Patterns, Waveforms, and Timing 2

Introduction

This chapter presents the basic information for creating the test patterns and the associated waveforms and timing relationships of an enVision™ test program. You should be able to create the pattern files for most applications after reading this chapter and referring to the chapters for *WaveTool*, *PatternTool*, and *PatternSequenceTool* in the *Tools* manual.

Before reading about the fundamentals of patterns, waveforms, and timing, read the simplified overview of the enVision software architecture and tools and of an enVision test program to get a better understanding of the enVision software components.

enVision Software

enVision Architecture

The enVision software is based on an object database, which contains the data and the methods that act on the data; see Figure 2.1. The database, which is created and maintained by the enVision Tools, is a set of data, like the information from the device's data sheet:

- DUT and its environment
- a test sequence
- Test values entered by the user
- Test values returned by the DUT
- Test methods describing the DUT test techniques, such as open / shorts tests, parame tric tests, timing tests, and functional pattern tests.

These test descriptions are object-oriented and device-oriented. They are similar to the underlying device tests, oriented like a device's data sheet; refer to Top Level Objects.

Tools

enVision provides GUI tools for entering and manipulating the device data and viewing the resulting test data. A test program developed in enVision consists of a set of interconnected data objects stored in the tester controller. Each data object can be viewed and manipulated by a tool.

The main enVision interface is the Operator Tool, which can load a enVision Test Program and start other enVision tools, among other features.

Another GUI tool, WaveTool, lets you graphically define digital waveforms in the same way that timing diagrams for waveforms appear in the device's data sheet.

For more information about a particular enVision tool, refer to the appropriate chapter in the *Tools* manual.

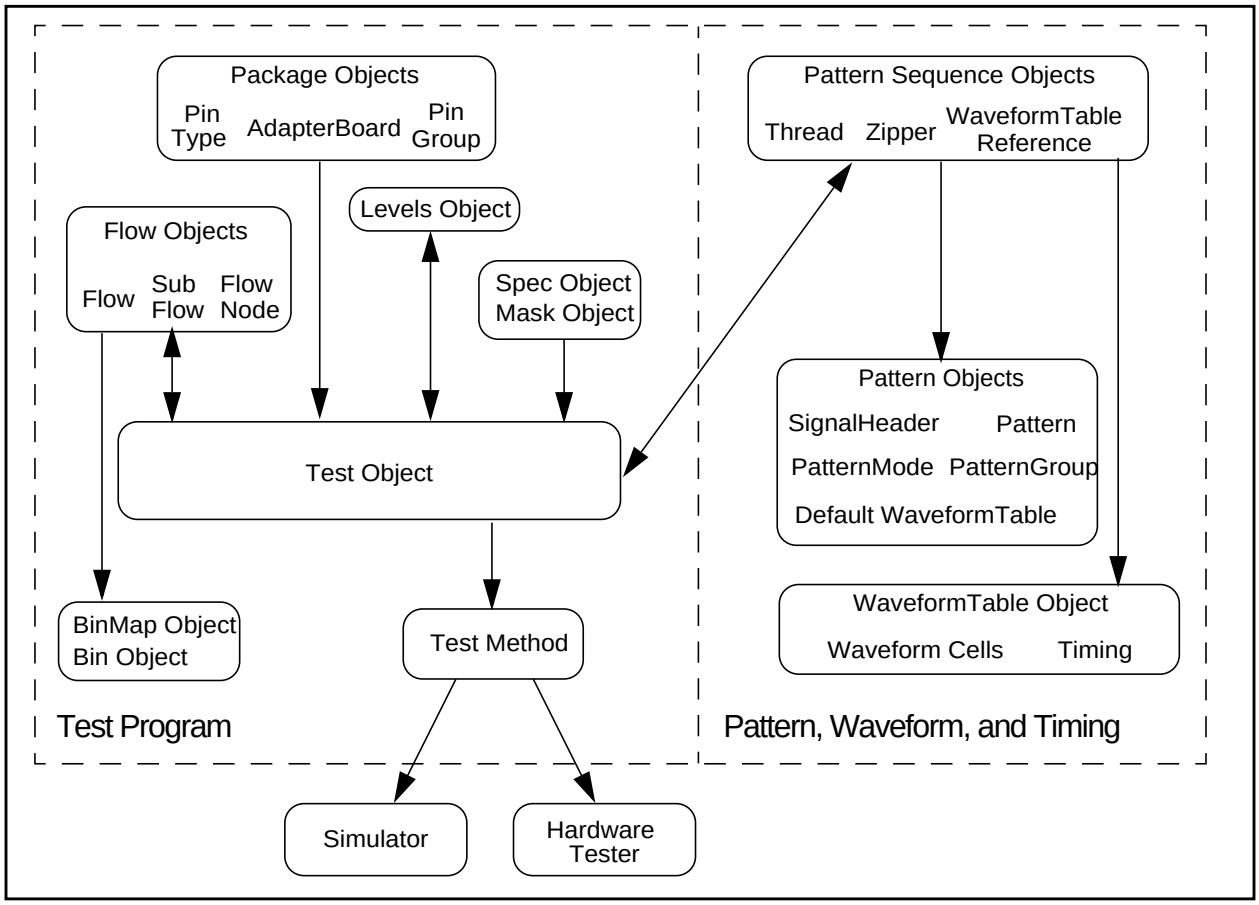


Figure 2.1: enVision Object-Oriented Architecture

enVision Test Program

A test program consists of top level objects that are instances of various object types; refer to Pattern Data Objects. When you run a test program, the tester hardware is controlled by the executing test methods associated with the objects in the database. Note the test methods (procedures) for executing each object are part of each object, not the test methods using them.

Top Level Objects

An enVision test program contains three types of top-level objects that contain other objects or pointers to other objects:

- `Adapterboard`—relates the tester resources to the DUT and the loadboard.
- `Test Flow`—controls the test program sequence.

- Spec—defines the parameters and expressions related to the device's specifications.

These objects interact with each other through their methods that encapsulate the object functions and its data. The objects types are defined by the enVision architecture; refer to Types of Pattern Data Objects.

Test Program Files

Although the management of most test program files is transparent to the user, you should know about the enVision file formats and how enVision manages them; refer to Managing Pattern Files.

Accessing enVision Database via the .eva File

The enVision database is not directly accessible; however, you can access and manipulate the underlying database by using the enVision GUI development tools. Also, you can edit the database by exporting it as an ASCII .eva file.

Pattern System

The pattern system is a key component of a test program. A separate pattern file describes certain aspects of the DUT behavior and maps this information to specific hardware resource that execute these patterns. This pattern information requires objects to integrate the patterns into the test program environment; refer to Types of Pattern Data Objects.

A pattern file contains only one `Pattern` object that defines a sequence of vector executions, whose data will be loaded into the tester hardware. A vector is the set of waveshapes associated with every pin, for one tester cycle.

After the pattern file is created, you load the test program, and use the GUI tools to create pattern sequences and to define the waveforms and timing. For tutorials on these tools, refer to the *VX250 Digital Tutorial* manual.

The pattern system is described in this chapter in the order patterns for a test program are developed. In addition to the descriptions of the pattern objects, pattern related topics of High Frequency operating

modes and Serial scan patterns are also described; however, the APG patterns designed for testing memory are described in documentation for the MTO option.

Test Pattern Structure

A functional test pattern file consists of four data structures:

- Pattern data—describes the device functions to the test system. It is the 0- and 1- bit definitions that are the truth table for testing a DUT.
- Waveform reference data—references each pattern data vector to a set of wave shapes, per pin. Also, refer to Zipper Table.
- Waveform definitions—comprise all wave shapes in a pattern execution. They describe the transitions that create a waveform. The waveform data has two components:

a. High-speed data is waveform data that can change from vector to vector during the execution of a functional pattern.

b. Low-speed data is waveform data that cannot change during a functional pattern and must be common to all waveform definitions for a specific tester pin resource. (This is true within one `PatternSequence` object. Low-speed attributes can change, if you change `PatternSequence` objects.)

- Timing values and relationships—execute the events of a waveform at a specific time relative to the start of the pattern or other waveforms.

Test Pattern File

Pattern files describe certain aspects of the DUT behavior and map this information to specific hardware resource that execute these patterns. Be aware of the different pattern file formats supported by different enVision software releases; refer to Pattern File Formats.

After the pattern file is created, you load the test program, and use the GUI tools to create pattern sequences and to define the waveforms and timing.

Pattern File Components

A sample pattern file contains the following basic information:

- Keyword `enVisionObject` at the beginning of the file identifies the file as an enVision object. Includes the software revision.
- Optional `comment` describes the `Pattern` object. Comments are within double quotes.
- One Pattern object is a group of individual pattern vectors. This object is identified by the starting keyword `Pattern`, followed by its name, and a brace signifying the beginning of the pattern set and another brace to signify the end of the file.

  Each `Pattern` object must include its variation type: type of hardware or software to execute the `Pattern` object. There are different types of data generators for supplying DUT pattern data. Specify this object with the keyword `Type`, followed by one of several types: `DPM`, `CPM`, or `Keep Alive`; refer to Types of Pattern Types (Variations).
- Default `SignalHeader` is specified with the keyword `Default SignalHeader`, followed by the header name specified as a separate object. An alias character in a waveform table of Waveform Tool specifies the waveform of each pin; refer to Signal Direction and Alias Symbols.
- One or more labels as symbolic address references throughout enVision, including one or more `Pattern` objects. A label is a valid enVision name preceded by the `$` character.
- `WaveformTable` refers to waveform formatting and timing; refer to Waveforms.
- Optional micro-instruction field; refer to CPM Micro-Instructions.

Pattern Syntax Example

```
enVisionObject:"bl8:R5.6";
Pattern Pat2 {
Type CPM;
Default SignalHeader SimpSigHeader1;
$Start2     * 10LH * TS1; "Inverter Mode"
            * 11HL * TS1; "Inverter Mode"
            * 10LH * TS1; "Inverter Mode"
            * 11HL * TS1; "Inverter Mode"
```

```
                * 000H * TS0; "Nand Gate Mode"
                * 001H * TS0; "Nand Gate Mode"
                * 010H * TS0; "Nand Gate Mode"
$End2           * 011L * TS0; "Nand Gate Mode"
}
```

Methods for Creating a Pattern File

You can use several methods to create a pattern file. It can be a single file or multiple files. By dividing a larger pattern file into smaller files, the `Pattern` objects are easier to manage and recombine in a pattern sequence. Use any method to create this file; however, you can decide on the most efficient method to create a file based on its size:

- Very small patterns—use PatternTool.
- Medium sized patterns—use a text editor and epc, the enVision stand-alone compiler.
- Very large patterns—use the data generator or translator of another test system or CAD system. For example, the LTX TVCBridge is a translator that converts patterns from an older LTX system software to enVision.

Since our example is a medium-sized pattern, it was created with a text editor. If you examine the example file, vector specific `WaveformTable` references and `SignalHeaders` are used; however, the default references could have been used.

You can use the enVision tools to create any or all of these components of a test pattern or write a complete test pattern by using a text editor.

Pattern File Formats

enVision compiles a ASCII pattern file with the extension .evo when a test program is loaded, producing a compressed, fast loading, binary version of the pattern file with the extension `.flex` or `.epf`:

- `epf (R10.8 and above)`

  enVision R10.8 and above introduces a compiled pattern format used with an external, stand-alone pattern compiler. You must enable enVision to save in this format by setting the environment variable `PATTERN_FORMAT` to `EPF` before starting enVision++. In this mode, enVision no longer manages

compilation: it just loads the pre-compiled patterns. To compile patterns into the .epf format, use epc, enVision pattern compiler, refer to Offline Pattern Compiler.

- flex (R10.8 and above)

  When an ASCII pattern file with the suffix .evo is loaded, enVision performs a partial compile on the pattern and stores the data into a file that is named pattern_name.flex (or .cpg).

NOTE Below R10.4, flex format was used.

Time Stamp of Pattern Files

When a test program is loaded, enVision examines the creation data and time of the .evo and .flex files. If the .evo file was modified after the .flex file was created, enVision++ recompiles the .flex file. Consequently, if you move the pattern file, be aware of the time stamp of the file to prevent unnecessary compiles. Also, refer to Managing Pattern Files.

