := (CycleMinimum | CycleMaximum | CycleResolution |CycleResolutionTolerance |CycleMatchResolution | ScanCycleMax | ScanCycleMinScanCycleResolution)	3-20
CycleMatchResolution ::= “cyclematchresolution” “:=” Time	3-20
CycleMaximum ::= “cyclemax” “:=” Time	3-20
CycleMinimum ::= “cyclemin” “:=” Time	3-20
CycleResolution ::= “cycleresolution” “:=” Time	3-20
CycleResolutionTolerance ::= “cycleresolutiontolerance” “:=” Time	3-20
CycleSteal ::= “cyclesteal” “:=” Boolean	3-39
DC ::= (“pininvoltage” | “pinoutvoltage” | “pinoutcurrent”) “:=” DCvalues	3-22
DClabel ::= (<dcName> | <dcString>)	3-22
DCsign ::= (“+” | “-”)	3-22
DCunit ::= (“ma” | “ua” | “a” | “v”)	3-22

Table 3-22. BNF Entries for Test Control Language (TCL).

BNF Entry	*Page*
DCvalue ::= [DCsign] <voltsOrAmps> DCunit	3-22
DCvalues ::= [DClabel] “[” DCvalue { “,” DCvalue } “]”	3-22
DelayChannelRange ::= “delaychannelrange” Subscript “:=” Range	3-32
DestBlock ::= “destination” { BurstDestSpec “;” } “end” “destination”	3-105
Destination ::= “destination” “:=” DestName { “,” DestName } “;”	3-104
DestinationBlock ::= “destination” { DestSpec } “end” “destination”	3-84
DestName ::= (“program” | “pinmap” | “channelmap” | “socket” | “pattern” | “timeset” | “relevance” | “acspec” | “dcspec” | “directory” | “netlist” | “prefix” | “extension” | “format” | “setup” | “database”) “:= ” [‘ “ ’] <fileName> [‘ ” ’] “ ; ”	3-105
DestSpec ::= (WaveDestination | DestTime)	3-84
DestTime ::= (“start” | “stop”) “:=” TimeSpec “;”	3-84
DisableUnused ::= “DisableUnusedChannels” “:=” Boolean “;”	3-93
Drive := (DriveControl | DrivePulseMinimum | DriveResolution)	3-33
DriveChannelRange ::= “drivechannelrange” Subscript “:=” Range	3-33
DriveConstraints ::= “driveconstraint” Subscript “:=” TimeExpr	3-34
DriveControl ::= (DriveChannelRange | DriveMatchResolution | DriveConstraints | DriveOnMinimum | DriveOffMinimum)	3-33
DriveMatchResolution ::= “drivematchresolution” “:=” Time	3-34
DriveOffMinimum ::= “driveoffmin” “:=” Time	3-34
DriveOnMinimum ::= “driveonmin” “:=” Time	3-34
DrivePulseMinimum ::= “drivepulsemin” “:=” Time	3-34
DriveResolution ::= “driveresolution” “:=” Time	3-34

Table 3-22. BNF Entries for Test Control Language (TCL).

BNF Entry	*Page*
EquationSheetSetName ::= <EquationSheetName> “:” <ExpressionSetName>	3-92
EventBlock := “events” { EventSpec } “end” “events”	3-84
EventSpec ::= (WaveSource | StartTime | StopTime)	3-84
EveryColumn ::= “every” <colNum>	3-92
FileFormat ::= “fileformat” “:=” (“binary” | “ascii”)	3-105
Fixture ::= (SocketType | VernierRange | FixtureOffset)	3-28
FixtureOffset ::= “fixtureoffset” “:=” [“-”] Time	3-28
Force ::= (ForceChannelRange | ForcePulseMinimum | ForceConstraints | ForceResolution | ForceMatchResolution |DelayChannelRange)	3-32
ForceChannelRange ::= “forcechannelrange” Subscript “:=” Range	3-32
ForceConstraints ::= “forceconstraint” Subscript “:=” TimeExpr	3-33
ForceMatchResolution ::= “forcematchresolution” “:=” Time	3-33
ForcePulseMinimum ::= “forcepulsemin” “:=” Time	3-33
ForceResolution ::= “forceresolution” “:=” Time	3-33
FormatCharMap ::= “formatcharmap” “:=” <formatCharMap>	3-39
FormatSet ::= “formatsetmax” “:=” <formatsetMaximum>	3-39
FormatUsageSet ::= “formatusage” “:=” “[” SigList “]” “[” NameList “]” “;”	3-105
FormWidth ::= “formwidth” “:=” <colNum> “;”	3-92
GlobalRepetition ::= “repetition” “:=” <integer>	3-81
GreedyTGs ::= “notgsharing” “:=” “[” SigList “]” “;”	3-105
HizRowMaximum ::= “hizrowmax” “:=” RowMax	3-49
IgnoreWhen ::= “ignore” “:=” (“informative” | “warning”)	3-81
IntRange ::= <lowerBound> “..” <upperBound>	3-32

