



# Velocity CAE<sup>®</sup> Program Generator

---

For **Simulation to ATE** and  
**ATE to ATE Conversion**

Release 8.0.0.0

# Configuration Guide

---

---

# Velocity CAE Program Generator Configuration Guide

---

## **COPYRIGHT NOTICE**

Copyright © 2016 Alliance ATE Consulting Group, Inc.



All rights reserved

Documentation version 3.0

Any technical documentation that is made available by Alliance ATE Consulting Group is the copyrighted work of Alliance ATE Consulting Group and is owned by Alliance ATE Consulting Group.

**NO WARRANTY.** The technical documentation is being delivered to you AS-IS and Alliance ATE Consulting Group makes no warranty as to its accuracy or use. Any use of the technical documentation or the information contained therein is at the risk of the user. Documentation may contain technical or other inaccuracies or typographical errors. Alliance ATE Consulting Group, Inc. reserves the right to make change without prior notice.

No part of this publication may be copied without the express written permission of Alliance ATE Consulting Group, 3080 Olcott St Suite 110C, Santa Clara, CA 95054.

## **TRADEMARKS**

Velocity CAE Program Generator, and ShellConstructor are trademarks of Alliance ATE Consulting Group.

SmarTest, 93000, and 93K are trademarks of Advantest Corporation.



---

OBSERVATIONS:

1. In the **PINS** definition:
  - a. The symbols *pinList*, *startStopList*, *condition*, and *map* are high-level syntactical elements that are subsequently broken down into smaller elements.
  - b. The use of regular style square brackets around *condition* and *map* means that they are optional.
2. In the *pinList* definition (*pinList*::=):
  - a. The symbol *pinList* is defined as a comma-separated list of elements in which each element can be either a *pinName* or a *groupName*.
  - b. Note the use of the regular style vertical bar to indicate a choice of either *pinName* or *groupName*.
  - c. Note the use of the regular style braces to indicate 0 or more additional *pinName* or *groupName* elements, each preceded by a required comma.
  - d. The fact that the first occurrence of *pinName|groupName* is not enclosed in square brackets or braces means that at least one element must be specified. Any others are optional.
3. In the *condition* definition (*condition*::=):
  - a. The use of bold style braces means that braces are to be typed as a required part of the syntax.
4. In the *conditionList* definition (*conditionList*::=):
  - a. Note that the symbol *relativeCycle* is enclosed in two sets of square brackets.
  - b. The innermost brackets are in bold font, indicating that square brackets are to be typed as a required part of the syntax.
  - c. The outermost brackets are in regular font, indicating that the element within is optional.

---

# CONFIGURATION GUIDE

## TABLE OF CONTENTS

Page #

### Contents

<i>Copyright Notice</i> .....	<i>ii</i>
<i>Trademarks</i> .....	<i>ii</i>
<b>1.0 GENERAL INFORMATION</b> .....	<b>1-1</b>
<b>1.1 What is a Configuration File?</b> .....	<b>1-2</b>
<b>1.2 Creating a Configuration File</b> .....	<b>1-2</b>
1.2.1 Automatically Generating an Initial Configuration File .....	1-3
<b>1.3 What Happens During the Conversion Process?</b> .....	<b>1-4</b>
1.3.1 Conversion Process Inputs .....	1-4
<b>1.3.2 “Cyclized” vs. “Uncyclized” Pattern Formats</b> .....	<b>1-5</b>
<b>2.0 CONFIGURATION FILE STRUCTURE</b> .....	<b>2-8</b>
<b>2.1 Syntactic Elements</b> .....	<b>2-1</b>
<b>2.2 Control Definitions</b> .....	<b>2-1</b>
2.2.1 Single-line.....	2-1
2.2.2 Multi-line .....	2-1
2.2.3 Comments .....	2-2
2.2.4 Keywords.....	2-2
2.2.5 Parameters.....	2-3
2.2.6 Use of Whitespace .....	2-3
<b>2.3 Line-Oriented Structure</b> .....	<b>2-4</b>
<b>2.4 List of Control Types</b> .....	<b>2-5</b>
<b>2.5 Order of Control Definitions</b> .....	<b>2-6</b>
<b>2.3 Example Configuration File</b> .....	<b>2-7</b>
<b>3.0 CONTROL DEFINITION REFERENCE</b> .....	<b>3-1</b>
<b>3.1 Environment Definitions</b> .....	<b>3-1</b>
3.1.1 PATH.....	3-1
3.1.2 DEVICE.....	3-1
3.1.3 PROGRAM.....	3-2
<b>3.2 General Build Definitions</b> .....	<b>3-3</b>
3.2.1 SOURCE_PORT Definition .....	3-3

3.2.2	TARGET_PORT Definition .....	3-4
3.2.3	LIBRARY .....	3-5
3.2.4	HEADER .....	3-6
3.3	Pin Configuration Definitions.....	3-7
3.3.1	PINLIST Definition .....	3-7
3.3.2	BIDIRECTIONAL CONTROL.....	3-9
3.3.3	GROUP Definition .....	3-10
<b>3.4</b>	<b>Source-Port-Specific Variables .....</b>	<b>3-12</b>
3.4.1	IGNORE TIMEPLATE .....	3-12
3.4.2	JOB .....	3-12
3.4.3	MACROSTYLE .....	3-13
3.4.4	MASTER.....	3-14
3.4.5	PERSISTENCE .....	3-14
3.4.6	SYNC.....	3-15
3.4.7	UNDERSAMPLE.....	3-15
3.4.8	OVERSAMPLE.....	3-16
3.4.9	VCDPAGE .....	3-17
<b>3.5</b>	<b>Target-Port-Specific Variables.....</b>	<b>3-18</b>
3.5.1	BINARY .....	3-18
3.5.2	COMBINATION.....	3-19
3.5.3	CONTEXT.....	3-20
3.5.4	CONFIGURATION (PIN FILE).....	3-20
3.5.5	CTIM .....	3-21
3.5.6	DOMAIN TRACKING.....	3-21
3.5.7	FASTMODE.....	3-22
3.5.8	FSPINS.....	3-22
3.5.9	MEMORY .....	3-23
3.5.10	METHOD .....	3-23
3.5.11	MODEL .....	3-24
3.5.12	STATEMAP .....	3-24
<b>3.6</b>	<b>Verilog Feedback Variables.....</b>	<b>3-25</b>
3.6.1	MAXDELTA Definition.....	3-25
3.6.2	MODULE .....	3-25
3.6.3	VERILOG Definition .....	3-26
3.6.4	WINDOW.....	3-26
3.6.5	FEEDBACK FILTER.....	3-27
<b>3.7</b>	<b>General Purpose Variables .....</b>	<b>3-29</b>
3.7.1	COMMENT .....	3-29
3.7.2	DDRMODE .....	3-30
3.7.3	DELAY.....	3-31
3.7.4	INIT .....	3-32
3.7.5	PAGE.....	3-33
3.7.6	INIT .....	3-34
3.7.7	SUBROUTINE .....	3-35
3.7.8	TERMINATION.....	3-36
3.7.9	TRISTATE .....	3-37
3.7.10	WARNINGS .....	3-37
3.7.11	MINREPEAT .....	3-38
3.7.12	MAXLOOP.....	3-38
3.7.13	DEBUG.....	3-39
3.7.13	BURST.....	3-40
<b>3.8</b>	<b>Timing Variables .....</b>	<b>3-41</b>
3.8.1	CAPTURE .....	3-42

---

3.8.2	DATARATE.....	3-43
3.8.3	EDGES .....	3-44
3.8.4	EQUATION.....	3-45
3.8.5	NORMALIZE.....	3-45
3.8.6	OPTIMIZE.....	3-46
3.8.7	PERIOD.....	3-47
3.8.8	DRIVE Block.....	3-49
3.8.9	STROBE Block .....	3-50
3.8.10	SURROUND.....	3-51
3.8.11	RUNNINGCLOCKS .....	3-51
3.8.12	GLOBALSPEC .....	3-52
<b>4.0</b>	<b>Custom Levels .....</b>	<b>4-53</b>
4.0.2	Power Sequences.....	4-54
4.0.3	Power down sequencing .....	4-55
<b>5.0</b>	<b>Custom Timing.....</b>	<b>5-56</b>
5.01	TEST Definitions .....	5-58
5.02	FLOW Definitions .....	5-65
<b>6.0</b>	<b>CUSTOMIZING PATTERNS.....</b>	<b>6-66</b>
6.1	Pattern Syntax .....	6-2
6.2	BASE Syntax.....	6-2
6.3	Command and Parameter List Syntax .....	6-3
6.3.1	TYPE ( <i>optional</i> ).....	6-3
6.3.2	DOMAIN ( <i>optional</i> ) .....	6-3
6.3.3	FUNC ( <i>optional</i> ) .....	6-4
6.3.4	LOOP .....	6-4
6.3.5	REPEAT.....	6-5
6.3.6	MATCH .....	6-5
6.3.7	START .....	6-5
6.3.8	STOP.....	6-5
6.3.9	WAIT .....	6-6
6.3.10	CUSTOM PATTERN EXAMPLES .....	6-7
6.4	Logical Masking.....	6-8
6.4.1	PINS.....	6-9
6.4.2	MAPS .....	6-9
6.4.3	CONDITIONS .....	6-11
6.4.4	PIN DUPLICATION .....	6-12
6.4.5	EXAMPLES: .....	6-12
6.5	Serial Masking .....	6-13
6.5.1	SYNTAX.....	6-13
6.5.2	EXAMPLES .....	6-14

## 1.0 GENERAL INFORMATION

*A brief look at what a Velocity CAE configuration file entails and how it is create and used.*

## 1.1 What is a Configuration File?

A Configuration File is a human-readable, ASCII text file used by Velocity to control the conversion process.

Some of the aspects of the conversion process that a Configuration File controls are:

- The directory into which files generated by the conversion are to be written
- The period by which a VCD pattern is to be divided into cycles
- The target pin list, including test system resource assignments
- Pin groups
- Custom timing
- Custom levels
- Rules for creating custom patterns from existing patterns
- Standardized test and power up/down definitions
- Test flow

Every Velocity conversion – whether the ShellConstructor or Design-to-Test (D2T) or Tester-to-Tester (T2T) – requires the use of a Configuration File.

If the user does not specify a Configuration File and attempts to run a conversion, Velocity will display the following error message:



Configuration Files can be given any name, within the limitations of the host operating system. But, all names use a .cfg extension. They can reside in any directory that the user chooses.

## 1.2 Creating a Configuration File

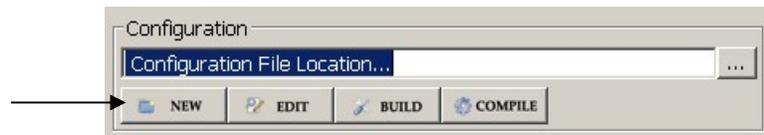
As a human-readable, ASCII text file, a Configuration File can be created and edited using any text editor. The user may choose to start from nothing and create the entire Configuration File in the text editor; or, use an existing file as a template and edit those elements which differ.

As an alternative, Velocity offers a way to speed up the Configuration File creation process. The Velocity GUI can quickly and automatically generate an initial Configuration File from an existing pattern file that is to be converted.

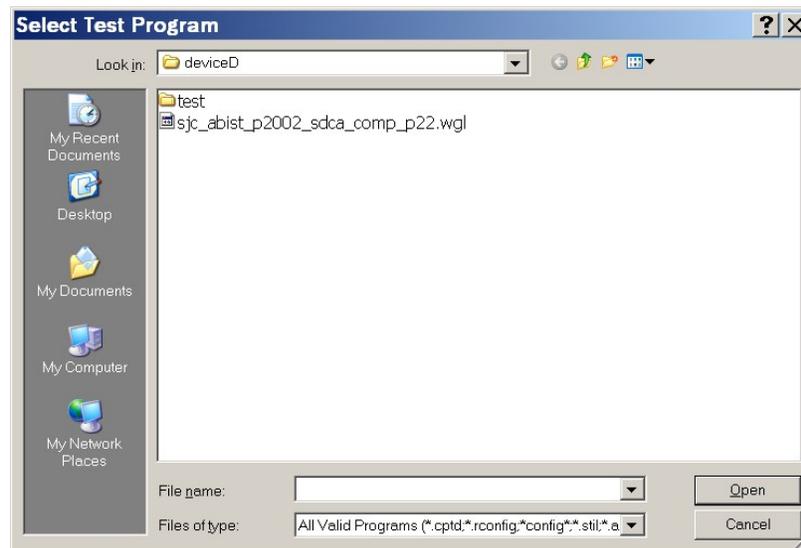
The automatic process will create a file containing, at a minimum, the definition of the target file path and the pin list. The user can then add any other required elements in the text editor.

### 1.2.1 Automatically Generating an Initial Configuration File

- From the GUI Configuration menu, select New.



- A window similar to the following will appear.



- Navigate to the directory containing the simulation output files or ATE files from which you want to build a test program.
- Select any one file which, at a minimum, defines all of the required pins to be used in the test program.
- Click the Open button. A progress indicator window will pop up, following by a completion message, similar to the one shown next:



- Note the location of the new Configuration file, as shown in the message. Click the OK button to acknowledge.

## 1.3 What Happens During the Conversion Process?

In order to better understand the aspects of the pattern conversion process that are controlled by the Configuration File, it is useful to have a basic understanding of what happens during conversion.

### 1.3.1 Conversion Process Inputs

Every Velocity conversion takes, as input, one or more pattern files of one of the following supported types:

- STIL
- VCD/EVCD
- WGL
- VCT
- CPTD (Credence ASL3000)
- XLS/ATP (Teradyne J750)
- XLS/ATP (Teradyne UltraFlex)
- ADR (Teradyne J973)
- AVC/DVC (Advantest 93000)

### 1.3.2 “Cyclized” vs. “Uncyclized” Pattern Formats

ATE test systems output functional stimulus to the device (and sample functional responses from the device) in the form of a vector sequence. The vectors are presented at a particular rate defined by the **cycle time** (also known as the **period**).