Managing Pattern Files

- enVision creates a pattern_name.flex.LOCK file as a pattern is being compiled. It also creates a pattern_group.cpg.LOCK file as a group information is being updated.

  If a .LOCK file exists when the system is not compiling (monitor the file size), you should recompile the associated pattern group after their .LOCK, .cpg and .flex files are removed.

- If you move a pattern file from one pattern group to another, the orignating pattern group must be recompiled because it holds a reference to the pattern name. The same pattern name cannot be referenced in two pattern groups. The .cpg references are referenced to WaveformTable references in the ZipperTable.

You should not have to delete any pattern file; however, if either the .flex or .cpg is deleted or corrupted, delete all .flex and .cpg files associated with a pattern group. Once you reload the test program, enVision will recompile the files.

Pattern Data Objects

For the test engineer's convenience in developing, managing, maintaining, and reusing patterns, the enVision software divides the pattern set into multiple pattern files called *objects*. Not only are the patterns defined as objects, but the enVision waveform and timing subsystems are also data objects; refer to Types of Pattern Data Objects.

These subsystems are an integrated set of data objects that define the setup and execution of the high-speed functional pattern hardware in the tester. Figure 2.2 shows these data objects and their relationships. These data objects are collectively referred to as *patterns*. One of these objects, `Pattern`, tells enVision how the DUT operates. These objects are connected by the PatternSequence Object.

Types of Pattern Data Objects

- `SignalHeader` object—defines an ordered list of signal names used by enVision to process patterns. Signals are individual pin names, pin groups, or both. A pattern cannot be loaded without an existing referenced `SignalHeader` object.

- `PatternGroup` object—contains the common information for a set of patterns, such as where to store the pattern file and the list of `SignalHeader` objects supported by the pattern file.

- `WaveformTable` object—defines the waveshape and timing for each of the alias characters in a pattern. Figure 2.2 shows this object either connected to a pattern group or a pattern sequence:

a. If a `WaveformTable` object is associated with a pattern group, all data defined in the `WaveformTable` defines the format, timing, data source, among others, for the patterns in the group. These patterns are fully defined entities. The minimum requirement for this `WaveformTable` is that the data level (0/1) state of each alias must be defined. This data is the first level of data reduction during pattern compiling.

b. All other waveform data is not entered; thus, the pattern sequence must resolve the remaining waveform and timing information. `WaveformTables` may inherit from other waveform tables, so common waveform information can be defined in one table and used by other `WaveformTables`; refer to Inheritance.

- `TimingExpression` object—defines a constant or variable relationships for a `WaveformTable`: either a timing marker or period value of a waveform. It consists of constants, Spec variables from a `SpecTool` data object, or functional operators. For more information about SpecTool, refer to the *SpecTool* chapter in the *Tools* manual.

- `PinExpression` object—defines a set of pins. In `WaveformTables`, you specify alias characters for each cell with a pin expression; thus, the same alias characters can be used with different waveshapes on different pins. For more about defining pins by using Waveform Tool, refer to the *Waveform Tool* chapter in the *Tools* manual.

  Pin expressions are similar to `PinGroups` data objects except the pin expression is specified within the context of the `WaveformTable`. It is used only in that table. In contrast, the `PinGroup` data objects are used throughout the test program; however, LTX recommends you use the existing `PinGroups` object when possible, rather than generate new ones.

- `Adapterboard` object—defines the basic pin structure of the DUT. Pin name and `PinGroup` names referenced by the `Pattern` and `SignalHeader` must exist before you load the test program. For more information about defining the pin structure of a DUT, refer to the *PackageTool* chapter in the *Tools* manual.

- `PatternMap` object—specifies the pattern files to load, and the `PatternGroups` to load them into for all patterns in an enVision program. Different `PatternMaps` enable a single `Pattern` object to map different pattern files or `PatternGroups`, or both, or to implement pattern variations or revisions for device testing. Note this object replaces the functions performed by the `ExternalRef` statement of earlier enVision releases.

- `PatternMode` object—specifies the pattern mode, enabling users to easily modify patterns that execute at rates faster than the base rate of the tester. This object lets you combine *N* user vectors into one tester vector.

- External References—not a data object, but a reference to an external file containing other objects.Any number or mix of non-pattern objects may exist in an `ExternalRef` file, but they cannot be mixed with a ASCII pattern file. All objects in the external file are loaded, referenced or not. Once you have saved the test program, all objects are re-written.

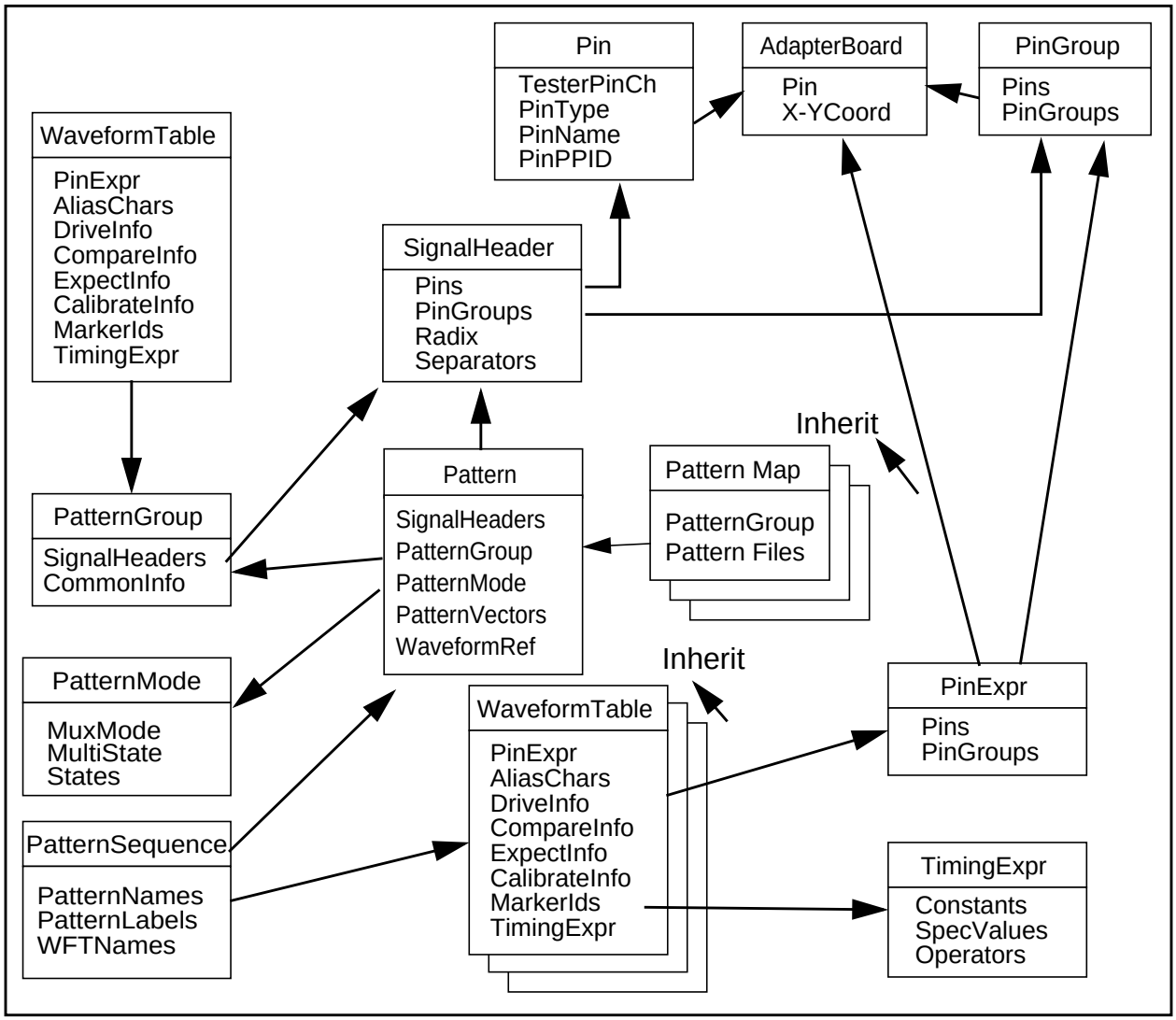


Figure 2.2: Pattern Data Objects

Pattern Object

A `Pattern` object defines the vectors applied to the DUT. Its components include pattern data objects and other information that tell enVision how the DUT operates; see Figure 2.3. It is managed differently from other enVision data objects because pattern files are usually very large, containing millions of vectors; refer to Managing Pattern Files.

Pattern Components

- Set of alias characters for each device signal—specify the signal direction: Input, Output, Bidirectional, or Don't Care.

- Optional SignalHheader—defines the order of the signals.
- Waveform Reference Table—defines the alias symbols used for the pins in the vectors in the Pattern object.
- Optional micro-instruction; refer to CPM Micro-Instructions, DPM Micro-Instructions, and APG Micro-Instructions.
- Optional labels
- Optional ExternalRef statements
- Optional comments

Creating a Pattern Object

To create a this object, you must create the following objects or use existing ones:

1. PatternGroup object specifies the pattern file and the SignalHeader. enVision creates a Common Pattern Group xxx.cpg file, which stores the alias symbols common to the group, GWIDs (Global Waveform Identifiers), SignalHeaders of the group, patterns identifications compiled in the group, and common waveform reference information (waveform compression). Use Pattern Tool to create the PatternGroup object.

2. SignalHeader object defines the pin order in a vector. It is defined globally for the whole pattern file or referenced vector- by-vector. It must exist before you can enter the data in your pattern.Use Pattern Tool to create it. Information is stored in the .eva file.

3. Default WaveformTable reference defines the alias representations of the data source, format and data. Every character referenced must have a complete set of events and timing information. For more complex applications, you can also store waveforms information, inheritance expressions, or error checking translations.

   To create a WaveformTable Reference object, use Waveform Tool; refer to Using WaveformTool to Develop the WaveformTable Object Waveform data and timing values are referred to by Wave Tool.

   To create the Waveform objects, use the Pattern Sequence Tool.

To connect all `WaveformTables` to a given `WaveformTable` Reference, use the Zipper Table in the PatternSequenceTool; see Figure 2.3.

4. Define the pattern type: DPM, CPM, Base, FDM, or KAP.

5. Define the optional micro-instructions, optional labels, optional `ExternalRef` statements, and optional comments.

PatternSequence Object

This object specifies the sequence pattern execution by determining which patterns to load and their loading sequence into the tester memory. A pattern requires at least one `PatternSequence` object. It also specifies the `WaveformTable` object, which determines the formats and timing during pattern execution. The result of the `PatternSequence` object are executable threads, callable by the methods; refer to Pattern Threads.

The `PatternSequence` object is referenced by a particular `Test` object in a test program flow.

PatternSequence Object Entities

The `PatternSequence` object consists of two entities:

- Pattern `Thread` objects.
- Timing associated with the waveform table (Zipper Table).

Pattern Threads

Threads are a collection of patterns executed sequentially and controlled in a test program; refer to Figure 2.4. A pattern sequence may have one or more threads of execution; refer to PatternSequence Elements. Their order of definition is the order of execution.

Thread Entities

A thread is composed of pattern label references and `Pattern` objects of the following types: Base Pattern Type, Control Pattern Type (CPM), Data Pattern Type (DPM), Keep Alive (KAP) Pattern Type, or Fast Data Mode Pattern Type (FDM).