Table 3-22. BNF Entries for Test Control Language (TCL).

BNF Entry	*Page*
IORowMaximum ::= “iorowmax” “:=” RowMax	3-49
LabelBlock ::= “labelprefix” { LabelName “;” } “end” “labelprefix”	3-106
LabelName ::= (“format” | “timing” | “control” | “source” | “pattern”) “:=” [‘ “ ’] <labelName> [‘ ” ’] “;”	3-106
LabelStruct ::= “labelprefix” “:=” LabelName { “,” LabelName } “;”	3-104
LoadAddr ::= “loadaddress” “:=” <testerMemoryAddr>	3-105
LocalTimeSet ::= “localtimesetmax” “:=” <localtimesetMaximum>	3-68
Loop ::= (LoopCompression | LoopCountMaximum | LoopCountMinimum | LoopNest | LoopRowMinimum |LoopRowMaximum | LoopConstraint)	3-42
LoopAtEndLegal ::= “loopatendlegal” “:=” Boolean	3-43
LoopAtStartLegal ::= “loopatstartlegal” “:=” Boolean	3-43
LoopCompression ::= “loopcompression” “:=” Boolean	3-42
LoopConstraint ::= (LoopSpacingMin | LoopAtStartLegal | LoopAtEndLegal| LoopRepeatCountMinimum | LoopRepeatCountMaximum)	3-42
LoopCountMaximum ::= “loopcountmax” “:=” <loopCountMaximum>	3-42
LoopCountMinimum ::= “loopcountmin” “:=” <loopCountMinimum>	3-42
LoopNest ::= “loopnestmax” “:=” <loopNestMaximum>	3-42
LoopRepeatCountMaximum ::= “looprepeatcountmax” “:=” <loopRepeatMaximum>	3-43
LoopRepeatCountMinimum ::= “looprepeatcountmin” “:=” <loopRepeatMinimum>	3-43
LoopRowMaximum ::= “looprowmax” “:=” <loopRowMaximum>	3-43
LoopRowMinimum ::= “looprowmin” “:=” <loopRowMinimum>	3-43
LoopSpacingMin ::= “loopspacingmin” “:=” <loopSpaceMin>	3-43

Table 3-22. BNF Entries for Test Control Language (TCL).

BNF Entry	*Page*
MaskRowMaximum ::= “maskrowmax” “:=” RowMax	3-49
MatchBody ::= { MatchSpecs “;” }	3-83
MatchDirectives ::= { “match” MatchBody “end” “match” }	3-10, 3-83
MatchSpecs ::= (EventBlock | TimingBlock | DestinationBlock)	3-83
MapInitialXtoZ ::= “mapinitialxtoz” “:=” Boolean “;”	3-33
Mclk ::= (MclkType | MclkRate | MclkLeading | MclkTrailing | MclkRecovery | MclkEdgeResolution | MclkCountMaximum)	3-45
MclkConstraints ::= “multiclockconstraint” Subscript “:=” TimeExpr	3-46
MclkCountMaximum ::= “multiclockcountmax” “:=” <countMaximum>	3-46
MclkEdgeResolution ::= “multiclockedgeresolution” “:=” Time	3-46
MclkLeading ::= (MclkLeadingMinimum | MclkLeadingMaximum)	3-45
MclkLeadingMaximum ::= “multiclockleadingmax” “:=” Time	3-46
MclkLeadingMinimum ::= “multiclockleadingmin” “:=” Time	3-46
MclkPulseMin ::= “multiclockpulsemin” “:=” Time	3-46
MclkRate ::= (MclkRateResolution | MclkRateMinimum | MclkRateMaximum)	3-45
MclkRateMaximum ::= “multiclockratemax” “:=” Time	3-46
MclkRateMinimum ::= “multiclockratemin” “:=” Time	3-46
MclkRateResolution ::= “multiclockrateresolution” “:=” Time	3-46
MclkRecovery ::= (MclkPulseMin | MclkConstraints)	3-46
MclkTrailing ::= (MclkTrailingMinimum | MclkTrailingMaximum)	3-46
MclkTrailingMaximum ::= “multiclocktrailingmax” “:=” Time	3-46
MclkTrailingMinimum ::= “multiclocktrailingmin” “:=” Time	3-46
MclkType ::= “multiclocktype” “:=” (“none” | “cyclic” | “acyclic”)	3-45