The following are excerpts from a STIL pattern file and timing file, respectively, showing how digital pattern sequences and corresponding cycle timing are represented in an ATE environment:

```

////////////////////////////////////
//   Pattern Block: example_vectors
////////////////////////////////////
Pattern example_vectors {
Start_example_vectors:
    W "tps66000_10000";
    V { all = 0XXXXXXXXXXXXXXXXXXXXX00XXXXXX1XXXXXXXX1XXXXX0X00X0XXXX1X; }
//0
    V { all = 0XXXXXXXXXXXXXXXXXXXXX00XXXXXX1XXXXXXXX1XX0XX0X00X0XXXX1X; }
//1

```

```

////////////////////////////////////
//   Timing Blocks
////////////////////////////////////
Timing "customTiming" {
    WaveformTable "tps66000_10000" {
        Period 'PERIOD';
        Waveforms {
            "addr[10]" {
                01Z { '0.000*PERIOD' D/U/Z; }
                LHXM { '0.091*PERIOD' L/H/X/T; }
            }
            "addr[11]" {
                01Z { '0.000*PERIOD' D/U/Z; }
                LHXM { '0.091*PERIOD' L/H/X/T; }
            }
        }
    }
}

```

Many other simulation and test data formats, such as WGL (Waveform Generation Language), also have a concept of vectors and cycle times, which can be translated to tester independent STIL format in a relatively straightforward manner. These kinds of pattern formats can be categorized as **cyclized** formats.

The following are excerpts from a WGL file, showing how digital pattern sequences and cycle timing, corresponding to the STIL example above, are represented in a WGL format:

```

pattern Chain_Scan_test("extal", "dft_setup", "dft_atpg", "dft_shift",
...
{ Pattern 0 Cycle 0 Loop 0 }
vector(+, tps66000_10000) := [ 0 0 0 0 0 0 0 - - - - -
...
{ Chain_test }

```

```

    { Pattern 0 Cycle 1 Loop 1 }
    { Begin chain test }
    repeat 6      vector(+, tps66000_10000) := [ 0 1 0 0 0 0 0 - - - - -
- -

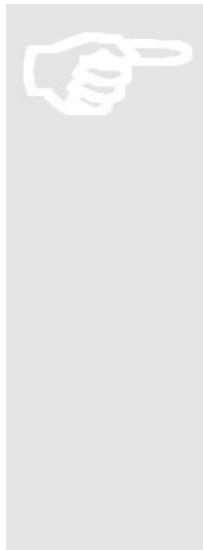
```

```

timeplate tps66000_10000 period 66000ps
...
"addr[10]" := input[0ps:S];
"addr[11]" := input[0ps:S];
...
"addr[10]" := output[0ps:X, 6000ps:Q'edge];
"addr[11]" := output[0ps:X, 6000ps:Q'edge];

```

## OBSERVATIONS:



1. In the above comparison of STIL and WGL formats, the pins were not defined in the same order; so, the vector columns will not match up. However, the same underlying vector data, per pin, would be contained in each format.
2. In the STIL example on the previous page, note how vectors and cycle timing are brought together by preceding a sequence of vector lines (those lines that begin with “V”) with a waveform table selection line beginning with “W”. The waveform table specified after the “W” is defined within the Timing Block shown on the same page.
3. In the WGL example above, cycle timing is defined within a **timeplate** definition, and then brought together with vectors in individual vector lines (those lines containing the keyword **vector**), by referencing the **timeplate** name.

Not all pattern formats are cyclized. The most notable examples of **non-cyclized** formats are the VCD (Value Change Dump) and EVCD (Extended Value Change Dump) formats. In these non-cyclized formats, signal patterns are represented as a continuous stream of events, where an event is a change of state at a particular point in time relative to the beginning of the pattern.

The following is an excerpt from an example VCD file:

```

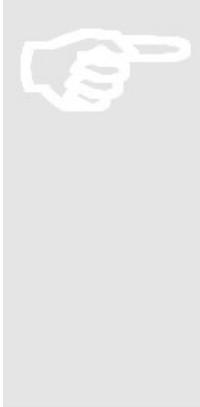
...
#1000
pT 0 0 <262
pT 0 0 <263
pX 6 0 <265
pX 6 0 <266
pL 6 0 <267

#3000
pb 6 6 <9

```

```
pb 6 6 <10
pb 6 6 <11
pb 6 6 <12

#4000
pN 6 6 <96
pN 6 6 <97
...
```



### OBSERVATIONS:

1. The lines beginning with “#” are timestamps, with the time unit being specified previously in the file with the \$timescale statement. (In this example, the time unit is 1ps; so, 1000 represents 1ns.)
2. Following each timestamp line is a sequence of value change lines, one for each signal which changes state at that timestamp. (Signals which do not change state at that timestamp are not listed.)
3. The first field of each value change line is the state to which the signal changes. The fourth field is an arbitrary, user-defined symbol for a specific signal.

For VCD, Velocity will analyze the spacing of timing events for each signal, and determine a best-fit tester cycle time and edge delays for your test program.

## **2.0 CONFIGURATION FILE STRUCTURE**

*Information on the structure and syntax of a Velocity Configuration File.*

## 2.1 Syntactic Elements

Configuration Files are made up of a number of different types of syntactic elements.

At the top level, there are two main types of elements. These types are:

- **Control Definitions**, which define particular aspects of the conversion and program generation process; and,
- **comments**, which begin with the '#' symbol and continue to the end of the line.

## 2.2 Control Definitions

Control Definitions can be categorized into two forms: **single-line** and **multi-line**.

### 2.2.1 Single-line

A single-line definition begins with a **keyword**, includes one or more **parameters**, and continues to the end of the line or to the beginning of a comment, whichever comes first.

The following **PERIOD** definition is an example of a single-line Control Definition:

```
PERIOD      5.000ns default
```

In this example, the keyword is **PERIOD**, and the two parameters are **5.000ns** (the value of the target period for cyclization) and **default** (the name given to this particular target period, or *Clock Domain*).

### 2.2.2 Multi-line

A multi-line definition (also called a **block**) consists of a **starting line**, zero or more **sub-parameter lines**, and an **ending line**.

- **Starting Line**

The starting line begins with a **keyword** and includes zero or more **parameters**.

- **Sub-parameter Line**

A sub-parameter line consists of one or more keywords and/or user-defined symbols or values whose order depends on the type of Control Definition. Each line provides further details in the definition of the Control.

- **Ending Line**

The ending line consists of the keyword **END** followed by the starting line keyword.

The following **PINLIST** block definition is an example of a multi-line Control Definition:

```
PINLIST
    ANALOG_VDD          default IO    ANALOG_VDD
    CVDD                default IO    CVDD
    HOLDn              default IO    HOLDn
END PINLIST
```

Note that the block begins with a starting line consisting only of the keyword **PINLIST** and ends with an ending line consisting of **END PINLIST**. In between are sub-parameter lines that begin with a pin name and consist of several parameters that define properties of the pin.

### 2.2.3 Comments

Comments can appear anywhere within the Configuration File, with the following restrictions:

- They only extend to the end of the line. Multi-line comments require a separate starting “#” for each line.
- Everything from the starting “#” to the end of the line is part of the comment. No part of a Control Definition will be recognized by Velocity if placed after the “#”.
- If a comment is placed at the end of a Control Definition line, the starting “#” must be separated from the last Control Definition line character by whitespace. (See below for more information on the use of whitespace in Configuration Files.)

The following is an example of a multi-line comment in a Configuration File, with the comment on each line taking up the entire line:

```
#####
## PinList Definition
#####
```

The following is an example of a comment at the end of a Control Definition line (in this case, the starting line of a TEST definition block):

```
TEST contNegative 150 # Continuity test with negative forcing current
```

### 2.2.4 Keywords

Keywords are Velocity reserved words. That is, they may not be used for user-defined names, such as ClockDomain names, Pin names, and Pattern names.

Keywords are NOT case-sensitive. For example, Velocity would interpret **period** the same as **PERIOD** or, even **pErIoD**. However, for readability purposes and for establishing a standard convention, it is recommended that all keywords be in **UPPER-CASE**.

### 2.2.5 Parameters

Parameters are elements of a Control Definition that allow the user to provide details for a particular instance of the Control. The user does so by giving a user-defined symbol or value, called an **argument**, at the corresponding parameter location.

For example, the first parameter in the starting line of the TEST block definition is the test name. In the example above, the argument for that parameter is “contNegative”.

Arguments for parameters ARE case-sensitive. So, a later test flow definition referencing the TEST called “contNegative” would have to specify the exact same case.

### 2.2.6 Use of Whitespace

Whitespace in a Configuration File includes spaces and tabs.

A Configuration File may contain any amount of whitespace at the beginning and end of lines, and between keywords, parameters, and comments. Some parameters, such as the pin list of a PINS masking definition, can be specified with multiple sub-elements separated by a non-whitespace character. The following example shows a PINS sub-parameter definition within a PATTERN block definition:

```
PATTERN func_pat_masked
  PINS Q0,Q1,Q2,Q3 55-83
END PATTERN
```

Note that the pin list, “Q0,Q1,Q2,Q3”, is considered the argument to one parameter of the PINS definition. Therefore, it contains no embedded whitespace. The individual sub-elements (Pins in this case) are separated only by commas. Likewise, the cycle range parameter is made up of a start and stop address separated by a hyphen.

## **2.3 Line-Oriented Structure**

The main elements of a Configuration File – Control Definitions and Comments – follow, for the most part, a line-oriented structure. That is, the end-of-line (i.e. carriage return) marks the end of:

- Single-line Control Definitions;
- Starting and Ending lines of Multi-line Control Definitions;
- Sub-parameter lines of Multi-line Control Definitions (with exceptions noted below); and,
- Comments.

The only exceptions to the end-of-line termination are the masking sub-parameter definitions – ON, OFF, and PINS – of a PATTERN block definition. Those sub-parameter definitions are terminated by a semicolon (;) and are allowed to extend to multiple lines. This feature allows for long, complex masking definitions. Refer to the detailed description of the PATTERN block syntax later in this guide.

Also, as noted in the previous section of this guide on Comments, a Control Definition line may be terminated by the beginning of a Comment on the same line.

## 2.4 List of Control Types

The following table lists all of the available Control Types for a Configuration File, along with a brief description.

Control Type	Description
<b>PATH</b>	Base directory path for test program files
<b>DEVICE</b>	Sub-directory of path specified by PATH Control, used to hold test program files for a specific device
<b>PROGRAM</b>	Base file name used for various timing, levels, and pattern files and subdirectories
<b>SOURCE_PORT</b>	Default value for the Source port. This will cause the source port to be automatically defined when the Configuration File is loaded. Valid entries can be chosen from anything that is present in the Source Port drop down list in the GUI.
<b>TARGET_PORT</b>	Default value for the Target port. This will cause the target port to be automatically defined when the Configuration File is loaded. Valid entries can be chosen from anything that is present in the Target Port drop down list in the GUI.
<b>PERIOD</b>	Specifies the time period used for “cyclizing” a VCD or EVCD pattern, per “Clock Domain”
<b>EDGES</b>	Specifies maximum number of timing edges to expect within a tester period
<b>PINLIST</b>	Assigns type and tester resource to each active Pin
<b>MODEL</b>	(Advantest 83K- and 93K-specific) Specifies tester model
<b>MEMORY</b>	(Advantest 83K- and 93K-specific) Type of pattern memory to use
<b>METHOD</b>	(Advantest 83K- and 93K-specific) Type of test method to use
<b>SUBROUTINE</b>	Defines whether pattern subroutines from source will be flattened in-line with the calling pattern, or kept as a separate, called pattern
<b>MACROSTYLE</b>	For STIL sources
<b>PERSISTENCE</b>	Specifies whether Velocity samples a VCD file for a short or a long portion for calculating a best-fit cyclization period

<b>UNDERSAMPLE</b>	Specifies a strobe interval, N, to apply to a converted pattern, in which only every Nth cycle can have output strobes, and intervening cycles will be masked
<b>DELAY</b>	Assigns cycle delay to pins listed
<b>PATTERN</b>	Defines a custom pattern modified from an existing pattern
<b>TIMING</b>	Defines custom timing for a set of Pins to override the timing derived from the input files
<b>LEVELS</b>	Defines DC levels for a set of Pins to be used in the test program
<b>POWER</b>	Defines a power up or power down sequence
<b>TEST</b>	Creates a specific instance of a standardized test type
<b>FLOW</b>	Defines a sequence of previously-defined TEST instances to be inserted into the test program
<b>TERM</b>	Defines the beginning of termination block that can be used to Set the drive action for comparisons on IO pins

## 2.5 Order of Control Definitions

Many of the Control Definitions can reference elements that are defined in other Control Definitions elsewhere in the Configuration File. For example, a TIMING block definition can reference a Pin defined in the PINLIST block or a Group defined in a GROUP definition.

Elements must be defined in a Configuration File before they can be referenced. Therefore, the order of Control Definitions within the file is important. The order of the Control Types shown in the previous table is the recommended order in which those types should be defined.



**NOTE:** It is not necessary to define every Control Type in a Configuration File. Velocity uses a default set of properties and behaviors for those aspects of a conversion not defined in the Configuration File. Only those Control Types with properties which differ from the defaults need to be defined.

## 2.3 Example Configuration File

The following simple Configuration File example includes definitions from each of the Control Types:

```
#####
#
# ASCII Velocity Configuration File
#####
#
PATH /home/field/testPrograms
DEVICE 56374
PROGRAM ShellExample

#####
# PIN LIST: This will define regular IO and power pin defs
#####
PINLIST
    p0          default      I      0 11
    p1          default      I      0 83
    q0          default      O      0 69
    q1          default      O      0 76
    cpd         default      CLK     0 62
    cpu         default      CLK     0 55
    VS1        DPS          POW     4 0
END PINLIST

GROUP  clocks = "cpu, cpd"

#####
# Pattern lists
# The following patterns will be translated. If the pattern is
# not in the list, then it will be skipped. If the pattern is
# not in the source file then a warning will be issued.
#####
PATTERN multipleLoop
    BASE SpecFunc
    LOOP 5,10 16
END PATTERN

#####
# Timing and Levels
# definitions will define the values of specs. The following
# values will be assigned by default. Groups and pins can
# be defined to override defaults by using a pin name or group
# name.
#####
TIMING default
    period 100ns
    offset 0ns
```

```
duty 50%
drive 25%
receive 90%
END TIMING
LEVELS default
POWER 3.3V
VIL 10%
VIH 90%
VOL 40%
VOH 60%
END LEVELS

#####
# Power up and power down
#####
POWER nominal
VS1 1.25V 500mA 5uS
END POWER

#####
# Test Definitions
# The following tests will be defined as discrete functions
# that can be executed as user commands or as part of flows
#####
TEST contNegative 150
TYPE cont
FORCE -10uA
CLAMP 2V
LOW 400mV
HIGH 800mV
PINS ALL
END TEST
TEST funcSpec 1
TYPE func
PATTERN SpecFunc
END TEST

#####
# Flow Definition
# The following tests will be executed in the following
# order. If no flow is defined, then all the tests will
# be included in the order they are defined.
#####
FLOW experimentName
TEST contNegative
POWER nominal
TEST funcSpec
DELAY 15ms
POWER off
END FLOW
```

## **3.0 CONTROL DEFINITION REFERENCE**

*Definitions and examples for all configuration file variables and blocks*

## 3.1 Environment Definitions

The Environment section of the Configuration File consists of a set of definitions that define the location and naming of the target test program files. Typically, this is the first section in a Configuration File.