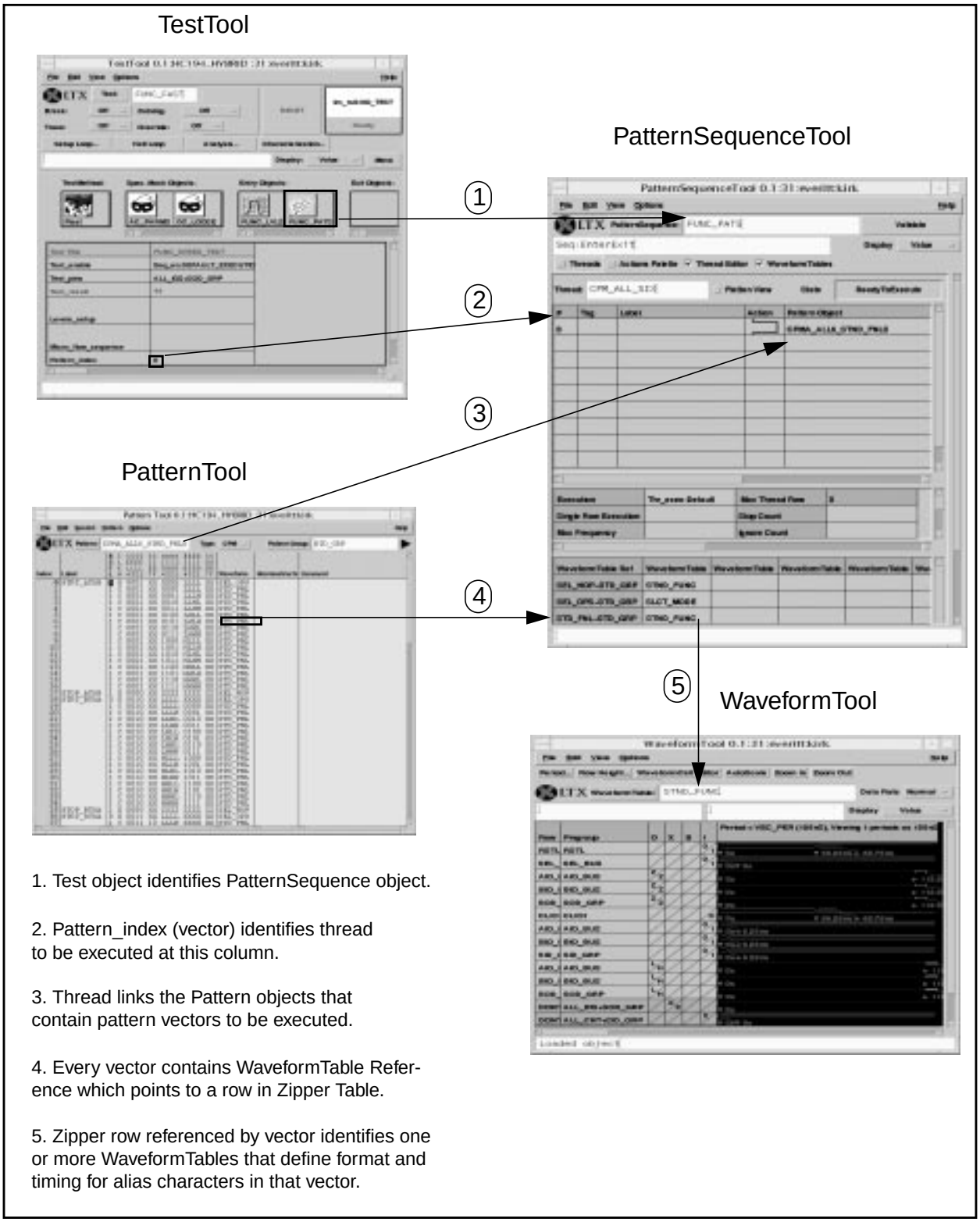


Figure 2.3: Creating Pattern, Waveform, and Timing Relationships

Thread Control Properties

You can define the control properties for the thread execution:

- If the thread is simple, specify the test start address and tell the tester to begin executing the thread.
- If the thread is complex, a control pattern type CPM pattern is generated in memory, which starts the desired execution sequence (also known as *auto-generated pattern*), such as:

a. portions of the patterns to ignore failures upon execution

b. specific points in patterns to trigger external events

c. additional stop and ignore fail modes

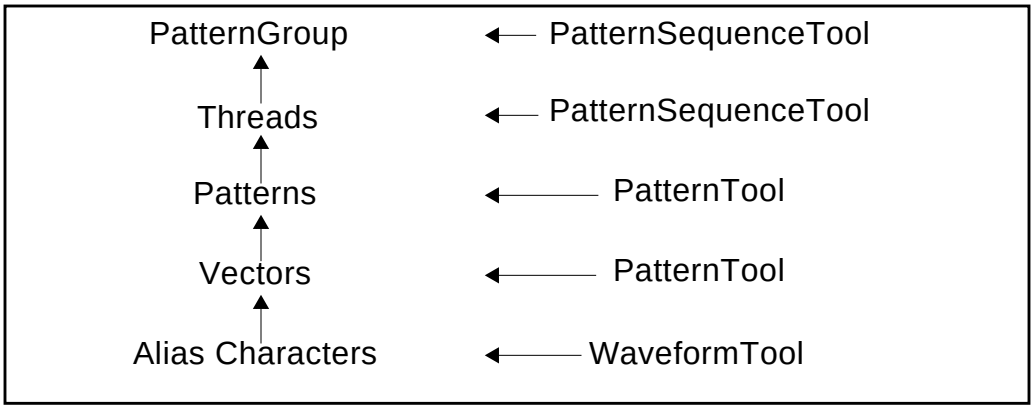


Figure 2.4: Hierarchy of Alias Characters, Vectors, Patterns, Threads and PatternGroup

Types of Threads

For debugging, characterization, and test development, you may have to run various tests that exercise the patterns sequentially on the tester, such as alternate start and stop locations in the patterns, skipping over failing patterns, selecting alternative timings, triggering external events upon some pattern execution condition, and others.

Thread actions are described in the *PatternSequence Tool* chapter in the *Tools* manual.

Zipper Table

The second major function of the PatternSequence object is the Zipper Table, which resolves the list of waveform references from each vector of all patterns in the pattern sequence to a WaveformTable object. Figure 2.2 shows this object may be directly associated with the Pattern object or connected via the PatternSequence object. The connection via the PatternSequence data object is the most flexible because you can reuse the pattern set in another sequence with a different set of format and timing relationships.

PatternSequence Elements

- Name of the object
- Another main component of a PatternSequence object is the Thread, specified by its name. It refers to a separate object, which may be reused across different pattern sequences. A Thread object may be composed of the following elements:

a. Row descriptor of the thread action expression, such as Enter, Exit, Trigger, and others.

b. Associated pattern as a Pattern object or pattern group pair, with an optional label.

- A Zipper Table is most likely to follow, where the description of the pattern specifies the WaveformTable References (Figure 2.3) relationships to the user WaveformTable defined in Waveforms. The zipper definition:

```
Row  {
   Pattern Group name, Waveform Reference name = {
      Waveform Table names
   }
}
```

Example: PatternSequence Object

```
PatternSequence DC_pats {
    Thread[0] = Simple_functional;
    Thread[1] = Bus_tristated;
    Thread[2] = Bus_low;
    Thread[3] = Bus_high;
    Zipper = Zipper {
        Row { Z86DC_pats, _0 = { DC } }
    }
    TransactionCount = 187;
}
```

```
Thread Simple_functional {
    Row {
        ThreadAction = Expr { String = "Seq:Enter"; }
        PatternLabel = Expr { String = "Z86DC.DC_START"; }
    }
    Row {
        ThreadAction = Expr { String = "Seq:PatTrigger"; }
        PatternLabel = Expr { String = "Z86DC.S6VOH"; }
    }
    Row {
        ThreadAction = Expr { String = "Seq:Exit"; }
        PatternLabel = Expr { String = "Z86DC.DC_STOP"; }
    }
}
Thread Bus_tristated {
    Row {
        ThreadAction = Expr { String = "Seq:Enter"; }
        PatternLabel = Expr { String = "Z86DC.DC_START"; }
    }
    Row {
        ThreadAction = Expr { String = "Seq:Exit"; }
        PatternLabel = Expr { String = "Z86DC.BUSLEAK"; }
    }
}
Thread Bus_low {
    Row {
        ThreadAction = Expr { String = "Seq:Enter"; }
        PatternLabel = Expr { String = "Z86DC.DC_START"; }
    }
    Row {
        ThreadAction = Expr { String = "Seq:Exit"; }
        PatternLabel = Expr { String = "Z86DC.BUSVOL"; }
    }
}
Thread Bus_high {
    Row {
        ThreadAction = Expr { String = "Seq:Enter"; }
        PatternLabel = Expr { String = "Z86DC.DC_START"; }
    }
    Row {
        ThreadAction = Expr { String = "Seq:Exit"; }
        PatternLabel = Expr { String = "Z86DC.BUSVOH"; }
    }
}
```

Example: Test Object PatternSequence

After defining the patterns, you specify the sequence of patterns used in the test objects. This sequence is referred to in the test methods as `Pattern_index`, and corresponds directly to the pattern sequence `Thread` object. To create a pattern thread, use PatternSequenceTool. The result of this operation is the following .eva code:

```
PatternSequence Pats {
    Thread[0] = Functional;
    Thread[1] = Abus_high;
    Thread[2] = Bbus_high;
    Zipper = Zipper {
        Row { LS245_pats, Std_timing = { Timing } }
        Row { LS245_pats, Tplh = { Tplh_tmg } }
        Row { LS245_pats, Tphl = { Tphl_tmg } }
        Row { LS245_pats, Tplz = { Tplz_tmg } }
        Row { LS245_pats, Tpzh = { Tpzh_tmg } }
        Row { LS245_pats, Tphz = { Tphz_tmg } }
        Row { LS245_pats, Tpzl = { Tpzl_tmg } }
    }
    TransactionCount = 0;
}
Thread Functional {
    Row {
        ThreadAction = Expr { String = "Seq:EnterExit"; }
        Pattern = Functional_pattern;
    }
}
Thread Abus_high {
    Row {
        ThreadAction = Expr { String = "Seq:Enter"; }
        PatternLabel = Expr { String = "Functional_pattern.Fct_st"; }
    }
    Row {
        ThreadAction = Expr { String = "Seq:EnterExit"; }
        PatternLabel = Expr { String = "Functional_pattern.Abus_Ios"; }
    }
}
Thread Bbus_high {
    Row {
        ThreadAction = Expr { String = "Seq:Enter"; }
        PatternLabel = Expr { String = "Functional_pattern.Fct_st"; }
    }
    Row {
        ThreadAction = Expr { String = "Seq:EnterExit"; }
        PatternLabel = Expr { String = "Functional_pattern.Bbus_Ios"; }
    }
}
```

SignalHeader Object

This object defines a set of waveshapes for each pin, with the following functions and properties:

- List of device pins—Each pattern vector refers to at least one `SignalHeader` representing all or a partial list of the device pins. It may be a default header or an explicit header; refer to SignalHeader Example.

- Relates alias characters to signals—`SignalHeader` is used by `Pattern` object to relate the alias characters of each vector to the device's signal names assigned in DeviceTool; SignalHeader with Defined Alias and SignalHeader with Inherit Alias.

- Global or local scope—This object is specified in either the pattern files for definitions local to the pattern file or in the main program file for global use by other pattern files.

SignalHeader Entities

- Header name, as referenced in the pattern file.

- List of signal names, as specified in DeviceTool, separated by white space or commas. Signal order must correspond to the vector alias order, left to right. An optional percent (%) character may precede pin name, which appears in PatternTool and the datalog output as a column separator so the data easier to read.

  The Pattern Loader does not require the correct input spacing, but the number of signal names in the `SignalHeader` must match the number of alias characters in each vector of the pattern.

  Only pins whose `PinType` is defined as `Type=Norm`; may be used (`Norm` refers to functional data pins rather than power supply or other special pins).

- List of signals may use `PinGroup` names so you can specify the radix of the entered data. Only one alias appears in the vector stream for the group, followed by its data in the specified radix (default is hexadecimal).

SignalHeader with Defined Alias

Example `Signals` statement with defined alias:

```
Signals { %AD_bus {Bus AD15 AD14 AD13 AD12 AD11 AD10
AD9 AD8 AD7 AD6 AD5 AD4 AD3 AD2 AD1 AD0; Radix 16;}
```

Alternatively, you can define Signals with a constant alias value for the entire pattern execution. With this method, enVision will not expect to find an alias character for the signal when the vector is processed, but it will always use the defined constant state. enVision does not support on-line changes to a constant pin.

SignalHeader with Inherit Alias

Example Signals statement with inherit alias:

```
Signals { %AD0 {Constant 1;}
```

Signals may inherit the exact aliases as another signal, without you have to re-specifying the aliases.