Table 3-22. BNF Entries for Test Control Language (TCL).

BNF Entry	*Page*
McodeCallCost ::= “microcodecallcost” “:=” <microcodeCallCost>	3-44
McodeLoopCost ::= “microcodeloopcost” “:=” <microcodeLoopCost>	3-44
McodeRepeatCost ::= “microcoderepeatcost” “:=” <microcodeRepeatCost>	3-44
McodeRowMaximum ::= “microcoderowmax” “:=” <microcodeRowMaximum>	3-44
McodeSubrCost ::= “microcodesubrcost” “:=” <microcodeSubrCost>	3-44
MemoryModel ::= “MemoryModel” “:=” <memoryModel> “;”	3-18
MemoryRadix ::= (“binary” | “octal” | “hexadecimal” | “bin” | “oct” | “hex”)	3-93
xSignals ::= (“true” | “false” | “new” | “new_flat”)	3-93
MessageBody ::= { (Severity | Repetition | MessageText) “;” }	3-78
MessageOverrides ::= { “message” <messageName> MessageParams MessageBody “end” “message” [<messageName>] }	3-10, 3-78
MessageParams ::= [“(” <parameterName> { “,” <parameterName> } “:” ParamType “)”]	3-78
MessageString ::= ‘ “ ’ <string> ‘ ” ’ { “+” ‘ “ ’ <string> ‘ ” ’ }	3-78
MessageText ::= (“terse” | “cause” | “location” | “remedy” | “generic”) “:=” MessageString	3-78
MicroCode ::= (McodeRowMaximum | McodeRepeatCost | McodeSubrCost | McodeCallCost | McodeLoopCost)	3-44
MoreCompressionItems ::= (“call” | “repeatedcall” | “subroutinebegin” | “subroutineend” | “scanrun”)	3-18
MuxConversion ::= “MuxConversion” “:=” (“OLDtoNEW” | ““NEWtoOLD” | “NONE”) “;”	3-18
NameList ::= NameOrString { “,” NameOrString }	3-105
NameOrString ::= (<name> | <nameString> | “edge” | “window”)	3-105

Table 3-22. BNF Entries for Test Control Language (TCL).

BNF Entry	*Page*
ParamType ::= (“signal” | “TimePlate” | “list” | “time” | “integer” | “string”)	3-78
Partition ::= <partitionName> [“radix” MemoryRadix]	3-93
PartitionReplace ::= (“include” | “replace”)	3-93
PatBoundary ::= “patternboundary” “:=” <compressionValue>	3-49
PatBurstMax ::= “burstrowmax” “:=” <burstRowMaximum>	3-49
PatCompression ::= “patterncompression” “:=” (“yes” | “no” | “threshold”)	3-49
PatRowMax ::= “patternrowmax” “:=” <patternRowMaximum>	3-49
Pattern ::= (PatCompression | PatBoundary | PatRowMaximum | PatBurstMaximum | CompressionMemRowMax | IORowMax | MaskRowMax | HizRowMask)	3-49
Pattern ::= “pattern” “:=” <patburstname> { ",“ <patburstname>} “;”	3-49
PatternLoadDirectives ::= { “burst” [<burstName>] BurstBody “end” “burst” [<burstName>] }	3-10, 3-104
Persistence ::= “persistence” “:=” <persistenceValue> “;”	3-84
Pin ::= (PinInOutMaximum | PinInOutMinimum | PinInMaximum | PinOutMaximum)	3-51
PinGroupBody ::= CardInfo Pins { AteConstraint “;” }	3-76
PinGroupName ::= (<name> | <nameString>)	3-76
PinGroups ::= { “pingroup” <PinGroupName> { PinGroupBody “;” } “end” “pingroup” <PinGroupName>}	3-10, 3-76
PinInMax ::= “pininmax” “:=” <pinInMaximum>	3-51
PinInOutMax ::= “pininoutmax” “:=” <pinInOutMaximum>	3-51
PinInOutMin ::= “pininoutmin” “:=” <pinInOutMin>	3-51

Table 3-22. BNF Entries for Test Control Language (TCL).