Velocity divides the test program location and file names into three parts:

- base path – Typically, points to the directory used as the parent directory of all test programs.
- Device name – Appended to the base path. Categorizes test programs by device.
- Program name – Specifies a base file name that Velocity will use for many of the generated test files. (Pattern files for the target tester are typically named for the source pattern files.)

The test program directory path and file names are defined by the following Control Types:

- **PATH**
- **DEVICE**
- **PROGRAM**

### 3.1.1 PATH

*Syntax:*

**PATH** *pathName*

where,

*pathName* is a directory path specifier

*Example:*

```
PATH /home/programs
```



**NOTE:** The directory path specifier must use valid syntax for the underlying file system.

### 3.1.2 DEVICE

*Syntax:*

**DEVICE** *directoryName*

where,  
*directoryName* is the name of a directory

*Example:*

```
DEVICE      coolChip
```

### 3.1.3 PROGRAM

*Syntax:*

**PROGRAM** *filename* [*equationNumber*]

where,  
*fileName* is the base name to be used for generated test files

*equationNumber* is the base number to be used for equation set numbering

*Example:*

```
PROGRAM      finalTest
PROGRAM      finalTest  10    # Begin with equation set number 10
```



Using the PATH, DEVICE, and PROGRAM Definitions in the above examples, Velocity would create test program files for the Build under the directory

**/home/programs/coolChip**

A number of the created test files would begin with base file name **finalTest**. Their location under the device directory (or subdirectories thereof) would depend on the specific target test system.

## 3.2 General Build Definitions

The General Build section of the Configuration File consists of a set of definitions that define the basic settings common to any conversion.

### 3.2.1 SOURCE\_PORT Definition

Specifies the Source Port for the Velocity conversion. Valid entries can be chosen from anything that is present in the Source Port drop down list in the GUI.

*Syntax:*

**SOURCE\_PORT** *sourcePortType*

where,

*sourcePortType* is a valid licensed source entry.

*Examples:*

```
SOURCE_PORT WGL
```

```
SOURCE_PORT VCD
```



**NOTE:** The source type must match a valid licensed entry or this variable will be ignored.

### 3.2.2 TARGET\_PORT Definition

*Syntax:*

**TARGET\_PORT** *targetPortType*

where,

*targetPortType* is a valid licensed target entry. Valid entries can be chosen from anything that is present in the Target Port drop down list in the GUI.

*Examples:*

```
TARGET_PORT STIL
```

```
TARGET_PORT 93K
```



NOTE: The target type must match a valid licensed entry or this variable will be ignored.

### 3.2.3 LIBRARY

This variable is used to define the name and location of any predefined libraries that you want to have included in the target program

*Syntax:*

```
LIBRARY   EXTERNAL|LOCAL LIBRARY_FILE_NAME
```

where,

**EXTERNAL|LOCAL** directs Velocity whether to physically copy this library to the target program or simply refer to its path through a makefile or some other method depending on the target.

*LIBRARY\_FILE\_NAME* is the path and name of the library file itself

*Examples:*

```
HEADER      LOCAL /usr/local/lib/someLibrary.so
HEADER      EXTERNAL  /usr/local/lib/ someLibrary.so
```



NOTE: referenced library file must be present and valid for the target

### 3.2.4 HEADER

This variable is used to define the name and location of any predefined headers that you want to have included in the target program

*Syntax:*

```
HEADER    EXTERNAL|LOCAL HEADER_FILE_NAME
```

where,

**EXTERNAL|LOCAL** directs Velocity whether to physically copy this header to the target program or simply refer to its path thro a makefile or some other method depending on the target.

***HEADER\_FILE\_NAME*** is the path and name of the header file itself

*Examples:*

```
HEADER    LOCAL /usr/local/include/someHeader.h
HEADER    EXTERNAL    /usr/local/include/someHeader.h
```



NOTE: referenced header file must be present and valid for the target

## 3.3 Pin Configuration Definitions

### 3.3.1 PINLIST Definition

*Syntax:*

**PINLIST**

*pinName*      *domain pinType*      [*slot*] [*channel*] [*alias1* [*alias2*...*aliasN*]]

**END PINLIST**

The PINLIST block defines, per pin, the tester channel assigned and any alternate versions of that name used in the simulation or ATE conversion source.

The tester channel information that can be specified includes:

- domain:
  - **For digital pins:** default or any port name that does not have a n underscore)
  - **For power supplies:** DPS16, DPS32, UHC4, MSDPS (type must be POW)
- *pinType*: **I, O, IO, CLK, TRIG, REF, POW, R, DIR, A, MASK** or **NC**.

*I*: pure input pin  
*O*: pure output  
*IO*: bidirectional  
*CLK*: special case of input which will use spec values that apply to CLK instead of regular simple drive actions when specs are available.  
*REF*: pure input pin that is used as a zero reference for all edge values in a VCD/EVCD.  
*POW*: power pin. This would then assume that the domain value is assigned with a power supply type  
**R**: relay pin (essentially unused. But, usefeul as a placeholder)  
**A**: analog pin (essentially unused. But, useful as a placegolder)  
**MASK**: pin will be included in the output files, but all source data will be ignored. Data will be assigned as “X” on all cycles  
**NC**: No Connect. Pin will be completely ignored as removed from outputs  
**TRIG**: pure input that will automatically default all data to “0” so that triggers can be added manually on the tester.

- Slot number (optional)
- Channel number (optional)

- **Aliases** This is a space delimited list of alternate names that can be used in source files that will be expressed in the target files as whatever is in the pinName field. You can specify as many Aliases on a pin line – separated by whitespace – as you need. Velocity uses Aliases to match simulation or ATE pin names that are different from the target pin name.



#### “REMOVE” as alias

Generally, aliases and pin names must be unique. The exception to this is when the alias “REMOVE” is used. This is a special alias that instructs Velocity to remove this pin from the resulting compiled target files, but leaves it in so that it can be used as part of MASK blocks or simply as markers in the ascii files. The pin will not be visible once loaded on the tester.

The following is an example of a PINLIST definition:

```
#####
##  PinList Definition
#####
PINLIST
  ANALOG_VDD          DPS16    POW    230    2
  HOLDn               default  I     101    1  hold_n holdn
  WPn                 default  O     101    8  wp_n
  anapadext_data_n   DPS16    POW    230    3
END PINLIST
```



Pin ANALOG\_VDD uses channel 2 of a DPS16 card in slot 230.

Also, note that pin HOLDn has aliases of hold\_n and holdn, meaning that it can take its data from simulation or ATE conversion sources that use either of those alias names.

### 3.3.2 BIDIRECTIONAL CONTROL

One special case within the PINLIST section is the bidirectional control pin. A control pin will only be needed for standard VCD translations. This source port format does not have a state character differentiation between input and output. Therefore, without the extra control wire, there is no way to determine the IO state of a bidirectional pin. For these types of simulations, there must always be a set of control wires that would also be included in the VCD file. These are “virtual” pins that are used to define the IO direction of other pins. In other words, these pins are controls for other pins.

The Velocity configuration syntax for these is similar to the regular pins except the alias column would be used to make reference to another pin instead of merely providing an alternate name for the active pin. There will then be two rows used to define each bidirectional pin. One for the pin itself, and another for the control wire defining its IO state. Once this is defined the control wire’s state is kept as the VCD file is processed. At any given time, if the control pin is actively high, then the pin which it controls is set to output mode. If the control wire is low, then the pin which it controls is set to input mode.

Example

DATA0	default IO	data[0]	
DATA1	default IO	data[1]	
DATA2	default IO	data[2]	
DATA3	default IO	data[3]	
control0	default DIR	DATA0	
control1	default DIR	DATA1	
control2	default DIR	DATA2	
control3	default DIR	DATA3	

In the above example, there are 4 pins defined as IO and 4 pins defined as DIR. For the IO pins, there is an alias that defines an alternate nomenclature that the simulation file might use to express the given pin name. For the DIR pins, the alias column contains an entry that is already defined as a column 1 pin name. This pseudo-alias is the key that provides the connection between the control pin and its target.

The pin listed as DIR type will not show up in the target test pattern. These are treated as virtual pins rather than real pins that would require data to be provided behind them.

As stated above, the default behavior for control pins is that a control pin high means output mode. Control pin low means input mode. This behavior can be inverted by also inserting the keyword “NEG” at the end of the control pin definition (after the alias). If the NEG keyword is used, then the convention will be opposite. Control pin high will indicate input mode. Control pin low will indicate output mode.

Example

DATA0	default IO	data[0]	
control0	default DIR	DATA0	NEG

### 3.3.3 GROUP Definition

The GROUP Control definition allows you to assign a name to a group of pins, for easier reference elsewhere in the Configuration file.

To define a Group, use the keyword **GROUP** followed by a Group name, followed by an equals sign (=) and a comma-separated list of pin names enclosed in double-quotes (""). The following is an example of a Group definition:

```
GROUP DBUS = "D0, D1, D2, D3, D4, D5, D6, D7"
```



If you use a group as a member of another group, this group must have already been defined. If not already defined a configuration loader error will occur.

### Automatic Group definitions

There are a number of groups that are automatically generated. These groups are generated automatically because certain APIs assume that they are there. For example, the functional test API does an automatic connect on all pins. This assumes that there is a group named “allpins” that is there and this group’s contents include all of the digital pins for a given device.

<b>Automatic Group Name</b>	<b>Description of Contents</b>
<b>allpins</b>	All digital pins not including any trigger pins that may be assigned
<b>allios</b>	All bidirectional pins
<b>allins</b>	All pins that can have input actions. Includes bidirectionals as well as input only pins
<b>allouts</b>	All pins that can have output actions. Includes bidirectionals as well as output only pins
<b>triggerPins</b>	Group to collectively define all trigger pins
<b>Allpins</b>	All digital pins including the trigger pins. This is used to connect and disconnect all pins.
<b>allSupplies</b>	All DPS defined power pins

## 3.4 Source-Port-Specific Variables

The Source-Port-Specific section of the Configuration File consists of a set of definitions that define build settings specific to the selected Source Port.

### 3.4.1 IGNORE TIMEPLATE

Sometimes WGL files have timeplates that the user wants to ignore like at the start of a simulation. You can instruct Velocity to ignore these timeplates.

**IGNORE TIMEPLATE** timeplatename

### 3.4.2 JOB

This directive is used to enable a specific job as defined in a J750 or UltraFlex source test program. This feature is ignored for all other input ports. When used, the active spec sheets for timing and levels are picked from a specific job. If the job does not exist in the source, by a typo or any other reason, the last job is always the one that is chosen. This is also what is chosen when no JOB is defined at all.

*Syntax:*

**JOB** ;jobName'

*Example:*

```
JOB      QA_TEST
```

### 3.4.3 MACROSTYLE

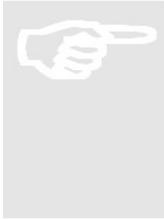
This variable will allow you to tell velocity how to interpret Macros and Procedure when loading STIL simulation files. Depending on the way these are created variables to pass values into subroutines will either be passed through STIL Macros or with STIL procedures. It will be one or the other but not both. By default these are done with Macros. Therefore, therefor the default value for this flag is “1”. But, if your source STIL files pass variables into procedures instead, you can handle this by disabling the passed variable usage in the macros by setting the MACROSTYLE flag to “0”

*Syntax:*

**MACROSTYLE** 0/1

*Example:*

MACROSTYLE	0	# Procedures pass variables
MACROSTYLE	1	# Macros pass variables (This is default behavior)



IEEE STIL is a very richly defined language. There are quite a few variations and these are not always compatible with one another. If you translate a STIL pattern and see scrambled data or experience a crash, it is very often because the MACROSTYLE variable is backward. In most cases, there will be header information that tells Velocity where the file was generated which will then allow Velocity to self determine the proper MACROSTYLE. But, sometimes hand generated STIL will not have the necessary header information. That is why this variable is present. It allows you to tune the STIL translations accordingly.

### 3.4.4 MASTER

When using any of the EVCD format, there may be places where the state characters that are used indicate that the bench and the DUT are both driving. If not specified, Velocity will take the DUT as the master. That means that the competing drive values will result in the DUT value being used instead of the bench. If you specify the bench as the master, the opposite will occur. EVCD state characters of “0” or “1” will then be assumed to be tester drives instead of tester strobe values.

*Syntax:*

**MASTER** *DUT/BENCH*

*Example:*

```
MASTER      DUT
MASTER      BENCH
```

### 3.4.5 PERSISTENCE

By default, a very small portion of a VCD file is used to evaluate the pins and calculate periods. Sometimes this is insufficient when multiple time domains are active. One domain may start toggling later than another. For this reason, there may not be enough actions in one domain to properly calculate a period. If this is the case, the user will receive a message that informs them that PERIOD values for one domain will track with the other. If this is undesired there is a secondary calculation scheme that might work. This is called “PERSISTENCE” mode. If enabled, a much larger page size will be used to calculate periods.

This is disabled by default, but can be explicitly assigned with the following syntax.

*Syntax:*

**PERSISTENCE** *ON/OFF*

*Example:*

```
PERSISTENCE ON      # persistence is enabled
PERSISTENCE OFF     # persistence is disabled
```

### 3.4.6 SYNC

SYNC will allow Velocity to tune itself to a given timestamp before it will start handling the calculations for defining periods for VCD/EVCD translations. In certain instances, simulations will have multiple data rates present in different domains. If the simulation is very large it may require a very large amount of one domain to be processed before the second domain begins to toggle.

The SYNC variable will essentially fast forward the self discovery algorithm to focus on a particular area of the simulation. This can speed the translation greatly for large simulations.