During vector processing, the alias character for the referenced signal will be used for the duplicated signal; however, enVision does not support on-line changes to a duplicate signal; refer to SignalHeader with Duplicate Signals.

SignalHeader with Duplicate Signals

Example Signals statement with duplicate signal:

```
Signals { %AD0 {Duplicate AD15;}
```

SignalHeader Example

Excerpt from .eva file with two examples of signal headers:

```
SignalHeader SH0 {
    Rate 1;
    Default Format Fixed 1;
    Signals {
        %AD14 AD13 AD12 AD11 AD10 AD9 AD8 AD7 AD6 %AD5 AD4 AD3 AD2
         AD1 AD0 NMI INTR Clk %RES Ready Test0 QS1 QS0 S0 S1 S2
         Lock RQ_GT1 %RQ_GT0 RD MN_MX BHE_S7 A19_S6 A18_S5 A17_S4
         A16_S3 AD15
    }
}

SignalHeader SH1 {
    Rate 1;
    Default Format Fixed 1;
    Signals {
        %NMI INTR Clk RES Ready Test0 MN_MX %QS1 QS0 S0 S1 S2 Lock
         RD %RQ_GT1 RQ_GT0 %BHE_S7 %A19_S6 A18_S5 A17_S4 A16_S3
```

```
        %AD15 AD14 AD13 AD12 AD11 AD10 AD9 AD8 AD7 AD6 AD5 AD4 AD3
         AD2 AD1 AD0
    }
}
```

NOTE The rate and default format in the example were not discussed because they are always the same and are ignored by enVision.

Creating a SignalHeader with PatternTool

Use PatternTool to create a signal header or add the `SignalHeader` information to your .eva file. LTX recommends you use PatternTool.

You must create a `SignalHheader` before entering the pattern data. Likewise, before a `SignalHheader` can be created, other objects must exist, such as an `AdapterBoard` object.

PatternMap Object

This object loads the pattern files.

NOTE If you are using `.flex`/.cpg pattern files, you must use the `ExternalRef` syntax to define a `PatternMap` object.

PatternMap Object Properties and Defaults in PatternTool

- Each row of this object maps a `Pattern` object name to the binary and source files of the pattern and to its pattern group; refer to PatternMap Example.

  The filename field defaults to the name of the `Pattern` object, which is usually the best choice for a pattern name, unless you require the pattern to be mapped to more than one single file.

- Fields in a pattern row can contain optional expressions rather than simple strings. If a field is blank, the default value is used. Most of these fields of a typical `PatternMap` are blank.

- By using expressions, pattern files can be selected at test time via operator variables or other expressions.

- Typical programs have a single PatternMap object with an entry for each pattern; however, multiple PatternMaps (Active PatternMap, inheritance, dynamic mapping using expressions are supported.

- The Remove field defaults to False. Also, refer to Preventing a Pattern From Loading.

Active PatternMap

While a program can have several PatternMaps, you must designate an *active* map, which may stand alone or optionally inherit data from other PatternMaps. The active one is specified through Operator Tool, through the PatternMap panel of the PatternTool, or in the .eva file via the PatternMap entry in the TestProg object.

Active PatternMap after Loading Pattern File

enVision re-evaluates the active PatternMap after it loads the .eva file, after the on-load flow, and before each test run. Evaluating a PatternMap can create and delete patterns or change the files from which they are read or both, or the pattern groups in which they are included. Interactive changes to a PatternMap object does not take effect until a test is run. For example, if you change the file in which a pattern is mapped in a PatternMap object, and then open the pattern in the PatternTool, you still see the old file.

Inheritance

Inheritance can be used to add revisions of patterns to sets of patterns or divide a PatternMap into sections. Entries in the PinMap take precedence over inherited entries. The format of the inheritance list in both the .eva file and in Pattern Tool is a space-separated list of PatternMap names:

- PatternTool: To inherit data from another PinMap, add the name of the map to the inheritance list by selecting Inherit From in the File menu in the PatternMap panel of PatternTool.

- .eva file: specify the InheritFrom keyword.

Inheritance Example. Inherited maps are processed by depth (deepest first) in their order in the inheritance list, as shown in a sample inheritance tree in Figure 2.5.

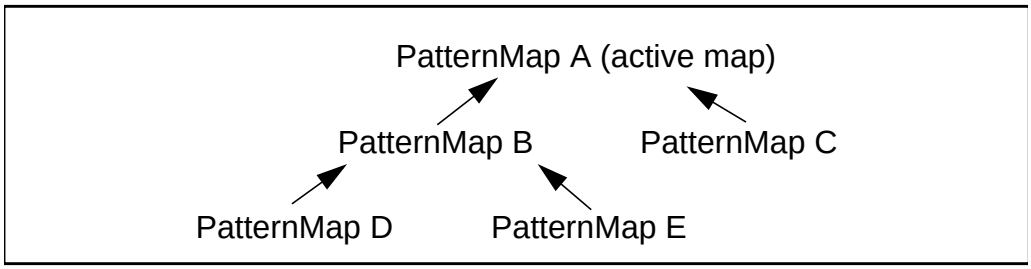


Figure 2.5: Sample Inheritance Tree

In this tree *PatternMap A* is the active pattern map, setting its inheritance list to *PatternMap B* and *PatternMap C* and setting *PatternMap B*'s inheritance list to *PatternMap D* and *PatternMap E*. If all maps contained an entry named `pat1`, the entry from *PatternMap A* is used. If *PatternMap A* did not have a `pat1` entry, the entry from *PatternMap B* is used. If neither *PatternMap A* nor *PatternMap* had a `pat1` entry, the `pat1` from *PatternMap D* is used.

Splitting a PinMap

To split up a map, use an empty active `PatternMap`, which inherits from a group of submaps. Be aware of filename conflicts, since maps occurring earlier in the inheritance list will override the later ones.

Using Multiple PatternMaps

You can use multiple `PatternMaps` as alternative maps, activated for certain part revisions or other program variations. In this case, a revised map inherits from the original, and includes only those patterns requiring changes.

Using Expressions in PatternMap Objects

You can configure `PatternMap` fields as the value of an enVision expression. For example, if a part had a number of variant ROM patterns, the filename field is specified:

```
strcat("RomPat_", RomVersion)
```

where **`RomVersion`** is an operator-entered variable.

Strings and Expressions. Most fields of the `PatternMap` support strings or expressions, with strings as the default. The fields support expressions: pattern filename, source path, binary path, `PatternGroup`, and remove field. The `PatternMap` default and

Pattern object name fields do not accept expressions; consequently, since strings are usually entered in these fields, both the ASCII pattern syntax and the PatternMap interface in the PatternTool expect strings and expressions to be qualified. To enter an expression in a PatternMap field, enclosed it with parenthesis in the PatternMap panel of the PatternTool. The tool examines entries for a parenthesis character to determine if the entered text is an expression. Entries without a parenthesis character are strings.

Pattern File Expressions. In the ASCII .eva pattern file, field values are either double quoted strings or standard expression format: Expr {...}. Characters must be enclosed with parenthesis because most expressions use character strings, and enVision does not support string operators. Any entry not enclosed with parenthesis characters is considered a string. In the ASCII .eva pattern file, field values are either double quoted strings or a standard expression format Expr {...}.

Preventing a Pattern From Loading

To prevent a pattern from loading, double click on the Remove field in the pattern row in the PatternMap panel of the PatternTool, or set the Remove field of the pattern in the map to True in the .eva file. Setting the PatternMap Remove field to True is the same as setting the thread action to *no action* for each thread referring to the pattern.

PatternMap Example

In the following example, pattern binaries are in a subdirectory of the program directory, pat_bins, and pattern sources are in pat_sources. The filename for pattern Z8600 depends on the expression variable, pat_rev, which may be an operator variable. The filename is similar to Z8600_2.2. This map is set as the active PatternMap from the Operator Tool or in the TestProg section of the .eva file:

```
PatternMap MyPatternMap {
   DefaultSourcePath = "./pat_sources";
   DefaultBinaryPath = "./pat_bins";
   DefaultPatternGroup = "Z8600_pats";
   Pattern Z8600 {Expr { String = "strcat('Z8600_',
pat_rev)"; }, , , , }
   Pattern Z86TMU {, , , , }
   Pattern Z86FREQ {, , , "Z86DC_pats", }
   Pattern Z86DC {, , , "Z86DC_pats", }
}
```

PatternMode Object

This object enables the pattern compiler, epc, to combine *n* user vectors into one tester vector. By using this object to combine vectors for either digital applications or mixed signal applications, the base tester rate may be increased under certain restrictions. Your test pattern files may have to be modified to support this object; refer to Modifying Existing Test Programs. Be aware that this object is optional: not specifying it in a pattern file will produce a normal CPM/DPM pattern after the pattern file is compiled.

PatternMode Concepts

The current enVision release introduces several pattern concepts. Understanding these concepts can help you implement the `PatternMode` object in a test pattern file:

- Full and partial vectors—A full vector refers to all pins in the `SignalHeader` (page 2-27), while a partial vector is a subset of these pins (page 2-28). A partial vector specifies more than one alias for a pin. The vectors for the even channels contain aliases for all pins used in the pattern, while the vectors for the odd channels contain only the aliases for the pins in the `MuxMode`.

- `VectorGroup`—is a enVision temporary object that a group of user vectors executed in one tester cycle. The number of vectors in the *VectorGroup* depends on the number of states used in a MultiState pattern, and whether the pattern is in the `MuxMode`:

a. Not in `MuxMode`—number of vectors in the group equals the number of MultiState states.

b. In `MuxMode`—number of vectors in the group equals the number of MultiState states times 2.

   For example, to compile a 3-state, MultiState, `MuxMmode` pattern, `VectorGroup` contains 6 vectors: 6 user vectors executed in 1 tester cycle.

Modifying Existing Test Programs

If you have created test patterns for digital applications running on LTX's family of test systems besides VX IV, you may have to modify your pattern files and re-compile them as `MultiState`, `MuxMode`, or `MuxMode MultiState` patterns; refer to Vector Syntax for MuxMode Patterns, Digital Applications. However, no additional changes to the

WaveformTable objects are required. For example, if your test pattern requires 5-ns periods, the WaveformTable should be programmed with this value, not 10 ns with two data states per cell.

Also, you can translate any of these types of patterns into a DVM pattern, which doubles the frequency again.

PatternMode Restrictions

- Only the first vector in VectorGroup supports (1) micro-instructions, which are applied to the entire group of vectors and (2) labels associated with the first vector.
- If the number of user vectors is not a multiple of the number of states, the compiler will pad the end of the pattern with copies of the last user vector.
- Only CPM and DPM patterns are supported; scan patterns and APG patterns are not supported.
- MuxMode and MultiState patterns use only one SignalHeader.
- All patterns in the sequence must be compiled with the same PatternMode. For instance, if pattern **Pat1** uses PatternMode MuxMode, all patterns in the pattern sequences containing **Pat1** must also be compiled with the PatternMode MuxMode statement.

  This limitation is more restrictive than the SVM/DVM pattern restriction, where all patterns must be run in the DVM mode based on the MaxFrequency of the thread (still applies to the MuxMode and MultiState patterns). Consequently, you cannot mix normal SVM patterns with SVM patterns that use the PatternMode statement in the same thread or PatternSequence.

PatternMode, Digital Applications

This object has two modifiers for digital applications: MuxMode and MultiState (X), which can be combined or used independently, depending on the digital application; refer to MuxMode Syntax, Digital Applications:

- MuxMode—doubles the base tester rate of the DUT without any timing limitations; also, refer to Vector Syntax for MuxMode Patterns, Digital Applications.

- MultiState—increase the base tester rate of the compiled file with some timing limitations without using any extra pins.
- MuxMode MultiState–uses both full MuxMode and MultiState to increase the base tester rate up to 8 times with some waveform restrictions.

MuxMode Syntax, Digital Applications

```
➥ PatternMode [MuxMode(<pin_group_name>) |
  [X <number of states>];
```

where:

- MuxMode—instructs the compiler to create a MuxMode pattern for the pins in the specified pin_group_name. This option requires all pins in the PinGroup to be assigned to the even tester channels. The compiler will create the odd tester channels. enVision has this restriction to ensure the odd tester channels are read only.
- X—specifies the number of states (or user-defined vectors) to combine into one tester vector: 2, 3, or 4. The more states used, the fewer timesets and types of waveforms can be created. For digital applications, all pins use the same number of states.