BNF Entry	*Page*
PinList ::= “[” Pin { “,” Pin } “]” “;”	3-76
PinNum ::= <pinNumber> [“..” <pinNumber>]	3-76
PinOutMax ::= “pinoutmax” “:=” <pinOutMaximum>	3-51
PinRange ::= <lowerBound> “..” <upperBound>	3-32
Pins ::= { “pins” “:=” PinList }	3-76
Probe ::= (Probers | Settlers | Pulsers | Monitors | Spikes)	3-53
ProbeCloseHoldTime ::= “probeclosehold” “:=” Time	3-54
ProbeCloseSetupTime ::= “probeclosesetup” “:=” Time	3-54
ProbeConstraints ::= “probeconstraint” Subscript “:=” TimeExpr	3-53
ProbeDefault ::= “probedefault” “:=” TimeExpr	3-53
ProbeHold ::= “probehold” “:=” Time	3-53
ProbeOpenHoldTime ::= “probeopenhold” “:=” Time	3-54
ProbeOpenMaximum ::= “probeopenmax” “:=” Time	3-54
ProbeOpenMinimum ::= “probeopenmin” “:=” Time	3-54
ProbeOpenSetupTime ::= “probeopensetup” “:=” Time	3-54
Probers ::= (Probers1 | Probers2)	3-53
Probers1 ::= (ProbeConstraints | ProbeOpenMinimum | ProbeOpenMaximum | ProbeWindowMinimum)	3-53
Probers2 ::= (ProbeOpenSetupTime | ProbeOpenHoldTime | ProbeCloseSetupTime | ProbeCloseHoldTime | ProbeDefault | ProbeSetup | ProbeHold)	3-53
ProbeSetup ::= “probesetup” “:=” Time	3-53
ProbeWindowMinimum ::= “probewindowmin” “:=” Time	3-54
ProgControlDirectives ::= { “programcontrol” { ControlSpecs } “end” “programcontrol” }	3-10, 3-91

Table 3-22. BNF Entries for Test Control Language (TCL).

BNF Entry	*Page*
ProgramStructure ::= “structure” { StructureBody } “end” “structure”	3-93
PulseProbeCloseDefTime ::= “pulseprobeclosedefault” “:=” TimeExpr	3-54
PulseProbeOpenDefTime ::= “pulseprobeopendefault” “:=” TimeExpr	3-54
Pulsers ::= (PulseProbeOpenDefTime | PulseProbeCloseDefTime)	3-54
QuitOn ::= “quiton” “:=” <messageName> [“after” <integer>]	3-81
Range ::= IntRange [“pins” PinRange] [“per” <perValue>]	3-32, 3-34
RepeatAtEndLegal ::= “repeatatendlegal” “:=” Boolean	3-59
RepeatAtStartLegal ::= “repeatatstartlegal” “:=” Boolean	3-59
RepeatAtSubrEndLegal ::= “repeatatsubrendlegal” “:=” Boolean	3-59
RepeatAtSubrStartLegal ::= “repeatatsubrstartlegal” “:=” Boolean	3-59
RepeatCompression ::= “repeatcompression” “:=” Boolean	3-59
RepeatConstraint ::= (RepeatAtStartLegal | RepeatAtEndLegal | RepeatInSubrLegal | RepeatInLoopLegal | RepeatAtSubrStartLegal | RepeatAtSubrEndLegal)	3-59
RepeatCountMaximum ::= “repeatcountmax” “:=” <repeatCountMaximum>	3-59
RepeatCountMinimum ::= “repeatcountmin” “:=” <repeatCountMinimum>	3-59
RepeatInLoopLegal ::= “repeatinlooplegal” “:=” Boolean	3-59
RepeatInSubrLegal ::= “repeatinsubrlegal” “:=” Boolean	3-59
Repeats ::= (RepeatCompression | RepeatConstraint | RepeatCountMinimum | RepeatCountMaximum)	3-58
Repetition ::= “repetition” “:=” <repetitionCount>	3-78
ReportFormat ::= “format” “:=” (“tabular” | “linear” | “both”)	3-81
ReportWidth ::= “formwidth” “:=” (“80” | “132”)	3-81

Table 3-22. BNF Entries for Test Control Language (TCL).

