Chapter 6 Waveform Generation Language

Introduction

Summit Design's Waveform Generation Language (WGL) is a data description language. It is used to convey an editable ASCII representation of the data contained in Summit Design's 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.

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. (See "Binary WGL" on page 6-108.)
- 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 Summit Design's TDS WGL In Converter and WGL Out Converter modules. See the "WGL In Converter", chapter, found in the *In Converters Guide*, and see the "WGL Out Converter" chapter, found in the *Out Converters Guide*, for details on how to use the WGL In Converter and the WGL Out Converter.

WGL and Wavemaker

Since WGL describes a WDB, the language necessarily reflects the structure of the WDB. If you have used Summit Design's 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 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.

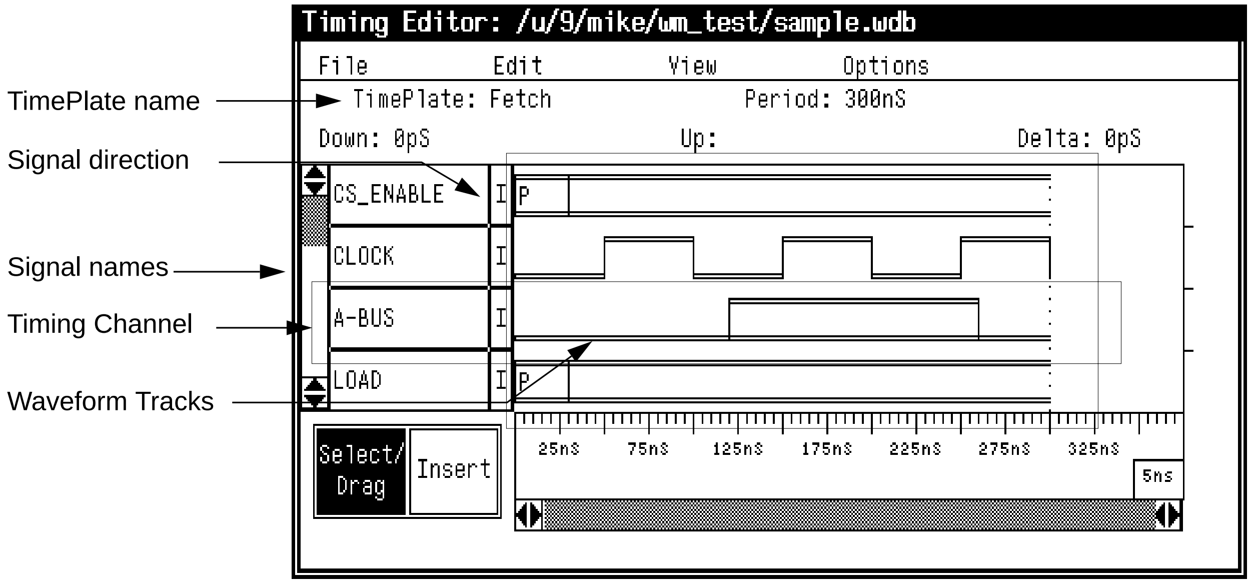


Figure 1. WaveMaker Timing Editor showing the TimePlate Fetch

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

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.

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.

1. Binary pattern files use the WGL syntax notation plus have additional notations. See “Binary File Format” on page 6-112.

- 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 (' ').
- 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 6-140.

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.

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 6-66.) 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

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

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

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)

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 (< >).

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 (< >).

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

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.