Vector Syntax for MuxMode Patterns, Digital Applications

In this enVision release, the pattern vector data supports more than one alias on a pin. It is specified with a partial vector. Only one waveform reference per VectorGroup is allowed. Consequently, if a pattern uses MuxMode on a subset of the pattern pins, you must modify the pattern source file; refer to Example: Pattern Source for Partial MuxMode. However, if all pins in a pattern use MuxMode, you do not have to modify the pattern source; refer to Example: Pattern Source for Full MuxMode (all pins).

Example: Pattern Source for Full MuxMode (all pins).

```
SignalHeader SH0 { p1 p2 p3 p4 %b7 b6 b5 b4 b3 b2 b1
b0 %p5 p6 %p7 p8 }
PinGroup AD_mux_pins { Group = Expr { String =
“p1+p2+p3+p4+b7+b6+b5+b4+b3+b2+b1+b0+p5+p6+p7+p8”}}
Pattern FullMuxPat {
Type Dpm;
Default SignalHeader SH0;
PatternMode MuxMode(mux_pins);
```

```
* HL01 11010011 01 HL *  wf1
* LL00 00011101 01 HL *  wf2
* HL11 00110110 01 LL *  wf1
* LL01 10010100 11 HL *  wf3
* HH00 10100001 00 HL *  wf2
* HL01 00010010 01 LL *  wf1
* HL01 11010011 01 HL *  wf3
* HL01 11010011 01 HL *  wf2
}
```

Example: Pattern Source for Partial MuxMode.

```
SignalHeader SH0 { p1 p2 p3 p4 %b7 b6 b5 b4 b3 b2 b1
b0 %p5 p6 %p7 p8 }
PinGroup AD_mux_pins { Group = Expr { String =
“b7+b6+b5+b4+b3+b2+b1+b0”}}
Pattern PartialMuxPat {
Type Dpm;
Default SignalHeader SH0;
PatternMode MuxMode(mux_pins);
* HL01 11010011 01 HL *  wf1
*      10101010       *  wf2
* LL00 00011101 01 HL *  wf2
*      11101010       *  wf2
* HL11 00110110 01 LL *  wf1
*      00000000       *  wf1
* LL01 10010100 11 HL *  wf3
*      10111110       *  wf2
* HH00 10100001 00 HL *  wf2
*      10000010       *  wf3
}
```

MultiStateMap Object, Mixed Signal Applications

Patterns for mixed signal applications have a different requirements than those patterns for digital applications. Compared with patterns for digital devices, where all pins execute at the same rate, patterns for mixed signal devices must define multi-state vectors pin by pin because some pins must execute at a higher rate than others, and some of these faster pins execute at different rates. Also, patterns for mixed signal applications may have extra data on each vector for the pins in the MultiState mode.

In mixed signal applications, you have the option to select the pins for the MultiState mode by using the MultiStateMap object. With this object, you specify the list of pins to be compiled and executed in the MultiState mode, and the number of states for each of those pins; refer to MultiStateMap Restrictions.

Also, when a mixed signal application requires MultiState capability on only some pins, use the PerPinMultiState object.

MultiStateMap Syntax, Mixed Signal Applications

```
➥ MultiStateMap <name> {
  (pin_name | PinGroup_name):<number_of_states>
  (pin_name | PinGroup_name):<number_of_states>
  .
  .
  .
  (pin_name | PinGroup_name):<number_of_states>
  }
```

MultiStateMap contains a list of pin names or PinGroup names separated by a space followed by a colon (:) and the number of states for a pin or PinGroup. You must specify in this object only those pins whose number of states is greater than 1. Also, refer to MultiStateMap Example.

MultiStateMap Restrictions

All patterns in the pattern sequence must use the same MultiStateMap object so the datalogger can determine any failed vectors, and so all pins can be properly set up for the pattern compiler.

MultiStateMap Example

In the example, the MultiStateMap object contains a 2-state pin named **twox_pin**, a 3-state pin named **threex_pin**, and a 2-state PinGroup named **group2x**:

```
MultiStateMap MixedSignalStates
{
   twox_pin:2
   threex_pin:3,
   group2x:2
}
```

PerPinMultiStateMap Syntax

When compiling a test pattern for mixed signal applications, use the following PatternMode directive to tell the compiler the name of the MultiStateMap object:

```
➥ PatternMode [MuxMode(<PinGroup_name>) ] |
  [PerPinMultiState <multistate_map>];
```

- MuxMode instructs the compiler to create a MuxMode pattern. The MuxMode pins to compile are defined by the specified PinGroup name. To compile pins in the MuxMode, all specified pins in the specified PinGroup must be even tester channels. The compiler and loader will create the odd tester channels.

- PerPinMultiState <**multistate_map**> is the name of the MultiStateMap object that specifies the number of states on a per-pin basis. Use this option for mixed signal applications requiring MultiState capability only on some pins.

Also, refer to PerPinMultiState Example, Mixed Signal Applications.

PerPinMultiState Example, Mixed Signal Applications

Assume a sample pattern has 4 pins, **p1**, **p2**, **p3** and **p4**; **p1** is MultiState with 2 states, **p3** is MultiState with 3 states:

```
SignalHeader SH1 {p1 p2 p3 p4}
MultiStateMap MixSigStates {p1:2 p3:3}

Pattern mixedup {
Type CPM2;
DefaultSignalHeader SH1;
PatternMode PerPinMultiState MixSigStates;

* HL01 * wft1
* L 1  *
*   H  *
* LL11 * wft1
* L 0  *
*   H  *
* HL10 * wft1
* H 0  *
*   H  *
* LL11 * wft1
* L 0  *
*   H  *
}
```

External References

The ExternalRef statement loads external pattern files or external non-pattern objects; refer to Supported Files Types and Loading Non-Pattern Objects. In either case, an .eva pattern file is accessed, and all defined objects are loaded; refer to Sample External References. It is like an include file with the following differences:

a. ExternalRef file must fully define the objects in it.

b. Pattern files are no longer specified by ExternalRef statements. Even though the ExternalRef syntax is supported for backward compatibility, enVision converts the pattern file ExternalRefs to the newer PatternMap syntax when you save the the test program.

Supported Files Types

An ExternalRef statement appears only in an .eva pattern file. The following external pattern files can be referenced:

- .evo—ASCII object files
- .flex—Binary pattern cache files.
- .epf—Binary pattern cache files (legacy).

  LTX does not recommend this legacy pattern format because a pattern file in this format is converted to the PatternMap syntax when you save the pattern file.

- .cpg—Binary PatternGroup files associated with the .flex pattern files.

Loading Non-Pattern Objects

If the Type=Pattern statement is not included in the .eva pattern file, only non-Pattern objects are loaded into the .evo file. When the test program is saved, this file is rewritten with all objects it originally contained; refer to Creating External References.

Elements of External Reference

- Type, of Pattern object, if a pattern file, otherwise it is omitted.
- File to load

- `Path`, location of source file in the file system
- `CachePath`, location of cache file in the file system
- `Cache`, pattern filename
- `ObjectInfo`, name of `PatternGroup` object that external pattern file is associated with

Sample External References

The following sample excerpt from an .eva file shows two external references; the first refers to a pattern file; the second refers to a regular object file:

```
ExternalRef {
   Type = Pattern;
   File = "pat1.evo";
   Path = "patterns";
   CachePath = "patterns";
   Cache = pat1;
   ObjectInfo = "Super";
}

ExternalRef {
   File = "Super.wft";
   Path = "waves/";
}
```

Creating External References

Create an external reference by using the Object Manager Tool or by entering the information in the .eva file. LTX recommends directly entering the data in the .eva file, anywhere prior to the last object, the `TestProg`. Refer to the following sample .eva code:

```
ExternalRef {
   Type = Pattern;
   File = "Functional_pattern.evo";
   CachePath="$ENVISION/$TARGET_NAME/$TESTER/$ARCH/customer/help_doc/";
   Cache = Functional_pattern;
   ObjectInfo = "LS245_pats";
}
```

PatternGroup Object

A test pattern requires at least one PatternGroup object, which collects the related pattern data in a single binary .cpg file. This object contains common information of all patterns referencing the group. The object is referenced in the .eva file like an external reference. If the .cpg file already exists, it is loaded; otherwise, one is created.

PatternGroup Elements

- CachePath, location of the cache file in the file system.
- One or more SignalHeader objects. At least one SignalHeader is defined so the PatternGroup can be sized to the number of signals used by the largest pattern vector.

Sample PatternGroup Object

Excerpt from .eva file showing a sample PatternGroup object:

```
PatternGroup MIXpats {
   CachePath "./patterns";
   SignalHeader SH0;
}
```

WaveformTable Object for PatternGroup Object

A test pattern requires a default WaveformTable object for a PatternGroup. It specifies the default definitions for all time set names within the PatternGroup.

This object must exist before you compile the patterns for the first time; enVision does not create it. enVision will stop compiling and warn you if a pin or character definition is missing.

Creating a WaveformTable Object

Using Waveform Tool to Define a PinGroup.

LTX recommends you use Waveform Tool to create this object. You must create at least one PatternGroup WaveformTable to support the data interpretations of the pattern file.

1. Create a default PatternGroup WaveformTable by using Waveform Tool:

➥ `Pattern_group_name pattern_group_name`

The `PatternGroup` name appears twice, separated by a single space. You have no control over this name:

`WaveformTable` name, as `Pattern_group_name pattern_group_name`

Optional default `WaveformTable`, referenced by vectors not deliberately specified. Specify with keyword `Default WaveformTable` followed by the waveform reference name specified as a separate object.

2. In the rows of data fields of this object, specify all pins in your test patterns. All pins in this object are (1) the predefined `PinGroups` from the `AdapterBoard` or (2) the user-specified `PinGroup` with an alias pair of the same data defined across all pins of the `SignalHeaders` of the `PatternGroup`.

Each row of cells specifies the defined `PinGroup` in quotes, the alias characters, and an optional cell name. This field is the `Data` statement of the object. The data alias definition is a combination of integers from 0 to 7 describing the binary pattern data (**`0`**/**`1`**), signal direction (**`In`**/**`Out`**), and the Mask of the comparators (**`Care`**/**`Don't care`**).

Using Text Editor to Define a PinGroup for WaveformTable.

Enter the object data directly into the .eva file, anywhere prior to the last object `TestProg` in the following example.

All other fields of the waveform table can be undefined. Although you can define completed waveforms in this table, LTX does not recommend this approach.

The following excerpt from .eva pattern file shows one example of a default `PatternGroup WaveformTable` object:

```
WaveformTable Z86DC_pats Z86DC_pats {
    Cell "Z86DC_pats.Pins" T _T_ {
        Data 0;
    }
    Cell "Z86DC_pats.Pins" L/H _L_H {
        Data 0/1;
    }
    Cell "Z86DC_pats.Pins" 2/3 _2_3 {
        Data 4/5;
    }
    Cell "Z86DC_pats.Pins" P/Q _P_Q {
```

```
        Data 0/1;
    }
    Cell "Z86DC_pats.Pins" 6/7 _6_7 {
        Data 6/7;
    }
    Cell "Z86DC_pats.Pins" c/C _c_C {
        Data 4/5;
    }
    Cell "Z86DC_pats.Pins" 0/1 _0_1 {
        Data 6/7;
    }
}
```

Creating a Default PatternGroup for WaveformTable. The following example creates the default `PatternGroup` by entering the information in the .eva file. Even though LTX recommends using WaveformTool, you can enter this object data directly into the .eva file, anywhere prior to the last object `TestProg`:

```
WaveformTable LS245_pats LS245_pats {
    Cell “LS245_pats.Pins” 0/1 Drive_data {
        Data 6/7;
    }
    Cell “LS245_pats.Pins” L/H Compare_data {
        Data 0/1;
    }
    Cell “LS245_pats.Pins” z/Z Tristate_data {
        Data 0/1;
    }
}
```

Types of Pattern Types (Variations)

enVision supports several pattern types or pattern variations. The required pattern type depends on the application:

- Base Pattern Type
- Control Pattern Type (CPM)
- Data Pattern Type (DPM)
- Keep Alive (KAP) Pattern Type
- Fast Data Mode Pattern Type (FDM)

Base Pattern Type

This pattern variation is a user-specified pattern vector executed by an auto-generated control pattern type (CPM). This type enables patterns specified by the PatternSequence Thread object to be concatenated.