BNF Entry	*Page*
RowMax ::= (“patternrowmax” | <rowMaximum>)	3-49
Scan ::= (ScanTimes | ScanPatterns | ScanType | ScanMode)	3-60
ScanChannelMax ::= “scanchannelmax” “:=” <scanChannelMax>	3-61
ScanCycleMax ::= “scancyclemax” “:=” Time	3-20, 3-60
ScanCycleMin ::= “scancyclemin” “:=” Time	3-20, 3-60
ScanCycleRes ::= “scancycleresolution” “:=” Time	3-20, 3-60
ScanMode ::= “scanmode” “:=” (“serial” | “parallel”)	3-61
ScanPatternMax ::= “scanpatternmax” “:=” <scanPatternMax>	3-60
ScanPatternMin ::= “scanpatternmin” “:=” <scanPatternMin>	3-60
ScanPatternRes ::= “scanpatternresolution” “:=” <scanResolution>	3-61
ScanPatterns ::= (ScanPatternMin | ScanPatternMax | ScanPatternRes | ScanChannelMax)	3-60
ScanRegistersOnly ::=”scanregistersonly” (“true” | “false”)	3-61
ScanTimes ::= (ScanCycleMax | ScanCycleMin | ScanCycleRes)	3-60
ScanType ::= “scantype” “:=” “[” ScanTypeName { “,” ScanTypeName } “]”	3-61
ScanTypeName ::= (“in” | “out” | “inout” | “mask” | “inmask” | “feedback”)	3-61
SettledProbeCloseDefTime ::= “settledprobeclosedefault” “:=” TimeExpr	3-54
SettledProbeOpenDefTime ::= “settledprobeopendefault” “:=” TimeExpr	3-54
Settlers ::= (SettledProbeOpenDefTime | SettledProbeCloseDefTime)	3-54
Severity ::= “severity” (“informative” | “warning” | “fatal”)	3-78
ShowFormatBlock ::= “showformatblock” “:=” Boolean “;”	3-93

Table 3-22. BNF Entries for Test Control Language (TCL).

BNF Entry	*Page*
SigList ::= SigName { “,” SigName }	3-105
SignalGroups ::= “signals” SignalName { “,” <signalName> }	3-92
SignalName ::= [‘ “ ’] <signalName> [‘ ” ’]	3-92
SigName ::= (<signalname> | <signalnameStr>)	3-105
Slot ::= <slotNumber> [“..” <slotNumber>]	3-76
SocketType ::= “sockettype” “:=” <socketTypeName>	3-28
SourceDB ::= (“directory” | “database”) [“modules”] “:=” (<dirNameAN> | <dirNameStr>) “;”	3-104
SourceBlock ::= “source” { SourceSpec } “end” “source”	3-104
SourceSets ::= (GreedyTGs | FormatUsageSet | TgUsageSet | StrobeUsageSet)	3-105
SourceSpec ::= (Source | StartTime | StopTime | Compress | IncrResAssign |SourceSets)	3-104
SpikePulseMinimum ::= “spikepulsemin” “:=” Time	3-54
Spikes ::= (SpikePulseMinimum)	3-54
StartTime ::= “start” “:=” TimeSpec “;”	3-84
StopTime ::= “stop” “:=” TimeSpec “;”	3-84
StrobeUsageSet ::= “strobeusage” “:=” “[” SigList “]” “[” NameList “]” “;”	3-105
StructureBody ::= Partition [“:=” ‘ “ ’ <fileName> ‘ ” ’ [PartitionReplace]] “;”	3-93
SubrAfterRepeatLegal ::= “subroutineafterrepeatlegal” “:=” Boolean	3-65
SubrAtEndLegal ::= “subroutineatendlegal” “:=” Boolean	3-65
SubrAtStartLegal ::= “subroutineatstartlegal” “:=” Boolean	3-65

Table 3-22. BNF Entries for Test Control Language (TCL).