The SYNC point defaults to time zero unless overridden by this variable which is defined in units of time. Nanoseconds (ns) are used if no unit is specified. Units can be used as well

*Syntax:*

**SYNC** *timeValue[unit]*

*Example:*

SYNC	1000	# SYNC period discovery to 1000ns
SYNC	5.52ms	# SYNC period discovery to 5.32ms

### 3.4.7 UNDERSAMPLE

This allows you to apply a global value that will block strobos except in cycles with a clean modulus to whatever is specified here. When OFF, all cycles are strobed.

*Syntax*

**UNDERSAMPLE** ON|OFF

*Examples:*

<b>UNDERSAMPLE</b>	OFF	# No under-sampling
<b>UNDERSAMPLE</b>	5	# Under sample by a factor of 5

### 3.4.8 OVERSAMPLE

This allows you to apply a domain specific value that will allow oversampling to be used. This is a good way to handle slow asynchronous behavior or to automatically insert oversampled strobing if edge placements are not deterministic in the source format. This applies only to VCD/EVCD. Other formats will ignore this feature



The oversample value is applied only if self discovered timing is used for VCD/EVCD translation. The calculated period would then be divided by the oversample value to give a faster (oversampled) cycle period

Syntax

**OVERSAMPLE**      **domainName** value

Examples:

```
OVERSAMPLE default 10 # undersampling the default domain by a factor of 10
OVERSAMPLE I2C     10 # Under I2C domain by a factor or 10
```

### 3.4.9 VCDPAGE

This allows you indirectly control the size of pages that are used during a VCD translation. The value is in percentage. This default to 100% but you can increase or decrease the page size depending on this value. This is used to limit the size of loops when made smaller or remove page boundary issues by making the page larger.



The default page value is 100%. The page is then calculated based upon the number of pins and the density of activity. This is not something that is specifically controlled to a fixed value. This is a general property that can be used to give you larger or smaller page sizes relative to the base.

Syntax

```
VCDPAGE value
```

Examples:

```
VCDPAGE 10 # Use smaller page  
VCDPAGE 500 # Use larger page
```

## **3.5 Target-Port-Specific Variables**

The Target-Port-Specific section of the Configuration File consists of a set of definitions that define build settings specific to the selected Target Port.

### **3.5.1 BINARY**

Compilation to the 93K will result in a pattern master file as well as a merged binary file. If you wish to include statistics in the compilation then you'll need to create the concatenated BINL file as well. This is enabled by turning BINARY "ON"

*Syntax:*

**BINARY** *ON/OFF*

*Example:*

```
BINARY    ON      # merged BINL will be created
BINARY    OFF     # Only PMFL will be created
```

### 3.5.2 COMBINATION

Velocity dynamically builds a combination file base on your XMODE state. If the user want to tell Velocity to use a specific combination file, here is the syntax. This is specific to the 93K testers.

Additionally, there are arguments to explicitly provide the MINIMUM number of combinations. By specifying MINIMUM, the resulting timing will contain ONLY those waveforms that are explicitly used. There will be none that are added for debug or online editing purposes.

If MAXIMUM is specified, there will be additional waveforms added that are logically similar to the ones that are used already. That is if an edge drives 0, then it will also have a waveform to drive 1, even if these never occurs in the source. It is logically reasonable to add both. Similarly, if a strobe is present on a given edge, both strobes high and low as well as the X will be used as valid combinations. MAXIMUM is the default value for this variable

If very large simulations are used, then evaluation of the results can be limited to a predefined number of cycles. For example if you have a long scan test, you don't necessarily need to evaluate the entire file before concluding on which combinations are needed. By using a number for this value, a cycle limit will be applied for evaluating the required combinations

*Syntax:*

**COMBINATION** *filename/MINIMUM/MAXIMUM/number*

*Example:*

```
COMBINATION      /home/demo/device/ACT74.cmb
COMBINATION      ACT74.cmb # This will look alongside the CFG file
COMBINATION      MAXIMUM   # combinations will be calculated and logically
                      #     similar states will be inserted automatically
                      #     for debug purposes
COMBINATION      MINIMUM   # combinations will be calculated and logically
                      #     Only those states that are used by each
                      #     pattern and pin will be used. Smallest wave
                      #     table possible will be created for debug
                      #     purposes
```

### 3.5.3 CONTEXT

There are certain instances where different configurations of a chip require the 93K pin types to be defined differently for different modes of operation. This can be accomplished by defining the “CONTEXT” of the PINLIST block directly. If not specified, the context will be defined as default. The Velocity CFG can assign the value to any string. Once this is done, any patterns, timing, and even levels that are created will be associated with that context only.

You can then add the extra context to the 93K pin file that you use for loading

*Syntax:*

```
CONTEXT contextName
```

*Example:*

```
CONTEXT          inputMode
CONTEXT          outputMode  3.5.3 CONFIGURATION (PIN FILE)
```

### 3.5.4 CONFIGURATION (PIN FILE)

This variable allows for a pre-defined 93K pin file to be referenced in the Velocity generated testflow instead of the auto-generated one that is used for compilation. If nothing is specified for this variable, then the pin file used by the testflow will be auto created from the PINLIST block of the CFG. Using this variable to reference pre-defined pins file is useful if you are applying patterns to an existing program where you have created you want to use multi-site, have analog pins, have relay setups defined, or certain power supply types.

There are a number of 93K pin file features that do not work when used with the pattern compiler. Multi-Site and Analog setups are two such features. The CONFIGURATION variable allows you to export to an existing test program directory without touching the pin file that is already in use.

It must be known that the pin list of the pre-defined CFG is compatible with the PINLIST generated for compilation. If this is strictly maintained then any binary pattern and timing will automatically work the pre-defined file.

```
CONFIGURATION          myCfgName  # testflow will use pre-defined pin file
```

### 3.5.5 CTIM

This variable will control whether or not timeset switching is allowed or not for the V93K. By default, CTIM's are disabled. These are disabled because this will result in either very long run times due to break cycles or will result in timing that is too large to compile. But, there are situations where this is the correct usage. So, this variable is present to enable such actions on the system

Syntax:

```
CTIM ON|OFF
```

*Examples*

```
CTIM ON
CTIM OFF
```

### 3.5.6 DOMAIN TRACKING

Domain tracker allows you to specify the timing relationship between ports. In the following example the device has three well defined ports: 1. CLOCK, 2. DDR and 3. SDR

*Syntax:*

```
DOMAIN
  domainName      MASTER|SLAVE [slaveDomainName [periodMultiple]]
```

where,

**domainName:** matches domains defined in PINLIST block

**MASTER|SLAVE:** determines whether the period is defined itself or dependent on another domain

**slaveDomainName:** If specified as a SLAVE, then this will refer to another port already defined as master

**periodMultiple:** defaults to 1.0, If other, then the current domain will track with the MASTER at the given period multiplier

*Example:*

```
DOMAIN
  CLOCK MASTER # PortName MASTER(Reference)
  DDR   SLAVE CLOCK 1.0 # PortName SLAVE Port2Track Ratio
  SDR   SLAVE CLOCK 75 # PortName SLAVE Port2Track Ratio
END DOMAIN
```

### 3.5.7 FASTMODE

This directive is used to enable or disable the use of Fastmode. Fastmode is an Advantest digital option that uses a software programmable switch to enable faster drivers. When this mode is used, special care will be taken in how the timing is exported. All pins that are set to toggle at a rate that is faster than 1.25ns will be set to use the FAST option. Data bit rates and all compilation and tester file options will automatically be adjusted. If nothing is specified, then this option is assumed to be OFF.

*Syntax:*

```
FAST ON/OFF
```

*Example:*

```
FAST      ON
```

### 3.5.8 FSPINS

This block specifically applies to the SmartScale series of cards for the Advantest V93000 tester. These channels have a property that allows the Z edges to be handled independently from the drive edges. This features allows for a zero turn around time setup for bi-directional pins. If any target other than V9300 coupled with a MODEL definition of PS6800 is used, then this block will be ignored.

*Syntax:*

```
FSPINS
  pinName1 | groupName1
  pinName2 | groupName2
  ...
  pinNameN | groupNameN
END FSPINS
```

This blocks serves as a container for a list of pins or groups that should be setup with timing in such a way that the drive and Z edges are separated. Any pin left off this list will be treated as a regular IO pin instead of a FAST IO pin.

*Example:*

```
FSPINS
  DQ
  MQ
COMBINATION
```

### 3.5.9 MEMORY

Compilation to the 93K can be done using Vector Memory or Sequencer Memory. By default, vector memory is used. If you want to only use sequencer memory you can override using this directive .

*Syntax:*

**MEMORY** *SM/VM*

*Example:*

MEMORY	SM	# sequencer memory
MEMORY	VM	# vector memory

### 3.5.10 METHOD

Specific to the Advantest 83K and 93K ports.) Specifies the type of test method to be used: Classic (CTM) or Universal (UTM). Used for generating the appropriate format for the test flow file.

*Syntax:*

**METHOD** *CTM/UTM*

*Example:*

METHOD	CTM	# Classic Testmethod
METHOD	UTM	# Universal Testmethod

### 3.5.11 MODEL

*Syntax:*

**MODEL** *modelType*

where,

*modelType* is a tester model type. This Control is specific to the Advantest 83K and 93K tester ports. Valid entries are:

- F330
- C400
- P1000
- PS400
- PS800
- PS6800
- PS3600

*Examples*

```
MODEL      PS3600      # PinScale
MODEL      P1000      # Single Density 93K
MODEL      PS6800     # SmartScale
```

### 3.5.12 STATEMAP

(Specific to the Advantest 93K ports.) Specifies that the “STATEMAP” blocks should be activated in the resulting compiled timing files. This block is required if reverse compilation from binary back to ascii is desired. It is also required when using some forms of the SCAN\_TML

*Syntax:*

**STATEMAP** *ON/OFF*

*Examples*

```
STATEMAP   ON
STATEMAP   OFF
```

## 3.6 Verilog Feedback Variables

There are set of configuration commands blocks that are intended specifically to affect the export of Verilog feedback files. These variables will have no effect on regular ATE output files. If “Enable Verilog Feedback” is not checked in the GUI or enabled from the command line, then all of these variables will simply be ignored by the Build process.

### 3.6.1 MAXDELTA Definition

When Verilog feedback is enabled, there may be instances where delta values for timestamps may be too large for the compiler that will be used. By default, the maximum delta is 400us. If a delta is larger than this it will be broken into multiple timestamps and spread.

This directive can be used to use a different value other than 400uS.

If no units are specified the number will be assumed to be in ns.

*Syntax:*

**MAXDELTA** *value*[ps/ns/us/ms/s]

*Example:*

```
MAXDELTA    200us    # max delta of 200us
MAXDELTA    500000  # max delta of 500000ns
```

### 3.6.2 MODULE

This variable will define the moduleName that is to be used in the Verilog feedback files if that option is chosen. If not defined at all, the default for the moduleName variable in the testbench and EVCD files created will be “moduleName”. This allows you to tune it so that resimulations can happen more seamlessly.

*Syntax:*

**MODULE** *moduleName*

*Example:*

```
MODULE      hx_5672
MODULE      dsp_1080
```

### 3.6.3 VERILOG Definition

When using the Verilog feedback path, testbenches can have their timings expressed in one of two ways. Sequential timing will express all timestamps as relative deltas from the previous stamp. On large simulations these numbers can end up too large for the target Verilog compiler. Parallel will express each timestamp as its wall time clock value. This can result in very large numbers that may crash. By default, parallel timing is used.

*Syntax:*

**VERILOG** *SEQUENTIAL/PARALLEL*

*Example:*

VERILOG	SEQUENTIAL	# Express edges as delta from previous edge
VERILOG	PARALLEL	# express every edge as a unique timestamp

### 3.6.4 WINDOW

Verilog files are always print on change. In simulations, that means that only transitions are exported as timestamps. For input actions this is simple. For output actions, the ATE versions of the same stimulus will inherently insert Z-actions before each strobe. By default, these Z actions are left out of the Verilog files. If the user wants to add these actions in the “WINDOW” variable can be used to essentially define the length of the active strobe window. If WINDOW is OFF, as it is by default, then there will be no closing actions added o close the strobe windows.

In the end, this is a more exact representation of what the ATE is doing, however, the resulting verilog files will be much larger.

*Syntax:*

**WINDOW** OFF| *value[ps/ns/us/ms/s]*

*Example:*

WINDOW	OFF	# no windowing off output actions
WINDOW	20ps	# Use 20ps strobe window

### 3.6.5 FEEDBACK FILTER

Verilog feedback files will export all ATE actions in all cycles and on all pins by default. In some cases, the user may want to limit the export. This can be limited to a certain cycle range. It can be limited by certain pins. Or, you can choose to export certain pins as running clocks instead of explicit data. All of these actions are taken to limit the size and scope of testbench. FEEDBACK filter blocks can be created that can tune the Verilog scope on a pattern by pattern basis

*Syntax:*

```

FEEDBACK default|patternName
    BASE                baseName
    START              BEGIN|timeStart|cycleStart
    STOP               END|timeStop|cyceStop
    pinName1|ALL      ON|OFF|CLK
    pinName2          ON|OFF|CLK
    pinName3          ON|OFF|CLK
    ...
    pinNameN         ON|OFF|CLK
END FEEDBACK

```

**default | patternName:** name of feedback block. If the name is “default”, then this filter will apply to all patterns. If the name is something else, then the filter will only apply to patterns whose name matches that of the FEEDBACK block.

If the desired *patternName* is different than the source file, you can specify the connection to a given simulation by additionally assigning the **BASE** variable. If this is done, the translation will use the base Name as the simulation hat is loaded, but will export to the name specified by *patternName*,

**START & -STOP:** This is an optional field that can assign the start and stop location for the translations. If left off then ALL cycles will be exported. These times can be expressed in either picoseconds or as a percentage. If the “%” is used, then you can export just the segments you like as a percentage of time. It is important to realize that this percentage is relative to the time value as opposed to the physical file location.

**pinNameN|ALL:** pin name to be added as regular pin (ON), removed from Verilog files (OFF), or included in Verilog as a running clock (CLK)

**Examples:**

```
# Export Verilog on all cycles for all pins except data[0-2].
FEEDBACK default
  ALL ON
  data0 OFF
  data1 OFF
  data2 OFF
END FEEDBACK

# Export Verilog for patternA for the first 25% only. Include only CLK32
# expressed as running clock and data[0-2] expressed as regular data
FEEDBACK patternA
  START    BEGIN
  STOP     100us
  CLK32    CLK
  data0 ON
  data1 ON
  data2 ON
END FEEDBACK
```

## 3.7 General Purpose Variables

The following list of variables are ones that can be applied to any combination of Source and Target port. In some instances these variables are meant simply to provide default states for objects in the GUI. In other cases, these variables will have no analogous command line or GUI feature and will affect outputs all on their own.