In a pattern sequence, you specify any order of execution of Pattern objects listed in the Thread objects. You can specify CPM, DPM, Keep Alive, or other types derived from these basic types. Multiple Base patterns may be used within a pattern thread; consequently, different Base vectors can be at different points in the pattern execution.

Example: Base Type Pattern

```
enVisionObject:"bl8:R5.6"; /* 5.5 */
Pattern Base_pattern.Base_ {Type Base;
* 0 0 00000000 ........* Std_timing DUT_pins;
}
```

Example: Thread Object With Base Pattern

The following example uses a Base pattern in a PatternSequence object. Recall that a Base pattern requires two or more DPM patterns to be run in different sequences.

```
Thread th1 {
    Row {
        ThreadAction = Expr { String = "Seq:EnterExit"; }
        Pattern = Pat_1    ;
    }

    Row {
        ThreadAction = Expr { String = "Seq:EnterExit"; }
        Pattern = Pat_2    ;
    }

    Row {
        ThreadAction = Expr { String = "Seq:SetRef"; }
        Pattern = Base_pattern.Base_    ;
    }
}
```

enVision examines this Base pattern and duplicates the vectors in CPM to create the microcode for loading and running the DPM patterns in the current thread. For each DPM pattern, the code Seq:EnterExit' Thread Action triggers enVision to create the auto-generated CPM pattern. The ThreadAction placed on the Base pattern is Seq:SetRef.

Control Pattern Type (CPM)

The CPM programs a pattern memory of the same name, like a truth table, by specifying the device state (high or low), and by referencing the hardware timing and waveform generators. This pattern type programs each vector in the pattern file with a unique memory address in the CPM memory and executes it as a single pattern vector.

Vectors may contain optional micro-instructions, which aid in generating algorithmic patterns, such as loops and subroutines. They significantly extend the limited capability of the CPM.

Because of the nature of the CPM, these patterns are fundamental to the execution of any types of patterns.

CPM Vector Hardware

- CPM counter is 16-bits wide: maximum count is 65536.
- It is 4K deep; refer to CPM Overlays. Because the memory is now embedded into the PATI chip in the Test Head, the CPM memory is limited to 4K. Even though the embedded memory is limited to 4K, the hardware threading and CPM reloading features should reduce the need for a larger CPM memory because these two give the user a virtual memory for CPM patterns; refer to Hardware-Based CPM / DPM Threading, CPM Overlays, and Sequence Table.

CPM Applications

The CPM is typically used for small vector sequences, with special micro-instruction requirements like match mode for synchronizing the tester to the DUT, or to control the execution and sequencing of data patterns.

Creating CPM Patterns

These CPM patterns may be explicitly developed by the user or they may be auto-generated by enVision.

Aliases

The number of aliases per vector within a CPM type pattern is associated with the `SignalHeader`. The alias must be one of the characters listed in the `Pattern` object syntax; refer to SignalHeader with Defined Alias and SignalHeader with Inherit Alias. It can be any

character in this list because these characters have no meaning until they are resolved by a waveform table. Once they are defined by the default `PatternGroup` `WaveformTable`, the pattern can be compiled.

Auto-Generated CPM Pattern Required for DPM Pattern

For most applications, LTX recommends letting enVision create the CPM vectors to execute the specified DPM patterns in the provided order. This is known as an *auto-generated CPM pattern*, which is executed as part of your patter: some vector executions that may not be desirable in your application are inserted. Also, refer to Sequence Table.

enVision auto-generated CPM patterns require you to specify a `Base` type vector for all pins specified in the pattern group executed by the auto-generated pattern.

This pattern type must contain a single vector, which is applied to all vectors executed as part of controlling the DPM patterns. If this single vector does not meet your device's requirements, you must write a custom CPM pattern.

CPM Overlays

Users of LTX VX 500 tester may recall that CPM pattern reloading is supported by using the workstation memory to store the CPM patterns. The enVision software now supports CPM pattern reloading, which stores the complete CPM patterns in DPM. This feature increases the effective size of the CPM and reloads the CPM pattern very quickly. The operation and usage of this feature is transparent to the user.

enVision reloads the CPM pattern in the available DPM between threads of execution, if required. From the point of view of a test program, this feature is a virtual memory for CPM patterns. The CPM pattern virtual memory is limited only by the available DPM.

If more than 4K of CPM is required by a test program, the pattern loader software loads the CPM patterns into this memory (DPM) and creates a map to track the patterns used in the CPM. If a thread requires a CPM pattern not loaded in the CPM, the pattern is reloaded from the CPM virtual memory (DPM) before the thread is executed.

Thread Checking

A single-thread execution is limited to 4K, the maximum amount of CPM memory. If a thread requires more the 4K of CPM, an error message appears, stating the name of the thread, amount of CPM memory required by that thread, and names and sizes of all CPM patterns in that thread. With this information, you can determine the cause of the problem and restructure the thread so it will load. Also, refer to CPM Overlays.

Hardware-Based CPM/DPM Threading

CPM / DPM threading, which is transparent to the user, can thread together the DPM and CPM patterns without using the CPM. From the user's point of view, threads work exactly as they did in previous LTX testers. The DUT will not see any gaps or dead cycles between the last vector of one pattern and the first vector of the next pattern.

In previous enVision software, the auto-generation pattern software created CPM microcode to join the various patterns in the loaded threads. This technique used CPM memory and required extra cycles to be inserted into the thread execution. In contrast, the VX 4 hardware uses a Sequence Table rather than creating CPM microcode to join the patterns in a thread when loading and executing a pattern.

From the user's perspective, the thread functions have not changed.

Sequence Table

The VX IV hardware implements the Sequence Table, a list of patterns that will be sequentially executed as if they were a single pattern. Each entry in this table contains a pattern start location, stop location and pattern type. The types of patterns that can be joined into a seamless pattern are DPM, CPM and KeepAlive (KAP) patterns. The list, which is stored in DPM, and thus, is limited only by the available DPM memory space. Its operation is transparent to the user. It cannot be accessed or modified by the user.

Note that certain usages, like ignore fails and enable fail actions in the thread, still require the auto-generation software to create the CPM microcode rather than using the Sequence Table.

CPM Micro-Instructions

CPM micro-instructions are a set of instructions that generate algorithmic patterns:

- CJMP
- CJSR
- CJSRI
- COND
- CRET
- DC1, DC2, DC3 and DC4
- FLAG
- JMP
- CJMP
- JSR and CJSR
- JSRI and CJSRI
- LC1, LC2, LC3, and LC4
- LDA
- MODE
- DNLD
- NOP
- RET and CRET
- RPT
- RPTP
- SDP
- SEQF (CPM branching)
- STOP and STOPF
- SWCCPM and SWCDPM

CJMP

➥ CJMP

Conditionally jumps to the specified label by examining the result of the condition flag set up by the COND and by branching to the specified label address if the condition is True; otherwise, it continues with the next micro-instruction. For details and examples, refer to JMP.

CJSR

➥ CJSR

Conditionally jumps to the subroutine, starting at the specified label by examining the result of the condition flag set up by the COND and by calling the subroutine at the specified label address if the condition is True; otherwise, it continues with the next micro-instruction. For details and examples, refer to JSR and CJSR.

CJSRI

➥ CJSRI

Conditionally jumps to the subroutine starting at the address defined by the Indirect register by examining the result of the condition flag set up by the COND and by calling the subroutine at the address stored in the Index register if the condition is True; otherwise, it continues with the next micro-instruction. For JSRI details and examples, see JSRI and CJSRI.

COND

➥ COND *condition*

> *condition*—specifies one of the active branching conditions listed in Table 2.1.

Specifies an active condition to be tested by future conditional instructions, such as CJMP and CJSR. The CONT, INTF, SEQF, APGF, Fl, F2, and F3 conditions are controlled through both the methods and the pattern file. When the True condition is set, all instructions become unconditional.

Table 2.1: Conditions for CPM COND Micro-Instruction

Condition	Description
CONT	Condition is True if CONT flag is ON. Flag is normally set ON by FLAG (CONT ON) micro-instruction and turned OFF by test_pattern_continue ETIC in a method. CONT is handshake for continuous mode looping, see COND examples.
FAIL	Condition is True if the vector test result is FAIL.
F1	Condition is True if Fl is True. Fl is a general purpose flag set and reset by the methods and the FLAG micro-instruction.
F2	Condition is True if F2 is True. F2 is a general purpose flag set and reset by the methods and the FLAG micro-instruction.
F3	Condition is True if F3 is True. F3 is a general purpose flag set and reset by the methods and the FLAG micro-instruction.
NZC1, NZC2, NZC3, NZC4	Condition is True if Loop Counter 1 to 4 (LCx) is non-zero.
SEQF	Condition is True if sequential match mode operation is True. Initially FLAG micro-instruction resets SEQF false and continues until any failing test vector, which sets condition True.Known as match mode; refer to FLAG micro-instruction.
TRUE	Condition is always True.

The following example shows a simple vector sequence loop using counter LC2 and a conditional jump. The rows of periods enclosed by asterisks in the examples represent functional vectors:

```
*...*; <COND NZC2> "Setup for future COND test of C2"
*...*; <LC2 30>    "Load counter 2 with value of 30"
$LI *...*;         "Address to jump to"
*...*;
*...*;
*...*;
*...*; <CJMP Ll,DC2> "Test current condition (C2) and if TRUE"
*...*; "jump to label Ll. Also decrement C2 at end of cycle"
```

The following example controls the test conditions by checking the status of a continuous loop. This mode leaves the DUT executing in a known condition during DC testing. To terminate this loop, use the Continue_patterns (Microflow method):

```
*...*; <COND CONT> "Set condition to the CONT flag"
*...*; <FLAG (CONT ON)> "Set the CONTinuous mode"
*...*; <CJMP .>  "Repeat until stopped from method"
*...*;            "Now, continue execution"
```

CRET

➥ CRET

Conditionally returns from a subroutine by examining the result of the condition flag set up by COND and by returning from a subroutine to the address following the call to CJSR, JSR, CJSRI, or JSRI if the condition is True; otherwise, it continues with the very next micro-instruction. For details and examples, see page 2-50.

DC1, DC2, DC3 and DC4

➥ DC1 DC2 DC3 DC4

Decrement the specified loop counter by 1. Normally, the counter is decremented at the end of the cycle. When a counter is decremented to zero, further decrementing has no affect: counter value remains zero and its condition stays False. These Type-2 instructions may be used with other Type-2 or Type-1 instructions. Typically, this instruction is used with a CJMP loop.

Restrictions:

- DC1 is not effective on the first cycle following a RET because counter DC1 is being restored (popped) from the stack; however, DC2, DC3, and DC4 are effective on the first cycle following a RET.
- DC1 trough DC4 may not be used with the Loop Counters LC1 through LC4 micro-instructions.

An example is listed under COND.

FLAG

➥ FLAG (*flag state, ..., flag state*)

flag—specifies the name of the flag turned on or off by the micro-instruction. More than one flag can be set or reset in a single FLAG micro-instruction by using a comma to separate the next flag from the previous state. For a list of flags, refer to Table 2.2.

state—enables (on) or disables (off) the specified flag. State of a flag may be a condition tested by all conditional instructions; refer to COND condition. Flags may also be set on or off and monitored by the methods. For example, a pattern may loop by testing a FLAG condition until the test program clears the flag, which is typical in a continuous mode of operation.

Sets or resets the specified flags on or off. One of several general purpose flags for communicating among the methods and CPM. Table 2.2 lists the set of CPM flags.