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\”\\”
```

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.

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

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

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 6-11.

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

Table 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

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	TimePlates
Scan Cells	Patterns
Scan State	Subroutines
Scan Chain	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.

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.

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 have optional strobe clauses following the direction attribute. Use this clause to specify that an input or output signal is valid only when another signal takes a certain value. The strobe clause uses the reserved word when to specify the condition to be met. This clause can also specify when a bidir signal has input direction and when it has output direction.

The <validityClause> portion of the construct must meet the syntax requirements of the TDS Signal Definition file. (See the “User-Defined Files” chapter, found in this guide, for details on the Signal Definition file.)

The following is an example of a WGL Signals block with strobe clause for signals `dr` and `data`:

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 6-20 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.

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.

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 6-102 of this chapter.

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. See “Binary WGL” on page 6-108. 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> } “)”

The *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 the “Test Control Language” chapter, found 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.

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 6-102 of this chapter.

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 ::= (OutEdgeSignalOnly | ChainMemList)

OutEdgeSignalOnly ::= “, ” ChainMemReference

ChainMemList ::= ChainMemReference { “ , ” ChainMemReference }

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. When signal names appear in a *ChainMembers* list, the signal names must be the first or last entry in the list.

A signal name appearing at the start of the chain must have been declared input or bidirectional. A signal appearing at the end must have been output or bidirectional. The reserved symbol ! indicates state inversion. Scan chains may be members of other chains as long as the declaration is not recursive.

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

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 defined previously 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 6-102 of this chapter.

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. (See “ExprSet” on page 6-57.)

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. (See “ExprSet” on page 6-57.)

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*. 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*. Conceptually, a *Channel* is a container for one or more signal names, each of which is followed by a *Track*. The *Track* itself contains the actual information about the shape and timing of the waveform, and its *Direction*. The *TDSstate*s that are used must be consistent with those available for the direction of input or output. (See Table 7 on page 6-75 for a list of TDS state characters.) All the signals that share the channel must also 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. (See “ExprSet” on page 6-57.)

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 *TDSstate*. 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 7 on page 6-75 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 6-38.

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 6-38.

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 6-30.) 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. (See "ExprSet" on page 6-57.)

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

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. See “Binary WGL” on page 6-108. 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.

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 are used to describe a way of partitioning 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 6-34.

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 TDS WGL In Converter. These fields are provided for compatibility with the TDS 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 6-18), 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 6-29.

It is important to make certain that signals that terminate scan chains have the proper state character supplied to them, as described on page 6-37 , 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 6-45, 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 6-102 of this chapter.

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

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” | “symbolic”)

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 *stateString*s. 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. See "Scan State" on page 6-27 for more information about *stateString*s.

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

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:

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.

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 6-6. 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 6-8) 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 6-57.) The <exprSetName> identifier must conform to the naming conventions for identifiers, as described in “Identifiers” on page 6-6.

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

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 6-6. 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 6-8) 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 6-6. See “Tester-Specific Program Blocks” on page 6-74 and “TimingSets” on page 6-81 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.

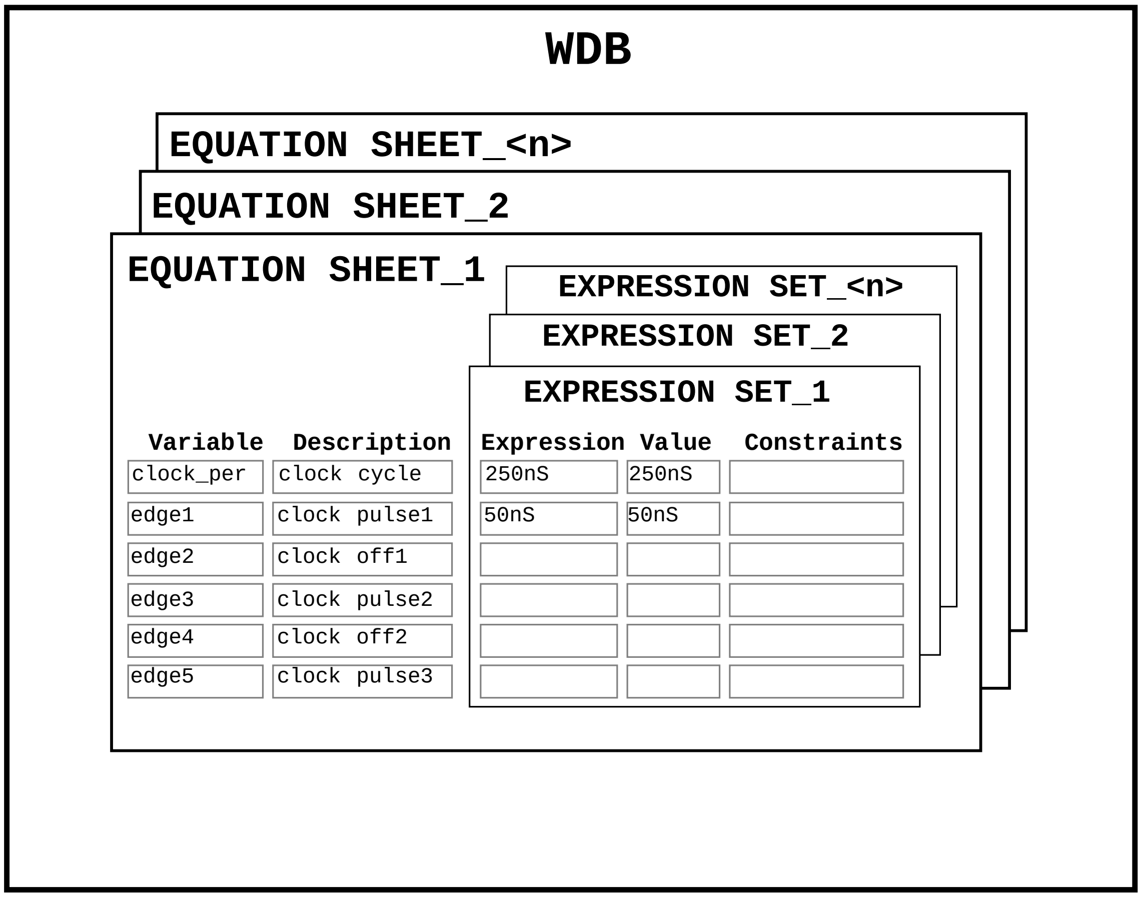
All variable declarations within an EquationSheet block are unique to that EquationSheet block. A variable of the same name cannot be declared in another EquationSheet block, but it can be declared again in another ExprSet sub-block contained in the same EquationSheet block. In fact, that is the main 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:



Variable	Description	Expression	Value	Constraints
clock_per	clock cycle	250nS	250nS	
edge1	clock pulse1	50nS	50nS	
edge2	clock off1			
edge3	clock pulse2			
edge4	clock off2			
edge5	clock pulse3			

Figure 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 6-69.

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 6-8) 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 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. 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 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 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

Table 4. Built-in Functions (Continued)

WGL BuiltInFunc	Performs Operation
FLOOR	floor (round down to integer)
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 6-92.

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 6-68) 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 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 6-68.

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 6. Units of Measurement

WGL Notation	Unit
A	ampere
H	hertz
S	Second
V	volt

You can add a scaling factor to modify the basic units of measurement, as described in “Scaling” on page 6-67.

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

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 below:

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

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.

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

Table 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

Table 7. TDS logic states (Continued)

TDS Logic State Characters	Meaning
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 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 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
```