### 3.7.1 COMMENT

When using any of the serial protocol formats, this variable will optionally add comments to the patterns That will inform you of the state of the protocol as data is transmitted. These comments will also Be viewable on the target system in the pattern viewers

*Syntax:*

**COMMENT** *ON/OFF/ALL/OPTIMIZED*

where,

ON: comments are passed in from source. Scan instances are marked

OFF: comments are blocked

ALL: comments are used like “ON” above, but additionally, timestamp markers are included

OPTIMIZED: comments are ignored when compression is used. If a comment is within a field of cycles that can be compressed, then the comment will be swallowed and dropped.

*Example:*

COMMENT	ON
COMMENT	OFF
COMMENT	ALL
COMMENT	OPTIMIZED

### 3.7.2 DDRMODE

In many high speed situations, such as DDR, there will be simulations that have bidirectional pins that go from input to output in a single cycle. The target platform may not be capable of doing this. If a pattern requests this, the end result will be missing input or output data at the DUT. The drivers may not be able to turn on or off in time. If this is the case, you can automatically account for this turn-around issue by applying the affected pins to the following syntax.

*Syntax:*

```
DDRMODE pinName1 [pinName2 ...pinNameN]
```

When `DDRMODE` is enabled for a given pin or group of pins, the resulting vector data will be automatically modified to account for driver turn around so that important data is not lost due to hardware constraints.

*Example:*

```
DDRMODE DQ DQS MDQ MDQS # turn around issues will be adjusted
```

### 3.7.3 DELAY

In certain instances it may be necessary to move data on a set of pins forward or backward by a number of cycles to get the pattern to match how it will work on silicon. This is done because either the simulation does not match silicon or even because the performance on one source is different than the performance on another target. The DELAY block can be used to adjust the data. The syntax for the block is as follows

**DELAY**

```
pinName value  
pinName value  
,  
,
```

**END DELAY**

*DELAY* is a keyword that indicates the beginning of a DELAY block

Only one *DELAY* block should exist within a given test configuration. No errors will be seen but only the last delay block listed will be inserted into the test program

*pinName* must explicitly match a *PIN* or *ALIAS* defined prior to the *PINLIST* block

*value* will define the cycle count for which data will be delayed for a given pin. Negative numbers will move data forward. Positive values will move the data backward

The example below delays s data pins by 3 cycles, while forcing theCLK to happen 1 cycle early.

```
DELAY  
  DATA1  3  
  DATA2  3  
  CLK    -1  
END DELAY
```

### 3.7.4 INIT

In certain instances it may be necessary to apply arbitrary sequences of data to pin that are not defined in the source. These sequences can be made up of any valid vector state characters

#### INIT

```
pinName defaultSequence  
pinName defaultSequence  
,  
,
```

#### END INIT

*INIT* is a keyword that indicates the beginning of a INIT block

*pinName* must explicitly match a *PIN* or *ALIAS* defined prior to the *PINLIST* block

*defaultSequence* will define the state to use instead of whatever state is defined by the source pattern. This will also assign this default state to use when no action is defined by a source pattern. This is useful for applying states to unreferenced pins that need to be biased in a particular way.

```
INIT  
  DATA1  0000011111  
  DATA2  1010101010  
  CLK     110011001100  
END INIT
```

### 3.7.5 PAGE

This directive is used to redefine the number of scan instances that will be included in a single vector file. By default, scan patterns will be broken into separate files that can be bursted together. The reason that these are broken up is because it can make processing and debug easier in that you can mask certain chunks of patterns to make loading quicker.

But, since some patterns don't like the way that bursts are issued, this flag will allow you to make the page size larger or smaller to increase or possibly remove entirely, the need for paging.

PAGESIZE will also be used to define the seed size to use for the paging calculation for VCD/EVCD translation. By default a seed of 1000 cycles is used for this calculation. In some cases, you may want to change the size of pages to address paging issues caused by staggered busses or asynchronous behavior in simulations.

*Syntax:*

**PAGESIZE** *numberPerPage*

*Example:*

```
PAGESIZE      10000  # This will break file every 10,000 scan instances
                # Or a seed value of 10000 for VCD/EVCD files

PAGESIZE      50     # This will break file every 50 scan instances
                # Or a seed value of 50 for VCD/EVCD files
```

### 3.7.6 INIT

In certain instances it may be necessary to override the termination used for the drive resources that are associated with the read on an IO pin. To drive high or drive low while reading, the termination block is employed. Pins can be set here. If not listed in the termination bloc, a pin will retain its regular Z termination.

#### INIT

```
pinName|groupName defaultState
pinName|groupName defaultState
,
```

#### END INIT

*INIT* is a keyword that indicates the beginning of a STATIC block

*pinName* must explicitly match a *PIN* or *ALIAS* defined prior to the *PINLIST* block

*groupName* must explicitly match a *GROUP* that as been defined prior to the *PINLIST* block. The exception to this is the special cases of “ALL\_IN”, “ALL\_O”, and “ALL\_IO” which will apply the static state to all pins tht match one of these pin types

*defaultState* will define the state to use instead of whatever state is defined by the source pattern. This will also assign this default state to use when no action is defined by a source pattern. This is useful for applying states to unreferenced pins that need to be biased in a particular way.

```
INIT
  DATA1  0
  DATA2  0
  CLK     Z
  dataBus X    # default all pins in group "dataBus" to X
  ALL_IN  0    # default all inputs to 0
END INIT
```



In older releases, this block was called the STATIC block. This was changed to INIT to reflect the fact that these are simply ways to redefine the default initial state, rather than to override and defined STATIC data. The new syntax is more straight forward.

### 3.7.7 SUBROUTINE

This variable will allow you to determine how subroutines are handled. By default, subroutines will be treated as separate pattern files. But if you turn these off, the calls themselves will be flattened and added directly to the calling pattern.

The default state for this is “ON”

If you want to additionally, expand subroutine calls that may have variable data to make them unique, you can additionally use the keyword “ALL”

*Syntax:*

**SUBROUTINE** *ON/OFF/ALL*

*Example:*

```
SUBROUTINE      OFF
SUBROUTINE      ON
SUBROUTINE      ALL # expand subroutines that might have different data
```

### 3.7.8 TERMINATION

In certain instances it may be necessary to override the termination used for the drive resources that are associated with the read on an IO pin. To drive high or drive low while reading, the termination block is employed. Pins can be set here. If not listed in the termination bloc, a pin will retain its regular Z termination.

#### TERMINATION

```
pinName LOW|HIGH
pinName LOW|HIGH
,
```

#### END TERM

*TERM* is a keyword that indicates the beginning of a DELAY block

*pinName* must explicitly match a *PIN* or *ALIAS* defined prior to the *PINLIST* block

*LOW|HIGH* will define the state to use instead of Z for the drive action. If the pins is not specifically terminated LOW or HIGH, then it will be terminated to a Z state.

The example below terminates 2 pins low and one pin highy.

```
TERMINATION
DATA1    LOW
DATA2    LOW
CLK      HIGH
END TERM
```

### 3.7.9 TRISTATE

This directive is used to enable and disable the tristate comparison feature. By default, tristate comparisons will be imported as active strobe conditions. If the tristate comparison is turned off, tristate comparisons will be mapped to X's. Turning on the tristate compare is equivalent to leaving the statement out completely.

*Syntax:*

**TRISTATE** *ON/OFF*

*Example:*

TRISTATE	OFF
TRISTATE	ON

### 3.7.10 WARNINGS

This value is used to allow the user to block highly repetitive warnings that can appear in some translations. If these warnings are deemed to be ignorable, which they are most of the time, this will block these warnings so the log file is easier to read. By default, warnings are all ON.

*Syntax:*

**WARNING** *ON/OFF*

*Example:*

WARNING	OFF
WARNING	ON
COMMENT	ALL # Include timestamps in binary comments

### 3.7.11 MINREPEAT

This variable is used to redefine the minimum value that is allowable for repeat blocks. The overall vector count of the repeat is assumed. So, the block length x the loop count is the loop length. MINREPEAT sets the minimum length for a loop. By default, the minimum length of this loop is 64.

*Syntax:*

**MINREPEAT** *value*

*Example:*

MINREPEAT	64	# same value as the default
MINREPEAT	8	# lower minimum repeat than the default
MINREPEAT	128	# larger minimum repeat than the default

### 3.7.12 MAXLOOP

This variable is used to redefine the maximum number of loops that will be allowed in a given translation. In some cases, too many loops can cause sequencer memory issues. This allows you to define a cap for the total number of repeat or loop commands that are allowed. By default 32,768 loops can be defined.

*Syntax:*

**MAXLOOP** *value*

*Example:*

MAXLOOP	32768	# same value as the default
MAXLOOP	4096	# lower minimum repeat than the default
MAXLOOP	65536	# larger minimum repeat than the default

### 3.7.13 DEBUG

This variable is used to define the percentage of the file that is used for debug mode. By default 1% of the source file will be used. This variable can override that so that a larger (or smaller) percentage of the file can be used.

*Syntax:*

**DEBUG**      **value**

where,

**value** is the percentage value

*Examples:*

```
DEBUG 50.7%
```

```
DEBUG 10%
```



NOTE: This value can be any number  $>0$  and less than or equal to 100

### 3.7.13 BURST

Pattern bursts can be automatically generated using BURST/END BURST syntax shown below.

```
BURST burstName
      patternName1
      patternName2
      ,
      'patternNameN
END BURST
```

*burstName* This will define the name of the burst. This will automatically be compiled along with all patterns in the list

*patternName1:N* patterns in the burst will be called in the order that they are defined, in the BURST block. If multi-port is used, the burst that are created will also automatically be multiport bursts. No additional syntax is required to convert from single to multiport burst. The PINLIST block itself will determine if single or multiport

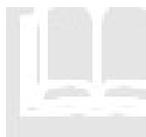
```
BURST  burstPatternA
      setupPattern
      functionalPattern1
      functionalPattern2
      closeAllFunction
END INIT
```



If defining a BURST, it must be known that ALL of the patterns in the burst list must be present in the setup before the burst will be activated. If a pattern is missing you will get a warning and the BURST will not be created.

## **3.8 Timing Variables**

The Cyclization Timing section of the Configuration File is used mainly for controlling the conversion of VCD and EVCD patterns, where the stream of events needs to be divided into tester cycles.



**BACKGROUND:** For more information on VCD/EVCD patterns and cyclization, refer to the previous chapter in this guide called “What Happens During the Conversion Process?”, and, specifically, the section called “‘Cyclized’ vs. ‘Uncyclized’ Pattern Formats”.

### 3.8.1 CAPTURE

Digital capture requires a couple of things to be in place in the pattern and timing. First, there needs to be timing waveforms available for capture. Capture uses different resources than what would be used for normal functional strobes. Secondly, these waveforms need to be used in the pattern with separate wave indexes for functional vs digital capture strobes.

Lastly, there needs to be digital capture variables in place that defines the bit order for MSB to LSB

The Velocity Capture block will ensure that the timing is prepared for the appropriate pins. Additionally, this block will define the MSB and LSB order so that proper variables can be created.

Syntax,

```
CAPTURE  [frameLength]
          pinNameN
          .
          .
          pinName1
          pinName0
END CAPTURE
```

Where,

The pin names for the digital capture are listed from MSB to LSB

```
CAPTURE  2
          Cap7
          Cap6
          Cap5
          Cap4
          Cap3
          Cap2
          Cap1
          Cap0
END CAPTURE
```



**NOTE:** The capture block shown here will is only used to define the bit order for a digital capture variable. The actual digital capture data needs to be defined in the source files or added to the pattern through Velocity MASK setups.



**NOTE:** The frame length is the number of cycles with capture to include per captured data value. In the above example, the frame length of 2 will result in each captured value being 16 bits long. 2 cycles of 8 bits each.

### 3.8.2 DATARATE

DATARATE refers to the number of ASCII vectors that will be expressed per single tester cycle. This variable will provide a default value that can then be overridden by the GUI. Expressing in the CFG will provide a failsafe way of defining this so that it is always set when loading a particular CFG.

This variable then sets the maximum data rate that will be used. By default, the same data rate will apply to all ports if more than one port is defined. However, this is only a target. Depending on the relative frequencies of each port, slower ports may be slowed down so that they do not use this value. This is done so that very slow ports are not burdened with the unneeded complexity of a high data rate.

If the optional `domainName` variable is used, the data rate will be applied only to the single port that is referenced. This will allow for arbitrary combinations of xModes to be used.

Syntax,

```
DATARATE [domainName] value
```

Where,

Value = some number between 1 and 8.

`domainName` = optional application that will assign the data rate to only the one domain.

EXAMPLE

```
DATARATE 3
DATARATE 6
DATARATE clocks 4
```

### 3.8.3 EDGES

EDGES refers to the number input drive edges per data strobe when cyclization of VCD/EVCD is being done. Note that this variable is ignored when pre cyclized formats are translated.

If you want to use R0 or R1 or SBC clock data then you would set the EDGES count to “2”. This would then set up your period so that a rise and a fall edge are present on input data for every single output data strobe.

If you want strobe on both the rising and falling edge of input clocks then you would set this variable to “1”.

Syntax,

```
EDGES      [domainName] value
```

Where,

Value = 1 or 2. This will represent number of input clock edges per strobe.  
 1 = DNRZ clock data  
 2 = R0/R1 clock data

domainName = optional application that will assign the data rate to only the one domain.

#### EXAMPLE

```
EDGES      1      #DNRZ used throughout
EDGES      2      # R0/R1 used throughout
EDGES      clocks 2 # R0/R1 used only on clock domain
```

### 3.8.4 EQUATION

This directive is used to define whether single or multiple equation sets will be created. SINGLE (common) or MULTIPLE equations 1 per pattern. By default SINGLE mode is used as this produces the smallest and cleanest timing setup. Multiple would be used if the user specifically requires tuning on each pattern to be independent.

*Syntax:*

**EQUATION** *SINGLE|MULTI*

*Example*

EQUATION	MULTI
EQUATION	SINGLE

### 3.8.5 NORMALIZE

This variable is used to provide an automatic update to the normalization check box of the GUI or command line. This value can be overridden by the GUI if the box is checked after loading of the CFG. This provides a default value so the user does not have to always remember to check it if desired.