Table 2.2: CPM Flags

Flag	Description
CONT	Sets or resets the continuous mode. While this flag is set, the pattern and the test program can execute concurrently, such as during DC measurements while the DUT is executing a pattern sequence. This flag can also be turned OFF by the `test_pattern_continue` ETIC in a method. Note: The CPM, DPM, and timing memories cannot be loaded while patterns are running. At this time, all high-speed memories for pattern storage and timing storage are locked off the system bus. With this flag set, only low-speed registers can be written and read.
F1, F2, F3	General purpose flags: available for any user application.

JMP

➥ `JMP` *`location`*

`location`—label representing a CPM address location where execution continues.

Unconditionally jumps to the address specified by the label.

CJMP

➥ `CJMP` *`location`*

`location`—label representing a CPM address location where execution continues if the pattern branches.

Conditionally jumps to the specified label by modifying the address only if a condition is `True`; otherwise, it continues with the very next micro-instruction; for example, refer to JMP.

The following is an example of an unconditional jump:

```
        *...*; <JSR S1> "Pattern set #1"
        *...*; <JSR S1> "Subroutine S1 used every time"
        *...*; <JSR S1>
        *...*; <JSR S1>
        *...*; <JSR S1>
        *...*; <JSR S1>
        *...*; <JMP M2> "Uncond. jump to pattern set 2"
    $S1 *...*;          "Minimum two cycle subroutine"
        *...*; <RET>    "Exit subroutine S1"
    $M2 *...*;
        *...*;
        *...*;
        *...*; <STOP>
```

JSR and CJSR

➥ JSR *location*

➥ CJSR *location*

location—label representing a CPM address location where execution continues, if the pattern branches.

Unconditionally (JSR) or conditionally (CJSR) jumps to subroutine starting at the address specified by the specified label. CJSR is a conditional jump on a True condition. Both save the current location plus one in the return location stack for later use by the RET micro-instruction. The return location stack depth is 16 levels.

These micro-instructions change the execution sequence of control pattern vectors. Typically they are used with counter conditions for looping or to unconditionally jump over subroutine areas to other pattern routines.

NOTE JSR or CJSR must not be the first instruction of a nested subroutine because the Fusion hardware does not support two consecutive JSR cycles in real-time. The hardware must have sufficient time to save the return address and Counter LC1 on the return stack; consequently, a subroutine must contain a minimum of two vectors so the hardware can prepare for a return or loop.

The following example is a subroutine call and return to the main pattern. It is single pass subroutine—one without loops:

```
       *...*; <JSR S1> "Unconditional jump to subroutine"
        *...*;          "Return to here"
        *...*;
        *...*;
```

```
                    *...*; <JSR S1> "Jump to S1 again"
                    *...*;          "Return to here"
                    *...*;
                    *...*; <STOP>   "Terminate mainstream pattern"
                $S1 *...*;
                    *...*;
                    *...*;
                    *...*; <RET>  "Exit subroutine to mainstream"
```

The following example is a loop count subroutine with a loop, where the loop count is defined within the subroutine:

```
    *...*; <JSR S1>      "unconditional jump to subroutine"
    *...*;               "Return to here"
    *...*;
    *...*;
    *...*; <JSR S1>      "Jump to S1 again"
    *...*;               "Return to here"
    *...*;
    *...*; <STOP>        "Terminate mainstream pattern"
$S1 *...*; <LC1 15>      "Beginning of subroutine S1. specify fixed"
    *...*;               "loop count"
$L1 *...*;
    *...*; <JSR S2>     "Nested subroutine - it can use also C1"
    *...*;               "C1 restore here, from stack"
    *...*; <COND NZC1>   "Make sure condition C1 is in effect"
    *...*; <CJMP L1,DC1> "Jump to L1 until counter C1-0"
    *...*; <RET>         "Makes 15 jumps to L1 and returns"
$S2 *...*; <LC1 22>      "Beginning of subroutine S2, specify
$L2 *...*;               "fixed loop count"
    *...*; <COND NZC1>   "Make sure condition C1 is in effect"
    *...*; <CJMP L2,DCl> "Jump to L2 until counter C1=0"
    *...*; <RET>
```

JSRI and CJSRI

➥ JSRI

➥ CJSRI

Unconditionally or conditionally jump to the subroutine starting at address defined by the Indirect register.

These two micro-instructions are identical to JSR except the micro-instruction does not explicitly provide the subroutine address—it is supplied by a method controlled by the test program.

This combination of the Indirect register and micro-instructions are used for mixed-signal testing of analog modules. For example, a test program controls the flow of the pattern execution based on external measurements.

The following example shows these micro-instructions calling different subroutine as a result of changes in the Indirect register:

```
    *...*; <FLAG(CONT ON) "Sets-up the continuous mode"
    *...*; <COND CONT>
    *...*; <CJMP .>       "Wait here for computer to load register"
    *...*; <JSRI>         "Unconditional jump to subroutine.  The"
    *...*;                "actual routine (S1, S2, or S3) is"
    *...*;                "set externally by the computer"
    *...*; <STOP>         "Terminate mainstream pattern"
$S1 *...*;                "Begin of first subroutine"
    *...*;
    *...*; <RET>          "Exit subroutine to mainstream"
$S2 *...*;                "Begin of second subroutine"
    *...*;
    *...*; <RET>          "Exit subroutine to mainstream"
$S3 *...*;                "Begin of third subroutine"
    *...*;
    *...*; <RET>              "Exit subroutine to mainstream"
```

LC1, LC2, LC3, and LC4

➥ LC*n value*

value—integer from 1 to 65536, the start count for the specified counter.

Load counters C1 through C4 with the specific count. Counter Cl accesses up to 16 registers: 1 for each allowable nested level. C2 through C4 access only one register each.

Counter Value During Branching. When branching from one level to the next, Cl assumes the value given at that nested level, while C2 through C4 maintain a single count, independent of the current level. Counters are decremented by the DC1 through DC4 micro-instructions. The counter condition is True until decremented to zero. The counter width is 16 bits, with a maximum value of 65536.

Complex Pattern Using Counters. Under unexpected or unpredictable conditions, a test program with complex patterns may loop infinitely, not normally exiting. By enabling the pattern log memory on HardwareTool and then pressing the RESET button in

OperatorTool, you can stop the test program and dump it to the pattern log memory, which can help you determine where the pattern was executing for cycles prior to the reset.

The LCx example uses Counter C1 for subroutine looping applications; refer to the example in COND. The Counter C1 value is saved on a 16-level stack when a nested subroutine is called. Upon return from a nested subroutine, Counter C1 is restored to its previous value in the current subroutine loop. Counter C1 should be loaded on the first cycle after a subroutine call (JSR). For looping, use a CJMP and a decrement instruction.

LDA

➥ LDA *location*

location—label representing a DPM address location where execution of the DPM vectors starts.

Loads the DPM address register with the 32-bits of the specified DPM label.

The vector address of the DPM is initialized to a 32-bit value before the patterns are executed from that source, under control of the CPM in a user's CPM pattern mode. After this value is written, which sets the DPM pointer, 64 execution cycles should elapse prior to the first SDP micro-instruction on the first usage of the data from DPM. These required cycles fill the DPM pipeline with the selected data from the interleaved memory.

The following example loads the DPM pipeline by using the RPT micro-instruction. Any micro-instruction can be substituted up to the required number of cycles before the DPM is loaded:

```
*...*; <LDA DPM_st,SWCCPM> "Load all 32 bits of DPM start address"
*...*; <RPT 64>          "Let DPM pipeline get filled from"
*...*;                     "interleaved memory"
*...*; <RPT 65,SDP,SWCDPM> "Execute 65 DPM vectors in straight line"
*...*; <RPT 100,SWCDPM>    "Execute 100 locations, same DPM address"
*...*; <SDP,SWCDPM>        "Increment DPM pointer"
*...*; <STOP>
```

MODE

➥ MODE (*mode state, ..., mode state*)

mode—specifies the high-speed features controlled by the micro-instruction; refer to list of modes, refer to Table 2.3.

state—specifies the state of the programmed mode: ON or OFF. For a list of states, refer to Table 2.4.

Sets the mode register to any valid combination of the specified high-speed modes: IFAIL, TCI, or WXS. Specify more than one mode in a single MODE micro-instruction by using a comma to separate the next mode from the previous state.

Table 2.3: Modes for CPM MODE Micro-Instruction

Mode	Description
DNLD	Downloads data to the Data Processor Response (DPRO) board. If DNLD is ON, logging is forced on, and the DPRO board is instructed to enter the log response mode and to download data. The DPRO transfers the data to the PPC processor, which processes the data. When DNLD is OFF, logging is turned off.
IFAIL	Ignore Failures. When IFAIL is ON, functional failures are ignored: all vector cycles pass. Use this mode to initialize the device when its output states are unknown.
TCI	Test Counter Inhibit. When TCI is OFF, the test cycle counter advances by one for each pattern cycle. When TCI is ON, the counter does not increment. The cycle counter would normally be turned off for indeterminate loops, then enabled when the repeatable sequences are produced. Refer to SEQF state for an application of TCI control.

Table 2.4: States for CPM MODE Micro-Instruction

Mode	States Available
DNLD	OFF or ON
IFAIL	OFF or ON
TCI	OFF or ON

The following example shows how to mask fails during initialization. In this typical initialization, the ignore failures is fixed in the pattern instead of relying on the external mode definitions when the pattern execution is started.

```
*...*; <MODE (IFAIL ON, TCI ON)>  "Ignore failures and inhibit"
*...*;                            "count during DUT initialization"
*...*; <RPT 100>                  "Initialize DUT - power on soak"
*...*; <MODE (IFAIL OFF,TCI OFF)> "Ready to monitor fails now"
*...*;
*...*;
*...*;
*...*; <STOP>
```

NOP

➥ NOP

Steps to the next micro-instruction by incrementing the address pointer by one. It is the default operation if no micro-instruction is specified by a vector. This micro-instruction is an explicit NOP, equivalent to leaving the instruction field blank. The following example shows two ways to implement NOP:

```
*...*;        "Implied NOP"
*...*; <NOP>  "Explicit NOP"
```

RET and CRET

➥ RET

➥ CRET

Unconditionally or conditionally return from a subroutine.

The RET and CRET (when the condition is True) micro-instructions exit from a subroutine and return to the saved return location (call plus l). The value of Counter C1 prior to calling the subroutine is restored from a stack. For an example of a RET loop count subroutine, refer to JSRI and CJSRI.

RPT

➥ RPT *value*

value—integer between 2 and 65536 specifies the number of times to repeat the current CPM pattern vector.

Repeats the vector the specified number of times. It is not dependent on any previously defined condition.

The repeat counter is separate from the Loop Counters Cl through C4 and provides the only condition for this instruction. The CPM counter is 16-bits wide: maximum count is 65536. Minimum value of 2 is required for this instruction.

The following example shows consecutive repeat vectors:

```
*...*; <RPT 2>     "Do this vector twice"
*...*; <RPT 65536> "Do this vector 65,536 times"
*...*; <RPT 12345> "Do this vector 12,345 times"
*...*; <STOP>
```

RPTP

➥ RPTP *count*

count—integer between 2 and 65536 specifies the number of times to repeat the current pattern vector.

Repeats the vector the specified number of times or until it passes, which ever comes first.

This micro-instruction is similar to RPT except it has the additional mode to terminate the repeat earlier if the vector passes the tests. Known as the *match technique* that waits for the DUT to reach a specified state prior to the test continuing with the next vector.

Note the pattern pipeline causes the device to wait another 64 executions of the vector before continuing with the next vector. If this wait is unacceptable for some devices, LTX recommends the match technique described in FLAG micro-instruction, and shown on page 2-52.

The following example shows a single vector match:

```
*...*; <RPTP 1000> "Wait up to 1000 clocks for DUT to reach state"
*...*;             "Test DUT here. If matched, it will pass"
*...*;             "If not matches should fail these vectors"
*...*;
```

SDP

➥ SDP

Steps the DPM one vector by incrementing the DPM vector address at the end of a cycle.

The earliest this micro-instruction may be used is 64 cycles after the DPM pointer was set by LDA micro-instruction. DPM vector data is always available to the Pattern Selection Multiplexer for the pin. For straight-line DPM execution, this instruction is used concurrently with a RPT instruction, as shown in the example under LDA.