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

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 the “Test Control Language” chapter, found 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 9.

Table 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 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 pinof a signal in the Signals block.

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

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 an existing timing generator name (having been defined in the TimeGens block, see “TimeGens” on page 6-80), followed by an optional numeric value for edge number enclosed in brackets ([ ]), followed by an assignment operator (:=), followed by a numeric value for time expressed in a supported unit of measurement or a variable having been previously defined in the ExprSet sub-block of an EquationSheet block. (See “ExprSet” on page 6-57.)

NOTE

A variable 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 6-57.)

You can also substitute variables for the literal time value associated with a previously defined timing generator. (See “TimeGens” on page 6-80.) 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

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.

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.

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 6-100.) 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.

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> }

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
```

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
```

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:

```
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.

1. Binary pattern files cannot be included in the WGL program via an Include file statement . See "Binary WGL" on page 6-108 for information on how to include binary formatted files in a WGL file.

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
```

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 6-105.

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.

pmode Attribute

The pmode attribute defines the state value of the first cycle for those cycles that adopt their state value from the previous cycle (the P Mode). This feature permits you to tailor the initial state value of waveforms that, by default, derive their initial state value from the previous cycle.

Table 10 defines the supported pmode attribute options. Refer to Table 7, on page 6-75, for a complete list of TDS state characters.

Table 10. P Mode definitions

P Mode Setting	P is Replaced by	Definition
Previous Force `(P_LAST_FORCE)`	a force state (D, U, N, or Z)	If force pattern data for the cycle (associated with the same signal) is Z: P is replaced by Z. If force pattern data for the cycle (same signal) is not Z: P is replaced by the last force state value on the same signal (D, U, N, or Z), whether the previous force state is itself a result of substitution, or is a fixed value.
Previous Driving `(P_LAST_DRIVE)`	D, U, or Z	If force pattern data for the cycle (associated with the same signal) is Z: P is replaced by Z. If force pattern data for the cycle (same signal) is not Z: P is replaced by the last D or U state on the same signal, whether the previous force state is itself a result of substitution, or is a fixed value.
Previous, if Force, else Z `(P_FORCE_OR_Z)`	last force state value, else Z	P is replaced by the last state value on the same signal, if the last state value is force (D, U, or N) or monitor (d, u, or n). If the previous state value is other than the above, P is replaced by Z.
Advantest `(P_ADVANTEST)`	a force state	If force pattern data for the cycle (associated with the same signal) is Z: P is replaced by Z. If force pattern data for the cycle (same signal) is not Z: P is replaced by the previous force state if that state is D, U, N, or Z. P is replaced by D if the previous state is L or T. P is replaced by U if the previous state is H or X, but ignores previous X states that follow force states and are not at the start of the cycle.