*Syntax:*

**NORMALIZE** *ON|OFF*

*Example*

NORMALIZATION	ON
NORMALIZATION	OFF

### 3.8.6 OPTIMIZE

This variable is used to provide an automatic update to the Optimization level pulldown box of the GUI or command line. This value can be overridden by the GUI if the box is updated after loading of the CFG. This provides a default value so the user does not have to always remember to check it if desired. It can also provide a way of self documenting what options are required

*Syntax:*

**OPTIMIZE** 0|1|2|3

Where,

0 = No Optimization. Additionally, existing loops in source will be expanded

1 = Timing is optimized, but pattern compression is disabled. Pre-existing loops in patterns will be left alone

2 = Timing is optimized, AND pattern compression is enabled

3 = Timing is unoptimized so each pattern will retain individual timing. Pattern Compression is enabled

*Example*

OPTIMIZATION	1
OPTIMIZATION	2

### **3.8.7 PERIOD**

The PERIOD definition specifies the time period used for “cyclizing” a VCD or EVCD pattern. A Configuration File can include one or more PERIOD definitions. In the case of multiple definitions, each definition will apply to a different group of Pins to be defined in the subsequent PINLIST block.

Velocity will attempt to divide the VCD/EVCD event stream into the specified period, and determine the resulting drive, tri-state, and compare edge delays within the period.

The period that is specified by a PERIOD definition is also known in Velocity as a **Clock Domain**. The term Clock Domain comes from the fact that devices with synchronous, digital functionality typically have a group of signals whose timing is referenced to a particular clock signal. Therefore, those signals can share the same test system period as the clock. Some devices have multiple clocks operating at different rates, each clock having an associated group of signals synchronized with it. Each group of signals synchronized to a different clock can be said to belong to a separate Clock Domain.

Optionally, each Period / Clock Domain definition can take a name as a second parameter. This name can be used within the subsequent PINLIST block to reference the Clock Domain on a Pin-by-Pin basis. That is, each Pin in the PINLIST can be assigned to a Clock Domain independently of other Pins.

*Syntax:*

**PERIOD** *cycleTime* *clockDomain*

where,

*cycleTime* ::= *timeValue*[*timeUnit*]

where,

*timeValue* is a numerical value expressed in integer, floating point, or scientific notation

*timeUnit* ::= [*scaleFactor*]**s**

where,

*scaleFactor* is one of the following scaling characters:

**T** means Tera, or 1E12

**G** means Giga, or 1E9

**M** means Mega, or 1E6

**k** means kilo, or 1E3

**m** means milli, or 1E-3

**u** means micro, or 1E-6

**n** means nano, or 1E-9

**p** means pico, or 1E-12

**f** means femto, or 1E-15

*clockDomain* is a character string

*Example:*

PERIOD	1608ps	domain622
--------	--------	-----------



Note: The time value parameter can include units immediately after the number (no whitespace in between). Units can include all the common scaling letters, such as n (for nano), u (for micro), m (for milli), etc. Also note that the name “domain622” has been assigned to the Clock Domain.



Note: There must be a domain name defined for each PERIOD statement. Without the domain assignment the defined period it would never be assigned to any pins.

### 3.8.8 DRIVE Block

When EVCD or VCD files are used as inputs, strobcs will be automatically placed at the end of cycles (when snapping is enabled) or at the exact point of transition (when snapping is disabled). Often this is not ideal. The DRIVE block here will allow the user to arbitrarily assign the drive points on a pin by pin basis. Additionally, this will result in spec variables that will be added that can clearly control this from the tester.

*Syntax:*

```

DRIVE
    pinName1    [driveValue]
    pinName2    [driveValue]
    ...
    pinNameN    [driveValue]
END DRIVE

```

Strobe values can be expressed either as raw time values, which default to ns if no units are expressly defined. Alternatively, these can be expressed as % values. In that case, the edge strobe will occur at a particular ratio of the period. Every pin can have its own unique strobe location. Pins that are not defined in the block will have their strobcs occur at the regular default location that Velocity calculates.

If the driveValue is left out, then the spec will be assigned a value based on the value as present in the simulation. VCD/EVCD files will calculate the edge and this value will be used for the spec value.

*Example:*

```

DRIVE
    TCK    40%    #    drive value for TCK at 40% of the tester period
    TDI          #    create a SPEC for TDI, but define value with simulation
END DRIVE

```

### 3.8.9 STROBE Block

When EVCD or VCD files are used as outputs, strobes will be automatically placed at the end of cycles (when snapping is enabled) or at the exact point of transition (when snapping is disabled). Often this is not ideal. The STROBE block here will allow the user to arbitrarily assign the strobe points on a pin by pin basis. Additionally, this will result in spec variables that will be added that can clearly control this from the tester.

*Syntax:*

```

STROBE
    pinName1    strobeValue
    pinName2    strobeValue
    ...
    pinNameN    strobeValue
END STROBE

```

Strobe values can be expressed either as raw time values, which default to ns if no units are expressly defined. Alternatively, these can be expressed as % values. In that case, the edge strobe will occur at a particular ratio of the period. Every pin can have its own unique strobe location. Pins that are not defined in the block will have their strobes occur at the regular default location that Velocity calculates.

If the driveValue is left out, then the spec will be assigned a value based on the value as present in the simulation. VCD/EVCD files will calculate the edge and this value will be used for the spec value.

*Example:*

```

STROBE
    TDO    40%    #    Strobe the TDO at 40% of the tester period
    TDO2   #    create a SPEC for TDO2, but define value with simulation
END STROBE

```

### 3.8.10 SURROUND

This directive is used to enable or disable the use of surround-by waveforms. By default, surround-by will be enabled. But, there are instances where this is not wanted. For example, sometimes the surround-by will introduce extra edges. Other times, the surround-by will use too many edges. This provides a simple way of disabling the feature if it is not wanted

*Syntax:*

```
SURROUND ON/OFF
```

*Example*

```
SURROUND          ON
```

### 3.8.11 RUNNINGCLOCKS

There are 2 modes of use for the RUNNINGCLOCKS. This block can either be used to pre-define running clocks independent of the incoming source format. The second format is used to modify running clocks as passed in from STIL FreeRun statements

Pre-defined running clocks:

Free running clocks are pins that are designated as free running and ignore the pattern data during the conversion. Typically these are attached to clock ports (CLOCK). A clock pin will have one of two states, pulse high, 1 or pulse low, 0.

*Syntax,*

```
RUNNINGCLOCKS
    pinName activeDataState
END RUNNINGCLOCKS
```

Where,

pinName: a pin previously defined in the PINLIST

activeDataState: This will be "0" or "1". "0" means return to one. "1" means return to

zero

EXAMPLE

```
RUNNINGCLOCK
    CP 1
END RUNNINGCLOCK
```

Tuning of STIL FreeRun:

Within STIL, there exists syntax that can be used to define freerunning clocks. In some cases, these clocks will be defined in stil as positive or negative pulses at a certain period. Withing Velocity the RUYNNINGCLOCK statement can be configured as a single line to only allow POS or NEG pulses. When this syntax is used all pulses will use the polarity assigned in the CFG as an override to whatever polarity is defined by the STIL syntax

EXAMPLE

RUNNINGCLOCK	POS	# 1 pulses only
RUNNINGCLOCK	NEG	# 0 pulses only

### 3.8.12 GLOBALSPEC

When multiple domains are used, there will be a separate spec defined to control the PERIOD for each domain. If these are clean multiple of one another it is possible to set this up so that a single variable can be used to control the period. Equations would then be used to scale the period from port to port.

Normalization must be used with this option. If GLOBALSPEC is enabled and Normalization is disabled, a warning will be issued. The translation will continue and Normalization will be automatically enabled anyway.

This spec is often used in combination with the Domain Tracking feature that documented in the next section. Domain tracking can be used to assign the relative multiples of each defined time domain.

Syntax,

GLOBALSPEC      ON/OFF

Where,

ON = one SPEC variable will be used and equations will be used to automatically expand from domain to domain.

OFF = separately controlled PERIOD spec will be used for each domain

EXAMPLE

GLOBALSPEC	ON
GLOBALSPEC	OFF

## 4.0 CUSTOM LEVELS



**BACKGROUND:** Simulation output files, and even STIL files, do not typically define DC levels for the signals. However, using configuration file structures, Velocity provides you with a way to include levels information with your auto-generated test program.

The LEVELS block allows you to define, for any pin or group of pins, power supply levels, input drive levels, and output threshold levels.

4.0.1 To define levels for a group of pins, create the following Control definition block.

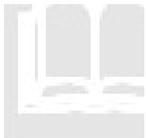
- On the first line, use the keyword **LEVELS** followed by a pin or group name. Optionally, you can use the word **default** for the pin specification to indicate all pins.
- On the next line, use the keyword **POWER** followed by a voltage value. This will be the master power supply voltage level.
- On subsequent lines, use the following keywords followed either by a voltage value or a percentage:

**VIH** – Input voltage for a logic high

**VIL** – Input voltage for a logic low

**VOH** – Output threshold voltage for a logic high

**VOL** – Output threshold voltage for a logic low



**BACKGROUND:** If you specify a level as a percentage, Velocity interprets it as a percentage of the POWER level. This provides a convenient way to scale levels with a device power supply voltage.

- For the last line, use the keywords **END LEVELS**.

The following is an example of a Levels definition:

```
LEVELS default
POWER 3.0V
VIL 0.8V
```

```

VIH  2.0V
VOL  30%
VOH  50%
END LEVELS

```

## 4.0.2 Power Sequences

This section is used to define the power up sequence. Although this section is technically optional, It is strongly suggested that this section be used. Otherwise, the power up will require user intervention in multiple locations in the source files. Syntax of this block is as follows

```

POWER powerStateName
    SupplyName    supplyVoltage    clampCurrent    delayAfter
    SupplyName    supplyVoltage    clampCurrent    delayAfter
    SupplyName    supplyVoltage    clampCurrent    delayAfter
    “”              “”              “”              “”
    “”              “”              “”              “”
END POWER

```

When used, this power sequence can be referenced in the same way that test's (defined next) are used. In other words, this block is treated as a special case of the tests that will allow execution with name pass/fail queries. The power sequence will always result in a pass value and will never log anything.

If a staged power up or power down sequence is required. This can be defined by generating multiple power blocks with unique names for each stage. Or, it can be defined explicitly within a single power block by defining the supply more than once in the block.

- Each entry in the power block is executed serially in the order is defined in the configuration.
- Multiple supplies can be referenced within a single block
- At the end of execution, supplies will retain the supply value last requested

The following is an example of a Power definition:

```

#####
# Power up and power down
#####
POWER nominal
    VS1  1.25V  500mA  5uS
    VS2  3.6V   500mA  0uS
    VS3  3.6V   500mA  0uS
END POWER

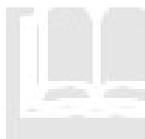
```

### **4.0.3 Power down sequencing**

There will always be a power sequence named “off” created by the ShellConstructor. This default sequence will do nothing more than disconnecting the power supplies. This default sequence will be overridden in the following cases.

- Power off case 1: If only one power up sequence is defined, the power off sequence will be assumed to occur in reverse order. Each supply will be set to 0V and then disconnected in the reverse order of the power up
- Power off case 2: If multiple power up sequences are defined, the power off will default to the reverse order of the last power sequence. Each supply will be set to 0V and then disconnected in the reverse order of the power up
- Power off case 3: If a special power down that is not explicitly equivalent to one of the above, a special POWER block named “off” can be defined that will automatically override the default case. **This is the recommended method**

## 5.0 CUSTOM TIMING



**BACKGROUND:** Although Velocity will create appropriate Time Sets for your program, based on the simulation or ATE files used as source for the conversion, you can create your own custom timing to apply to tests.

To define custom timing for a group of pins, create the following Control definition block.

- On the first line, use the keyword **TIMING** followed by a pin or group name. Optionally, you can use the word **default** for the pin specification to indicate all pins.
- On the next line, use the keyword **PERIOD** followed by a time value. This will be the period of the tester's pattern sequencer.



**BACKGROUND:** All **TIMING** blocks in a particular Configuration file must use the same **PERIOD** value. This ensures that the tester will be able to use the resulting STIL file.



**TIP:** In order to use **TIMING** blocks with different **PERIOD** values in your test program, use separate Configuration files for each of the different periods and run separate conversions with each.

- On subsequent lines, use the following keywords followed either by a time value or a percentage:

**DRIVE** – Time delay of a drive edge for a pin of type I or IO

**RECEIVE** – Time delay of a compare edge for a pin of type O or IO

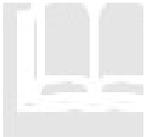
**PULSE** – Duration of a pulsed waveform for a pin that is not defined as a clock pin.

**OFFSET** – Time delay of first edge for a pin of type CLK

**RISE** – Time delay of second edge for a pin of type CLK, if a rising edge

**FALL** – Time delay of second edge for a pin of type CLK, if a falling edge

**DUTY** – Duty cycle for a pin of type CLK, expressed only as a percentage



**BACKGROUND:** If you specify a timing parameter as a percentage, Velocity interprets it as a percentage of the PERIOD time. This provides a convenient way to scale edge delays with a sequencer period.

- For the last line, use the keywords **END TIMING**.

The following is an example of a Timing definition:

```
#####
# Timing
#   These definitions will define the values of specs
#   values will be assigned by default.  Groups and pins can
#   be defined to override defaults by using a pin or group
#   name.
#####
TIMING default
  period 100ns
  offset 0ns
  duty 50%
  drive 25%
  receive 90%
END TIMING

# redefine the data pin and use "dataBus" as the spec bsae name for the
#   actions
TIMING dataBus d7 d6 d5 d4 d3 d2 d1 d0
  drive 15%
END TIMING
```

## 5.01 TEST Definitions

This section is used to create specific test instances. Each defined test will be accessible from both the main test program and command line execution scripts. The general syntax for the section is as follows. Each entry is then detailed

```

TEST testname testnumber
    TYPE           lib.family.testName
    PATTERN        patternName
    TIMING         timing declaration
    LEVELS        levelsDeclaration
    PARAMETERS
        PARM1      PARM2value
        " "          " "
        " "          " "
    END PARAMETERS
    LIMITS
        .           <=   testName1   <=   .           [units]
        .           <=   testName2   <=   .           [units]
    END LIMITS
END TEST

```

**TEST:** Keyword to tell the Velocity that a new test block is being created. This then requires that a unique test name and (optionally) a unique test number to follow. The testname will be the name as accessed by the command line execution script. Each test must have a unique name. The test number will provide a starting testnumber for every element logged. The test numbers should be unique and enough separated from one another so that tests will multiple events will not step on one another.