BNF Entry	*Page*
SubrCallConstraint ::= (SubrAtStartLegal | SubrAtEndLegal | SubrAfterRepeatLegal | SubrSpacingMinimum)	3-65
SubrCompression ::= “subroutinecompression” “:=” Boolean	3-65
SubrDefnMaximum ::=”subroutinedefnmax” “:=” <suberDeferMaxer>	3-65
SubrNestMaximum ::= “subroutinenestmax” “:=” <subrNestMaximum>	3-65
Subroutine ::= (SubrCompression | SubrDefnMaximum | SubrNestMaximum | SubrRowMaximum | SubrRowMinimum | SubrRepeatMaximum | SubrCallConstraint)	3-65
SubrRepeatMaximum ::= “subroutinerepeatcountmax” “:=” <subrRepeatMaximum>	3-65
SubrRowMaximum ::= “subroutinerowmax” “:=” <subrRowMaximum>	3-65
SubrRowMinimum ::= “subroutinerowmin” “:=” <subrRowMinimum>	3-65
SubrSpacingMinimum ::= “subroutinespacingmin” “:=” <subrSpaceMin>	3-65
Subscript ::= [“[” <constIndex> “]”]	3-32
TclBody ::= (AteConstraints | PinGroups | MessageOverrides | TrcDirectives | MatchDirectives |ProgControlDirectives | PatternLoadDirectives)	3-10
TclProgram ::= “testcontrol” [<tclName>] { TclBody } “end” “testcontrol”	3-9
TgUsageSet ::= “tgusage” “:=” “[” SigList “]” “[” NameList “]” “;”	3-105
Time ::= (<intTime> | <floatTime>) [TimeUnit]	3-20, 3-28, 3-33, 3-33, 3-46, 3-54, 3-60, 3-84

Table 3-22. BNF Entries for Test Control Language (TCL).

BNF Entry	*Page*
TimeClockFormat ::= (“freq” | “time”)	3-93
TimeEdgeFormat ::= (“freq” | “time”)	3-93
TimePeriodFormat ::= (“freq” | “time”)	3-93
Timeplate ::= TimeplateName [“<=” TimeplateName]	3-84
TimeplateName ::= (<timeplateName> | <timeplateString>)	3-84
Timeplates ::= “timeplates” “:=” Timeplate { “,” Timeplate } “;”	3-84
TimeSet ::= “timesetmax” “:=” <timesetMaximum>	3-68
TimeSetMerging ::= (“true” | “false”)	3-93
TimeSetType ::= “timesettype” “:=” (“allow_both” | “force_single” | “force_dual”)	3-68
TimeSpec ::= (“begin” | “end” | “prevstop” | Time)	3-84
TimeUnit ::= (“ps” | “ns” | “us” | “ms” | “s”)	3-20, 3-28, 3-33, 3-33, 3-46, 3-54, 3-60, 3-85
TimingBlock ::= “timing” { TimingSpec } “end” “timing”	3-84
TimingSecondarySheetSets ::= “TimingSecondarySheetSets” “:=” EquationSheetSetName { “,” EquationSheetSetName } “;”	3-92
TimingSheetSets ::= “TimingSheetSets” “:=” EquationSheetSetName { “,” EquationSheetSetName } “;”	3-92
TimingSource ::= “directory” “:=” <WDBname> “;”	3-84
TimingSpec ::= (TimingSource | Timeplates | Persistence)	3-84
Transform ::= (TransformPattern | TransformTiming)	3-71

Table 3-22. BNF Entries for Test Control Language (TCL).

BNF Entry	*Page*
TransformPattern ::= “transformpattern” “:=” Boolean	3-71
TransformTiming ::= “transformtiming” “:=” Boolean	3-71
TrcDirectives ::= [“trc” { TrcSpec “;” } “end” “trc”]	3-10, 3-80
TrcSpec ::= (ReportFormat | ReportWidth | GlobalRepetition | IgnoreWhen | QuitOn)	3-80
VernierRange ::= “vernierrange” “:=” Time	3-28
WaveDestination ::= “directory” “:=” <WDBname> “;”	3-84
WaveSource ::= “directory” “:=” <directoryname> “;”	3-84

Index

N

P

R

S

T