Table 10. P Mode definitions (Continued)

P Mode Setting	P is Replaced by	Definition
IMS `(P_IMS)`	last force state value, else Z	**For scalar (non-multiplexed) signals,** P is replaced by the last state value on the same signal, if the last state value is force (D, U, or N) or monitor (d, u, or n). If the previous state value is other than the above, 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 `(P_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

Examples

Using WGL Macros and Include Files to Simplify Testing

The following examples illustrate the use of include files and macros in a WGL program used to generate a test for a microprocessor. The WGL program in `example_Test_Chip.wgl` contains only the beginning and ending statements and four include invocations.

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
#---------------------------------------------------------------------------
signal   AS        : 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

Using WGL to Support Scan Test Hardware

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

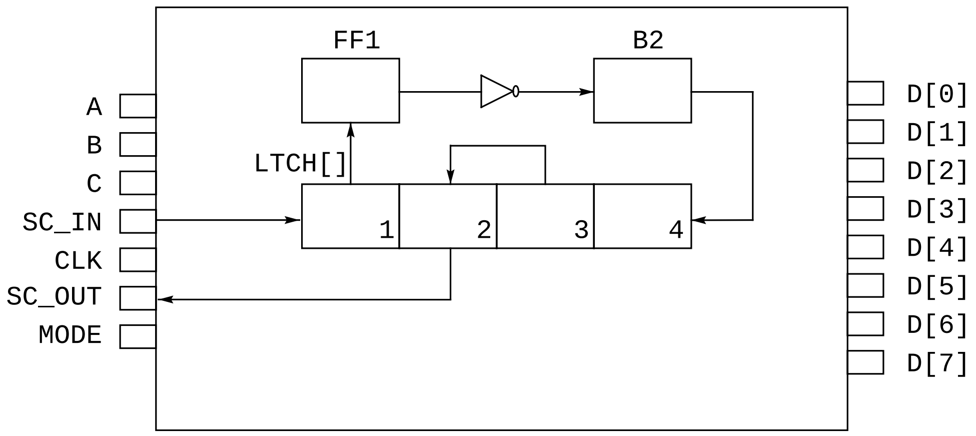


Figure 3. Example device with scan hardware

The device in Figure 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
         .   .   .
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.
```

End Example

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.

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

In the following example, the original WGL file, `anno.wgl`, has been converted to a WDB and then converted back to a WGL file, using the WGL Out Converter. Note that all annotations labeled `{ lost }` are no longer preserved in the database.

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

Binary WGL

A binary format of the pattern vectors, to be used in place of ASCII pattern data, 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

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

vector row in the pattern 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.)

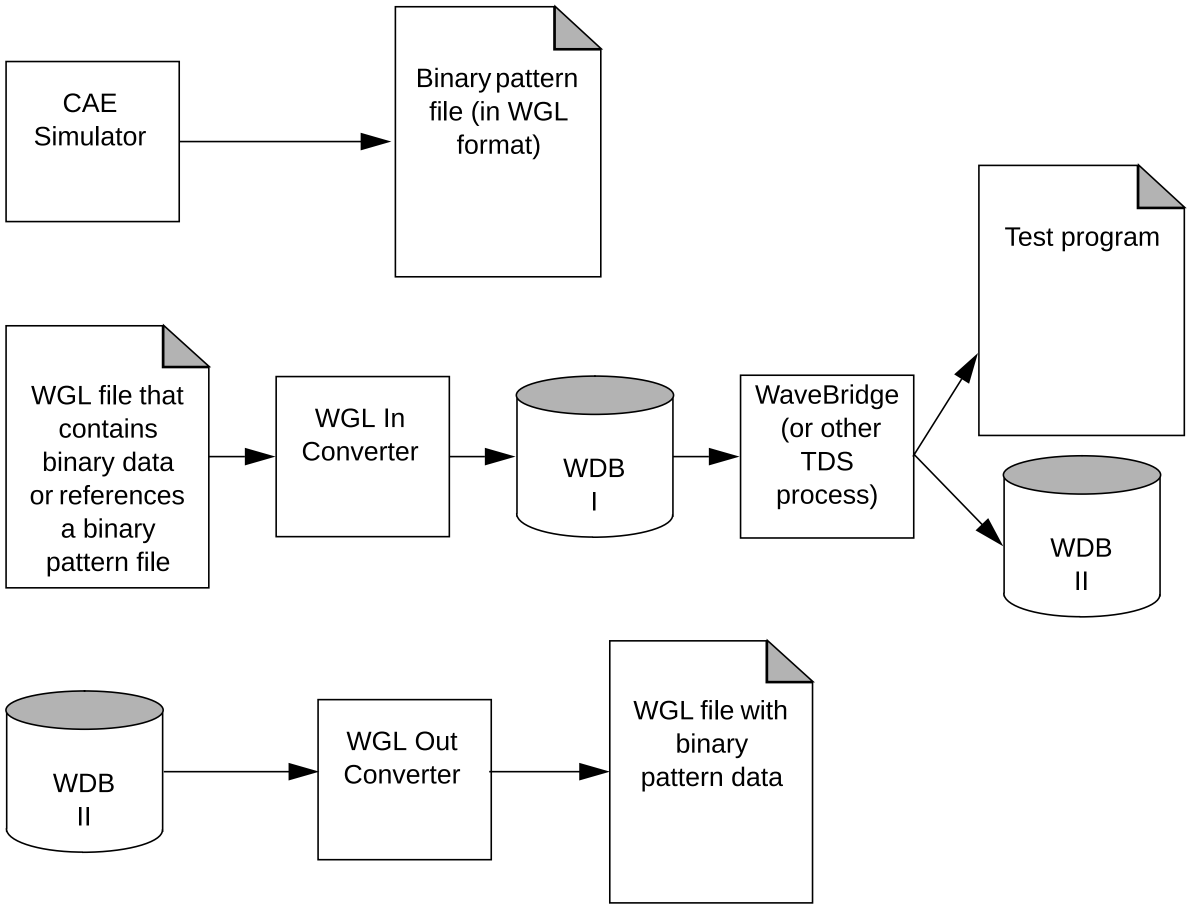

CAE Simulator
Binary pattern file (in WGL format)
Test program
WGL file that contains binary data or references a binary pattern file
WGL In Converter
WDB I
WaveBridge (or other TDS process)
WDB II
WDB II
WGL Out Converter
WGL file with binary pattern data

Figure 4. Using Binary Pattern Data

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

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

BinaryPattern file command, not via an Include file statement. (You cannot mix ASCII pattern vectors with binary pattern data.)

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.

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

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”.

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.

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 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

Line Format

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

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

Table 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 13.)