**PATTERN:** Pattern Block name that is to be executed. Note: This pattern name refers to the block name which may be different from the STIL file name that is derived from. The user must know the exact name for this to be valid. This parameters associated value will be case sensitive. Alternatively, "\$default" can be used for the pattern's name and the pattern will be chosen automatically from the input pattern list.

**TIMING:** Optionally defines the Timing Block that should be used for a given test. This can be left out an automation will automatically apply timing based on the patternList compilation. This is usually left out an automatically applied based on compilation results.

**LEVELS:** Optionally defines the Level Block that should be used for a given test. This is defined in terms of level set numbers in the case of single port. Or by multiport spec name if using multiport. This block is usually left blank and auto filled from the available levels.

TYPE: The keyword should be the first subparameter of each TEST block. This will tell the Velocity what type of generic function is to be applied. Depending on which parameter type is received, a different set of parameters will then be required. The typeKeyword can be included from any library as long as the library is part of the string applied in the CFG.

If no TYPE is specified at all for a TEST block, it will be assumed to be a straight functional test.

For a complete list of automatically supported test methods consult your Advantest Test Documentation Center. A few common examples are included here.

The format is of the form “library.class.testName”.

where,

library = the name of the test method library which is included with the test program  
class = an optional sub class within the library. You may not have a multiple classes within a library. When this happens you can drop the class designator entirely.  
testName = The name of the test method function that is to be executed.

- **ac\_tml.AcTest.FunctionalTest:** Functional Test executes a digital pattern and responds with pas/fail results
- **dc\_tml.DcTest.Continuity:** Continuity Test - tests a lists of pins for connectivity by examining voltage seen when small current is applied to pin with no power applied
  - **pinlist *pinNames:*** comma delimited list of pins or groups to be included in the test
  - **testCurrent *currentValue:*** force current applied to each of the pins in the pinlist
  - **settlingTime *settlingTimeValue:*** settling time after force current is applied before the voltage is measured.
  - **measurementMode *PPMUpar|ProgLoad:*** This is one of two options that will determine the type of measurement that is being done. PPMUpar will use the PMU per pin. The ProgLoad will use the programmable load and regular measurement pin electronics and can be done in parallel.
  - **polarity *SPOL|BPOL:*** This argument will choose whether single polarity or both polarities are used for the measurement.
  - **prechargeToZeroVol *ON|OFF:*** Determines whether the pinlist will be precharged to 0 volts prior to the test applying the force current
  - **testName *passVolt\_mV:*** This is the name of the test as it will appear in the datalog and how it will be connected to limits. For Advantest supplied version of this test, the testName can't change. But, this field is provided in the CFG so in case the user wishes to modify the default method to change the logging.
  - **output *None|ReportUI|ShowFailOnly:*** This determines the level to which data will be sent to the default report window. None will report nothing. ReportUI will report everything. ShowFailOnly will display only the failing pins.

- **dc\_tml.DcTest.ProductionIddq**: Production IDDQ Test - tests a lists of pins for connectivity by
  - **dpsPins** *pinList*: list of DPS pins that will be included in measurement.
  - **disconnectPins** *disconnectPinList*: List of pins that need to be disconnected during measurement. If left blank, then all pins will remain conected
  - **settlingTime** *settlingTime*: settling time before measurements will start
  - **stopMode** *ToStopVEC|ToStopCyc*: determines with the test will stop at a given vector number or at a certain cycle number
  - **strStopVecCycNum** *stopCycleNumber*: stop value for test. If left blank, this will execute to the end of the pattern specified
  - **samples** *numSamples*: Number of samples per output current measurement
  - **checkFunctional** *ON|OFF* : determines whether the functional tests's pass/fail result will be included in the result analysis or not
  - **controlTestNumOfFunctional** *ON|OFF* : If the functional result is being used as part of the pass fail, this argument will tell the method whether or not to use a separate test number for the functional pass/fail or not
  - **gangedMode** *ON|OFF* : enables or disables the ganging of supply channels during measurement.
  - **testName** *passCurrLimit\_uA*: This is the name of the test as it will appear in the datalog and how it will be connected to limits. For Advantest supplied vesion of this test, the testName can't change. But, this field is provided in the CFG so in case the user wishes to modiy the default method to change the logging.
  - **output** *None|ReportUI|ShowFailOnly*: This determines the level to which data will be senbt to the default report window. None will report nothing. ReportUI will report everything. ShowFailOnly will display only the failing pins.

- **dc\_tml.DcTest.Leakage:** Production IDDQ Test - tests a lists of pins for connectivity by
  - **pinlist** *pinList*: list of IO pins that will be included in measurement
  - **measure** **LOW|HIGH|BOTH** : determines which polarity to measure for each pin
  - **measurementMode** **PPMUpar|ProgSer|SPMUser**: 3 options to define whether each measurement will be made ins serial using the SPMU. In serial using the PPMU, or in parallel using the PPMU. These three options allow the user to choose precision vs speed of measurement.
  - **relaySwitchMode** **DEFAULT(BBM)|MBB|Parallel** : 3 options for defining the relay switching mode to use for each measurement.
  - **forceVoltageLow** **forceLowValue** : optional value to apply to the pins while measuring the low leakage.
  - **forceVoltageHigh** **forceHighValue** : optional value to apply to pins while measuring the high leakage
  - **spmuClampCurrentLow** **lowClampValue** : If the SPMU measurement mode is used, this value will be used for the low clamp value. Can be left blank otherwise.
  - **spmuClampCurrentHigh** **highClampValue**: If the SPMU measurement mode is used, this value will be used for the high clamp value. Can be left blank otherwise.
  - **ppmuPreCharge** **ON|OFF** : determines with the pre charge value will be applied before each measurement.
  - **prechargeVoltageLow** **lowPreChargeValue** : low value to be used for pre charge if enabled.
  - **prechargeVoltageHigh** **highPreChargeValue** : : high value to be used for pre charge if enabled.
  - **settlingTimeLow** **lowSettlingTime** : settling time before low value measurement is made
  - **settlingTimeHigh** **highSettlingTime** : settling time before high measurement is made
  - **preFunction** **YES|NO** : Allows a separate pre Functional pattern to be excecuted. If NO, then no functional test will be applied.
  - **controlTestNumOfFunctional** **YES:NO** : If functional test is used, then this allows the results of that functional test to use a separate test number or not in the datalog.
  - **stopCycVecLow** **stopLowLocation** : Optionally defined stop cycle for the low value measurement. Will run to the end if left blank
  - **stopCycVecHigh** **stopHighLocation** : Optionally defined stop cycle for the high value measurement. Will run to the end if left blank
  - **testName** **passCurrLimit\_uA**: This is the name of the test as it will appear in the datalog and how it will be connected to limits. For Advantest supplied vesion of this test, the testName can't change. But,

this field is provided in the CFG so in case the user wishes to modify the default method to change the logging.

- **output** *None|ReportUI|ShowFailOnly*: This determines the level to which data will be sent to the default report window. None will report nothing. ReportUI will report everything. ShowFailOnly will display only the failing pins.
- **dc\_tml.DcTest.OperatingCurrent**: Operating Current Test - tests a list of pins for connectivity by
    - **dpsPins** *pinList*: list of DPS pins to be included as part of measurement
    - **samples** *numSamples*: number of samples per
    - **delayTime** *delayValue*: delay after connect and pattern execution before measurement will be made
    - **termination** *OFF|ON*: Flag to turn termination of IO channels on or off
    - **testName** *passCurrLimit\_uA*: This is the name of the test as it will appear in the datalog and how it will be connected to limits. For Advantest supplied version of this test, the testName can't change. But, this field is provided in the CFG so in case the user wishes to modify the default method to change the logging.
    - **output** *None|ReportUI|ShowFailOnly*: This determines the level to which data will be sent to the default report window. None will report nothing. ReportUI will report everything. ShowFailOnly will display only the failing pins.

There are many other automatically provided methods. You can also include references to custom defined API libraries. At run time, Velocity will assume that the library has been included with the targeted test program. As long as the "lib", "class", and "testName" are properly setup, then the testflow will have access to these functions. Note that some test method libraries do not have multiple classes of tests. If this is the case, then the Velocity TYPE specified will be of the form "**library.testname**" with no "family" designator at all.

The following is a sample Test Blocks section that defines a number of tests. Specifically, this list of definitions will result in 8 specifically accessible test Functions defined in TestFunctions.cpp using the generic AC and DC test functions defined in GenericFunction.cpp. There will then be 8 script execution functions defined in user\_commands.cpp. These functions are also available to the Flow Block section of the configuration defined below which can be used to create instances of these functions in a user defined order in the "main" program.

```
#####
# Test Definitions
#####

TEST shortsPositive 100
  TYPE      dc_tml.DcTest.Continuity
  PARAMETERS
    pinlist  all
    testCurrent  10[uA]
    settlingTime  1[ms]
    measurementMode  PPMUpar
    polarity  SPOL
    prechargeToZeroVol  ON
    testName  passVolt_mV
    output  None
  END PARAMETERS
  LIMIT
    .  <=  passVolt_mV <=  .  []
  END LIMIT

END TEST

TEST contNegative 150
  TYPE      dc_tml.DcTest.Continuity
  PARAMETERS
    pinlist  all
    testCurrent  -10[uA]
    settlingTime  1[ms]
    measurementMode  PPMUpar
    polarity  SPOL
    prechargeToZeroVol  ON
    testName  passVolt_mV
    output  None
  END PARAMETERS
  LIMIT
    .  <=  passVolt_mV <=  .  []
  END LIMIT

END TEST

TEST IDDQdouble 500
  TYPE      dc_tml.DcTest.OperatingCurrent
  PATTERN  juno_soc_aplpll_x5
  PARAMETERS
    dpsPins  @
    samples  4
    delayTime  0[ms]
    termination  OFF
    testName  passCurrLimit_uA
    output  None
  END PARAMETERS
  LIMIT
    .  <=  passCurrLimit_uA <=  .  []
  END LIMIT

END TEST
```

```

TEST LeakageHi 250
TYPE      dc_tml.DcTest.L Leakage
PARAMETERS
  pinlist      GROUP_defaultInputs
  measure      BOTH
  measureMode   PPMUpar
  relaySwitchMode  DEFAULT (BBM)
  forceVoltageLow  400 [mV]
  forceVoltageHigh 3800 [mV]
  spmuClampCurrentLow  0 [uA]
  spmuClampCurrentHigh 0 [uA]
  ppmuPreCharge  ON
  prechargeVoltageLow  0 [mV]
  prechargeVoltageHigh 0 [mV]
  settlingTimeLow  0 [ms]
  settlingTimeHigh 0 [ms]
  preFunction      NO
  controlTestNumOfFunctional  NO
  stopCycVecLow   0
  stopCycVecHigh  0
  testName        (passCurrentLow_uA,passCurrentHigh_uA)
output           None
END PARAMETERS
LIMIT
  . < passCurrentLow_uA < . []
  . < passCurrentHigh_uA < . []
END LIMIT
END TEST

TEST IDDQ 400
TYPE      dc_tml.DcTest.ProductionIddq
PARAMETERS
  dpsPins      @
  disconnectPins
  settlingTime  0 [ms]
  stopMode      ToStopVEC
  strStopVecCycNum
  checkFunctional  ON
  controlTestNumOfFunctional  OFF
  gangedMode     OFF
  testName       passCurrLimit_uA
  output         None
END PARAMETERS
LIMIT
  . < passCurrLimit_uA < . []
END LIMIT
END TEST

```

## 5.02 FLOW Definitions

The testflow will insert a predefined set of tests in a particular order into the main program of the test. Each named test or power setting must be defined in prior to use or compilation errors will occur. This is the syntax for the section

```

FLOW flowName
    TEST|POWER|DELAY testname|powerSequenceName|delayValue [failBinNumber]
    TEST|POWER|DELAY testname|powerSequenceName|delayValue [failBinNumber]
    ,
    ,
END FLOW

```

*FLOW* is a keyword that indicates the beginning of a flow block

Only one *FLOW* block should exist within a given test configuration. No errors will be seen but only the last flow listed will be inserted into the test program

*testname* and *powerSequenceName* must explicitly match a *TEST* or *POWER* block defined prior to the *FLOW* block

*DELAY* will insert delays in resulting test flow. There must be a number following the *DELAY* statement

The example below assumes that the 2 tests and 1 power sequence have already been defined. The “off” power sequence can either be explicitly defined or implied as being defined because it will automatically be generated because as the reverse of the defined power sequence.

The following is an example of a Flow definition:

```

#####
# Flow Definition
#   The following tests will be executed in the following
#   order.  If no flow is defined, then all the tests will
#   be included in the order they are defined.  All will
#   be called inside user_main
#####
FLOW experimentName
    TEST    contNegative  10
    POWER   nominal
    TEST    funcSpec     5
    DELAY   15ms
    POWER   off
END FLOW

```

## **6.0 CUSTOMIZING PATTERNS**

Custom patterns are patterns that are created based on existing patterns but with additional sequencing features such as loops and breaks. By default, every custom pattern will have a base pattern that it is initially created from. After creation, the user can inject an arbitrary list of additional loops and branches to allow for varied execution of the predefined pattern. Therefore, this provides a simple way of automatically introducing modified execution of patterns when it is known beforehand that such changes should occur.

First, a new pattern file will be created as a copy of the base pattern. Any labels that are used within the original will be renamed automatically so that they are unique in the copied version.

Second, any new sequences that are requested will be added to the new STIL file. Once compiled, they will be visible to the tester's Pattern viewer.

Third, multi-port burst blocks will automatically be created for the new pattern if necessary. TEST blocks can then refer to just the pattern name. The pattern burst will be implied. The timing associated with this pattern will be identical to that of the original base pattern. Therefore, no extra work will be required to force timing. This can always be changed later. The custom pattern block itself, as stated in the introduction, is not meant to be the main user interface. But, rather, it is meant to provide a quick start for new test programs.



**BACKGROUND:** If your Velocity package includes Optimization options, Velocity can automatically search for compression opportunities when converting patterns, and create appropriate repeats and loops in your patterns.

However, even without Optimization, you can manually customize your pattern files using Configuration control. You can specify explicitly not only repeats and loops, but also selective output masking (pin-by-pin and cycle-by-cycle), pattern truncation, etc.