rest-of-line		Varies depending on the line type (See Table 14 through Table 32.)

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

Line Type

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

Table 13. Hexadecimal Values for Each Line Type

Hexadecimal	Line Type
0x0000	Vector Line
0x0001	Subroutine

Table 13. Hexadecimal Values for Each Line Type (Continued)

Hexadecimal	Line Type
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

Line Type Ordering

The binary pattern information must follow the same ordering restrictions required by ASCII WGL. (See “Patterns” on page 6-38.) 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.

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 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 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 WG, 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 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 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 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 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 (see “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 20 through Table 23.) Different map keys can be used for different pattern lines within the same file. For example, use map key 3 (Table 23) for all vector and scan pattern row lines and use map key 2 (Table 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 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 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 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 23. Map Key 3: General Mapping (map_key = 0x03)

Character	Bit Map
0	00
1	01
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 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 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 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 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 6-127 for state and chain information.)

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

Table 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
map_key	byte	Selects the map key
vector	[a]	Parallel vector data

a. Defined by map_key (see “Map Key” on page 6-121). 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 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 (see “Map Key” on page 6-121). 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 6-131 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 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 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 32. End Binary Line Type

Item	Type	Description
line_type	short	0x000f

Binary format:

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

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

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

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

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 6-100.)

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. (See the "User-Defined Files" chapter, found in this guide, for the syntax requirements of the Signal Definition file.) 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.