## 6.1 Pattern Syntax

The following syntax is used for the PatternBlock

```
PATTERN newPatternName  
           BASE           basePatternName  
           Command       commandParameterList  
           Command       commandParameterList  
END PATTERN
```

**PATTERN:** Keyword that tells the ShellConstructor that this is the beginning of a custom pattern block.

*newPatternName:* Must be a unique string to identify the name of the new pattern. This name can be used by subsequent TEST blocks.

## 6.2 BASE Syntax

**BASE:** Keyword to indicate that the new pattern is associated with a given base pattern.

*basePatternName:* This base pattern must be included in each PATTERN block program. The name of the base pattern must explicitly match the name of one of the original source files being translated. If no base is present, then any subsequent actions will not be applied properly.

## 6.3 Command and Parameter List Syntax

The commands and associated parameters are an optional member of the Pattern Block. However, there will generally be at least one command inserted. Otherwise, there is no real reason to create the custom pattern in the first place. There is no upper limit on the number of inserted commands that can be used. However, when using loops is important that these not be built to interleave. Only one level of looping is defined in this syntax. The following actions can be inserted

### 6.3.1 TYPE (*optional*)

This optional parameter can be used to determine whether the pattern is to be compiled as a regular pattern or if it is to be defined as a subroutine. (This is only available for the Advantest target port). Other ports will revert to the default type which will compile that patterns as regular functional patterns

**TYPE**    *MAIN|SVEC*

### 6.3.2 DOMAIN (*optional*)

This optional parameter can be used to lock a set of commands to a particular time domain. If multiple domains are used, then cycle counts that are used to define custom start, stop or loop parameters would need adjustment. This command assigns the reference domain to be used. For the subsequent list of commands

**DOMAIN**    *domainName*

### 6.3.3 FUNC (*optional*)

This optional pattern allows the user to insert a predefined bits stream to particular pin or set of pins. There are a list of predefined bit patterns that can be applied such as PRBS patterns or you can define with a hard path to a file name. This bit stream will be applied to a user defined cycle starting point and can be repeated as any times as desired

```
FUNC patternName[filePath pinOrGroupName startCycle  
[repeatCount]
```

The pin or group name must be defined above in the pin or group section. The usage of the FUNC keyword must occur after the BASE pattern has been defined. Otherwise, there will be nothing to attach this inserted cycles to. All pins that are not directly referenced by the FUNC statement will be treated as repeats of the previous cycle. If the start cycle is greater than the length of the base pattern, a warning will be thrown and the bit stream will be applied to the end of the pattern. If a repeat count is used and the bit stream itself is not a modulus of the data bit rate, then the pattern will be appended with continuation bits so that it is proper modulus.

### 6.3.4 LOOP

Loops can be added with the following syntax

```
LOOP startCycle,stopCycle [ loopCount ]
```

The start and stop cycle refer to the vector number of the beginning and ending of the inserted loop. A loop count is optional. If not defined, the loop count will be defined as infinite. The loop will have to be stopped by pressing the “abort” button in ITE, as the loop will be interpreted as infinite.

### 6.3.5 REPEAT

Single line repeats can be added with the following syntax

**REPEAT**    *cycle,loopCount*

Cycle defines the vector number for a single line repeat. LoopCount defines the number of times that line should be executed. A loop count of 1 would be equivalent to not having the REPEAT command in the first place.

### 6.3.6 MATCH

Match Loops can be inserted with the following syntax:

**MATCH**    *startCycle,stopCycle [jumpLocation]*

A match loop will execute until the entire range of the loop passes on all cycles. startCycle and stopCycle refer to the vector location of the beginning and the ending for the loop. Optionally, a jump location can be defined with the last argument. If used the pattern execution will jump to the given location after a match is found. If not used, the pattern will continue at the next line

### 6.3.7 START

The start location for a given pattern can be redefined with this syntax:

**START**    *newStartVector*

The start location for a given vector can be redefined with this command. The new pattern will have all previous vector information removed so that the new start location will occur at the vector defined by the parameter newStartVector.

### 6.3.8 STOP

The stop location for a given pattern can be redefined with this syntax:

**STOP** *newStopVector*

The stop location for a given vector can be redefined with this command. The new pattern will have all subsequent vector information removed so that the new stop location will occur at the vector defined by the parameter newStopVector.

### **6.3.9 WAIT**

The WAIT variable will allow you to insert arbitrary time delays at any point in a pattern. These can be inserted at time values or by cycles. If no units are specified for the location it is assumed to be a cycle number

WAIT location[units] duration

### 6.3.10 CUSTOM PATTERN EXAMPLES

The following is an example of a Pattern definition:

```
#####
# Pattern lists
#   The following patterns will be translated.  If the pattern is
#   not in the list, then it will be skipped.  If the pattern is
#   not in the source file then a warning will be issued.
#####
PATTERN loopInfinite
  BASE SpecFunc
  LOOP 5,20
#END PATTERN

PATTERN loopFinite
  BASE SpecFunc
  LOOP 5,18 16
END PATTERN

PATTERN multipleLoop
  BASE SpecFunc
  LOOP 5,10 16
  LOOP 16,20 16
END PATTERN

PATTERN changeStartStop
  BASE SpecFunc
  start 5
  stop 20
END PATTERN

PATTERN PRBS7
  BASE SpecFunc
  FUNC PRBS7  dataIn 800 8
END PATTERN

PATTERN delaysAdded
  BASE SpecFunc
  DELAY 100us 5ms
  DELAY 1ms 2ms
END PATTERN
```

## 6.4 Logical Masking

This keyword will tell the mask loader what command is being requested. These masks are build with a syntax that allows conditional logic to be applied to enable and disable the masking as well as syntax to define how characters are remapping.

Masks can be turned on and off by cycle or pin by pin. **Each command is terminated by a semicolon** at the end of the line. This allows complicated or long statements to be spread over multiple lines

The mask block can also be defined by itself outside of the PATTERN block. In this case the MASK block itself is given a name. If “default” is the name, then the contents of the MASK block are applied to all patterns that are loaded. Any other name will apply the mask only to patterns that match the name of the mask block

```
MASK  default|inputPatternName
        maskCommand maskPinList [map] [conditions]
        maskCommand maskPinList [map] [conditions]
END MASK]
```

It is legal to have both a default and a specifically applied mask for a single pattern. When both are defined, the mask that is specific to the given pattern will be applied first, followed by the default mask block. In the end both are applied. If the input pattern name does not match the name of the mask block, then that mask block is not applied.

### 6.4.1 PINS

It may be desirable to handle each pins masking separately or collect all masking conditions in a single statement. The syntax is as follows

```
PINS pin1,pin2...pinN start1-stop1,start2-stop2 [condition] [map];
PINS pin1,pin2...pinN start1-END [condition] [map];
PINS pin1,pin2...pinN ALL [condition] [map];
```

In this case, all of the starts and stops for a masking scheme are expressed in a single comma delimited list. Start and stop pairs are separated by the “-“ (dash).

Each start and stop must be an integer that corresponds to a valid cycle number in the loaded pattern. The only exception to the integer limitation is the use of END which all apply the active masked region all the way to the end of a pattern. The other is the use of “ALL”, which will apply the mask to every cycle in the active pattern (Conditions explained below)

### 6.4.2 MAPS

Character remapping:: By default a mask will take all L, H, and M characters and recast them as X characters. However, the MAP keyword can be used within a mask definition to reassign the state character mappings to any other combination of states. The map conditions specific will override the default. For example, you could use the MAP to turn off drive values. You might even use the map to swap 0 and 1 characters if you want to invert a signal. The MAP is applied with the following sequence

*{MAP sourceChar|targetChar}*

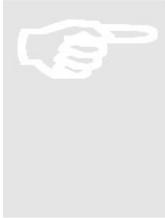
- sourceChar: This can be one or more state characters that might be present in the unmasked source vector.
- targetChar: This can be one or more state characters that would be used to replace the list of characters in the sourceChar listing. If only one character is provided, this character will be applied as the target for all of the states in the sourceChar. If more than one is listed, then the list MUST be the same length as the sourceChar list. The mapping will occur in a 1 to 1 fashion in the same order.

Examples:

{MAP HM:XX} This will turn off all compares

{MAP 01LH:10HL} This will invert all signals, input and output.

{MAP 01LH:ZZXX} This will turn off drives and compares.



Note that the ordering and count of the characters in the the source and target listing for the mask mapping section follows the same convention as STIL waveform tables. The order of the source will match the order of the target. If there is only one target char it will apply to all source chars



Note: Digital Capture can be setup using the MASK block by defining a MAP structure that remaps the L and H characters to C. { MAP [LH:C] } When this is done, the timing will automatically be adjusted to include the capture and don-t capture waveforms



Note the number of “from” states in the MAP string MUST be the same as the number of “to” states in the MAP



Note: The curly brackets surrounding the MAP statement are **required**.

### 6.4.3 CONDITIONS

Mask conditions can be used to fine tune the regions in which a mask is applied so to match conditions in the pattern on any pin at any state prior to the active cycle. Condition sequences will be analyzed as a comma delimited “or” of multiple conditions. Each condition is applied with the following sequence.

```
{COND refPin[relativeCycle]=pinState }
```

- **refPin:** This should be an explicit match to a pin in the given pin list. This can be equal to a pin the ON, OFF, or PINS state to which the condition is applied. Or, it may be equal to any other pin in the pin list as defined by the configuration
- **relativeCycle:** This parameter is optional. If no relative cycle is defined, it is assumed to be 0 and will search for the condition on the active cycle that is potentially being masked. This can also be a range of cycles. A positive number will look later in the pattern. A negative number will look at previous cycles.
- **pinState:** This will define that state for the refPin that activates the given mask sequence. If the reference pin is not explicitly equal to the given state, then the mask will be deactivated



Note: The curly brackets surrounding the COND statement are **required**.



Note: The condition can use “!=” as well as “=” to activate the actions when a condition is NOT true as well.

#### 6.4.4 PIN DUPLICATION

In certain instances it may be necessary to provide a complete duplication of one pin's data onto another pin. This is accomplished by setting one pin's equation to another pin in the PINS statement. This is the MASK block required because each simulation pin can connect to one and only one signal in the PINLIST block.

The closing semicolon is required

```
PINS targetPin=sourcePin;
```

#### 6.4.5 EXAMPLES:

```
PINS clkOut 0-100,1000-END;
```

- This will turn on masking for the pinNamed clkOut starting at cycle 0 and turning it off at cycle 100. After this the mask will be turned on again at cycle 1000 and will remain active until the end of the pattern because no OFF statement occurs

```
PINS clkOut 0-100,1000-END {COND RESET=0};
```

- This will turn on masking for the pinNamed clkOut starting at cycle 0 and turning it off at cycle 100. After this the mask will be turned on again at cycle 1000 and will remain active until the end of the pattern because no OFF statement occurs. Within the active ranges, the mask will only be active if the RESET pin is set to 0. Therefore, if RESET is at any other state during the range, the clkOut pin will not be masked.

```
PINS clkOut 0-100,1000-END {MAP H:X} {COND RESET[16]=0};
```

- This will turn on masking for the pinNamed clkOut starting at cycle 0 and turning it off at cycle 100. After this the mask will be turned on again at cycle 1000 and will remain active until the end of the pattern because no OFF statement occurs. Within the active ranges, the mask will only be active if the RESET pin is set to 0 16 cycles prior to the active cycle. In other words, the mask will be active until the RESET pin has been set high for at least 16 cycles. Lastly, Only the H's in the source will be masked. L's will be left alone

## 6.5 Serial Masking

Serial masking is an alternate method for applying arbitrary remappings of character sequences on a given pin. In general, anything that is defined as a serial mask can also be assigned with logical mask syntax as defined above. However, sometimes it is difficult to programmatically define the logical syntax. For that reason the serial masking block was created.

### 6.5.1 SYNTAX

```
GLOBAL MASK pinName1 [pinName2 pinName3 ... pinNameN]
    "sourceString" -> "targetString"
    "sourceString" -> "targetString"
    ...
    "sourceString" -> "targetString"

END GLOBAL
```

The block is initiated and terminated with "GLOBAL MASK" and "END GLOBAL". You can then apply this to one or more pins or groups. These pins or groups must have been defined already in the PINLIST or GROUP blocks.

The source and target strings will define what strings you are searching and replacing. These sequences are searched for vertically on each pin in the listing. If the source string is found, it will be automatically replaced with the target string.

The length of each source string must match the length of its associated target string.

Multiple source and target pairs are executed in the order they are defined inside the block

There can be more than one GLOBAL MASK block per configuration. That way you can assign different combinations of string pairs for different pins.

The serial masking blocks are ALWAYS executed before any logical masking block that has also been created.

## 6.5.2 EXAMPLES

```

GLOBAL MASK DQ DQS DQSb MDQ MDQS MDQSb
"MMMMMMMMMMMMMMMM0" -> "XXXXXXXXXXXXXXXXNNN0"
"MMMMMMMMMMMMMMMMZ0" -> "XXXXXXXXXXXXXXXXNNN0"
"MMMMMMMMMMMMMMMMZZ0" -> "XXXXXXXXXXXXXXXXNNN0"
"MMMMMMMMMMMMMMMMZZZ0" -> "XXXXXXXXXXXXXXXXNNN0"
"MMMMMMMMMMMMMMMMM1" -> "XXXXXXXXXXXXXXXXNNN1"
"MMMMMMMMMMMMMMMMMZ1" -> "XXXXXXXXXXXXXXXXNNN1"
"MMMMMMMMMMMMMMMMMZZ1" -> "XXXXXXXXXXXXXXXXNNN1"
"MMMMMMMMMMMMMMMMMZZZ1" -> "XXXXXXXXXXXXXXXXNNN1"
END GLOBAL

```

These will mask turn around for packets that are  
to close together to test

```

GLOBAL MASK DQ DQS DQSb MDQ MDQS MDQSb
"LLL0" -> "NNN0"
"LLH0" -> "NNN0"
"LHL0" -> "NNN0"
"LHH0" -> "NNN0"
"HLL0" -> "NNN0"
"HLH0" -> "NNN0"
"HHL0" -> "NNN0"
"HHH0" -> "NNN0"

"LLL1" -> "NNN1"
"LLH1" -> "NNN1"
"LHL1" -> "NNN1"
"LHH1" -> "NNN1"
"HLL1" -> "NNN1"
"HLH1" -> "NNN1"
"HHL1" -> "NNN1"
"HHH1" -> "NNN1"
END GLOBAL

```