Be aware that using SDP does not guarantee DPM data gets to tester pins. The pin data source is selected by the data source defined as part of the Waveforms.

SEQF (CPM branching)

➥ SEQF *state*

state—On or Off.

Examines a sequence of vector executions for failures. If vectors fail up to 70 cycles before testing this flag, SEQF is turned off or on: the opposite of specified state. For example, once SEQF is reset, a failing vector will set it; refer to the second example for a typical application.

Use this flag when a match sequence requires several vectors in a sequence to match the DUT response, which is a common matching sequence. For example, a conventional loop is repeated with a counter to abort execution if no match is found. Use SEQF to track a failure in a sequence of vector executions. Once SEQF is reset, a failing vector will set it; refer to the second example for a typical application.

Be aware of the following restrictions for SEQF matches:

- Any number of passing vectors can be matched.
- No failing vectors can be matched.
- Sequential match executes at the programmed test rate; thus, the period is not restricted.

The following example of a SEQF sequential match shows how to match the pattern system to a non-deterministic DUT, except for the DUT specific aliases. Once this device is matched to the expected behavior of the tester, the rest of the pattern execution is normal.

```
*...*; <MODE (IFAIL ON, TCI ON)>   "Ignore failures to the"
                                   "computer, and inhibit test"
                                   "count during loop"
*...*; <LC3 500>                   "Do sequential match loop"
                                   "500 times maximum"
```

```
$L1 *...*; <COND NZC3>                    "Test C3 value for end of"
                                          "match, if no match is found"
    *...*; <CJMP L2, DC3>                 "If counter > 0, jump around"
                                          "stop to match vectors"
    *...*; <RPT 8>                        "Pipeline requirement"
    *...*; <STOP FAIL>                    "Match failed"
$L2 *...*; <FLAG (SEQF OFF)>              "Set SEQF flag off. If vectors"
                                          "fail, hardware sets SEQF on"
    *...*;                                "These vectors contain the"
    *...*;                                "match sequence, and strobes"
    *...*;                                "must be enabled"
"As many vectors as needed"

    *...*; <RPT 64>                       "System pipeline requirement"
    *...*; <COND SEQF>                    "Let's see if SEQF is on"
    *...*; <CJMP L1>                      "If it is, no match, go to L1"
    *...*; <MODE (TCI OFF, IFAIL OFF )    "Match, resume normal mode"
    *...*;
    *...*;
    *...*; <STOP>
```

STOP and STOPF

- STOP
- STOPF

Stop the pattern execution or stop pattern execution and set the FAIL condition. They unconditionally terminate pattern execution, even if the PatternSequence thread stop address or count or both are beyond this vector.

If only the STOP instruction is specified, the system pass/fail status is determined by the hardware during pattern execution. If both instructions are specified, the functional fail indicator is set.

For examples of both micro-instructions, refer to the example under SEQF (CPM branching).

SWCCPM and SWCDPM

- SWCCPM
- SWCDPM

Set the waveform control select mode bit to CPM or DPM.

In the single vector mode (SVM), these micro-instructions select the source of Waveform Control data, which generates the waveshapes. They do not select the mode, but each vector is selected vector-by-vector.

When using SWCDPM, do not select the DPM as source for at least 64 cycles after a LDA instruction to prevent indeterminate data from the DPM Waveform Control system.

For CPM, SWCCPM is not required. In contrast, the DPM selection must be explicit for every vector requiring DPM waveform control.

NOTE In the *Waveform* section of this chapter another form of DPM selection is described. Do not confuse the two. SWCDPM selects ALL pins to have their waveshapes controlled by the DPM. In the WaveformTable, an alias can be defined, on a pin basis, to select DPM data, which includes the binary information, signal direction, and strobe masking information, but not waveshapes.

The following example of SWCDPM serial shift shows how the control micro-instructions serially shift DPM words onto a tester channel that connects to the DUT:

```
*...*; <LDA DPM_st, SWCCPM> "DPM start address"
*...*; <RPT 64>             "Let DPM pipeline get filled"
*...*; <COND NZC2>          "Enable C2=0 condition"
*...*; <JSR SS1>            "Unconditional call to apply serial data"
*...*;                      "Apply parallel data to device"
*...*;
*...*; <JSR SS1>            "Unconditional call to apply serial data"
*...*;                      "Apply parallel data to device"
*...*;
*...*;
*...*; <STOP>
$SS1
*...*; <LC2 99>             "Do 100 times 16 = 1600 serial shifts"

*...*; <SHD 16,SWCDPM>      "Do 16 cycles, shifting DPM data"
*...*; <CJMP .-1,DC2>       "Do another 1600 cycle shift"
*...*; <RET>
```

The following example is a complete pattern file that shows several CPM-type syntax aspects:

```
enVisionObject:"bl8:R5.6";
Pattern Parallel_pat {
Comment = "Example for a CPM type pattern.";
```

```
Type CPM;
Default SignalHeader SH0;
Default WaveformTable _0;

$Start_here

* 0000000000....0000000000....00 *; < LDA DPM_vectors, SWCCPM >
* 0000000000....0000000000....00 *; < RPT 64 >
* 0010000000....0000000000....00 *;
* 0000000000....0000000000....00 *;
* 0000000000....0000000000....00 *;
$leak_sp
* 0010001001000010111110101....10 *;
* 0010001001000010111110101....10 *;
* 0010001001000010111110101....10 *;
* 1010001001100010111110101....10 *; <RPT 4068, SDP, SWCCPM >
* DDDDDDDDDDDDDDDDDDDDDDDDDDDDDD *; < RPT  19, SDP, SWCDPM >
* 1111100100100000100111001....01 *; <RPT 4068, SDP, SWCCPM>
* DDDDDDDDDDDDDDDDDDDDDDDDDDDDDD *; < RPT  19, SDP, SWCDPM >
* 1010101110100010000110101....10 *; <RPT 4068, SDP, SWCCPM>
* DDDDDDDDDDDDDDDDDDDDDDDDDDDDDD *; < RPT  19, SDP, SWCDPM >
* 1110111111111000000010111....10 *; <RPT 4068, SDP, SWCCPM>
* DDDDDDDDDDDDDDDDDDDDDDDDDDDDDD *; < RPT  19, SDP, SWCDPM >
* 1010000110111010000101101....00 *; <RPT 4068, SDP, SWCCPM>
* DDDDDDDDDDDDDDDDDDDDDDDDDDDDDD *; < RPT  18, SDP, SWCDPM >
* 1110110101111110101010011....11 *; <RPT 4068, SDP, SWCCPM>
* DDDDDDDDDDDDDDDDDDDDDDDDDDDDDD *; < RPT  18, SDP, SWCDPM >
* 1011011100100000001010100....00 *; <RPT 4068, SDP, SWCCPM>
* DDDDDDDDDDDDDDDDDDDDDDDDDDDDDD *; < RPT  18, SDP, SWCDPM >
* 1110010000101111110011110....00 *; <RPT 4068, SDP, SWCCPM>
* DDDDDDDDDDDDDDDDDDDDDDDDDDDDDD *; < RPT  18, SDP, SWCDPM >
* 1111001010100000010001011....10 *; <RPT 4068, SDP, SWCCPM>
* DDDDDDDDDDDDDDDDDDDDDDDDDDDDDD *; < RPT  18, SDP, SWCDPM >
* 1111000110110100000001010....11 *; <RPT 4068, SDP, SWCCPM>
* 0110011100011010110110100....11 *;
* 0110011100011010110110100....11 *;
}
```

APG Micro-Instructions

APG micro-instructions are a set of instructions that generate algorithmic patterns.

Modes

APG instructions have two modes:

- Address Mode APG Instructions

- Data Mode APG instructions

Types of APG Micro-Instructions

Each mode has two types of micro-instructions, which refer to a pin group; thus, you can have two pin groups for address mode micro-instructions and two pin groups for data mode micro-instructions.:

- address mode: `type 5` and `type 6`
- data mode: `type 7` and `type 8`

NOTE In the following sections of this chapter, `type 5` pin group is referred to by X; `type 6, Y`; `type 7, D`; and `type 8, E`.

Address Mode APG Instructions

NOTE X = type 5 pin group and Y = type 6 pin group.

➥ `X=XF`

➥ `Y=YF`

Hold the counter value. No operation. Referred to as `NOP`.

➥ `X=XB`

➥ `Y=YB`

Output the BGD register. Counter is NOP. Referred to as `BGD`.

➥ `X=~XF`

➥ `Y=~YF`

Invert the cycle *n* counter output. Counter is NOP . Referred to as `INV`.

➥ `XB=XF`

➥ `YB=YF`

Copy the Foreground data from cycle *n* into the Background register. Foreground register is `NOP`. Referred to as `PUT`.

➥ XF=0

➥ YF=0

Load the counter with 0x0000. Referred to as Load or Reset.

➥ XF++

➥ YF++

Increment value of Foreground register by 1. Referred to as INC.

➥ XF--

➥ YF--

Decrement value of Foreground register by 1. Referred to as DEC.

➥ XF==XB

➥ YF==YB

Transfer the contents of the Background register into the Foreground register (counter) or transfer the contents of the Foreground register (counter) into the Background register. Referred to as SWAP.

Data Mode APG instructions

NOTE D = type 7 pin group and E = type 8 pin group.

➥ D=DF

➥ E=EF

Hold the counter value. No operation. Referred to as NOP.

➥ D=DB

➥ E=EB

Output the BGD register. Counter is NOP. Referred to as BGD.

➥ `D=~DF`

➥ `E=~EF`

Invert the cycle *n* counter output. Counter is NOP. Referred to as `INV`.

➥ `DB=DF`

➥ `EB=EF`

Copy the Foreground data from cycle *n* into the Background register. Foreground register is NOP. Referred to as `PUT`.

➥ `DF=0`

➥ `DF=0`

Load the counter with `0x0000`. Referred to as `Load` or `Reset`.

➥ `DF<<1`

➥ `EF<<1`

Left shift the data in the Foreground register and fill the LSB with 1. Referred to as `LSH1`.

➥ `DF<<0`

➥ `EF<<0`

Left shift the data in the Foreground register and fill the LSB with 0. Referred to as `LSH0`.

➥ `DF==DB`

➥ `EF==EB`

Transfer the contents of the Background register into the Foreground register (counter) or transfer the contents of Foreground register (counter) into the Background register. Referred to as `SWAP`.

Data Pattern Type (DPM)

NOTE To execute a DPM-type patterns, at least one CPM vector is required. You load DPM patterns and then write one CPM pattern to control the execution of those DPM patterns. This is a user CPM pattern; refer to Control Pattern Type (CPM).

This type programs a pattern memory of the same name, like a truth table. This data specifies the device state (high or low) and refers to the hardware timing and waveform generators. Each vector in the pattern file programs a unique memory address in the memory and is executed as a single pattern vector. Also, refer to DPM Applications and DPM Hardware.

It may include optional micro-instructions that re-execute the same vector a specified number of times, reducing the required number of pattern locations; refer to DPM Micro-Instructions.

DPM Applications

DPM is typically used for large vector sequences executed in a straight-line manner.

The DPM available for user patterns is slightly less than the size of DPM because some DPM is used for Stored States Levels, and some for the virtual memory for CPM patterns; refer to CPM Overlays.

DPM Hardware

Table 2.5 lists the DPM specifications.

Table 2.5: DPM Specifications

DPM Size	16M x 6 (standard) 32M x 6 (optional) 64M x 6 (optional)
Per-Vector Repeat Count	2047
Opcode Field	12 bits
Input Data Rate/Pin	24 bits per 6.4 ns
Maximum Vector Rate	156.25 MHz
Operating Modes	DPM auto-sequence or under CPM control
CPM micro-instruction for DPM Control	Step DPM address

DPM Stored States Levels

This enVision release introduces Stored States Levels that now loads and stores the levels data into the DPM as it loads the timing, modes and calibration setup data. This stored data is read out in parallel to the appropriate registers.

DPM Micro-Instructions

The DPM micro-instructions are a small set of instructions that define the number of executions of each vector.

NOP

➥ NOP

Same syntax as its CPM counterpart, NOP.

RPT

➥ RPT

Same syntax as its CPM counterpart, NOP; however, the number of RPT counts is from 1 to 2000.

SHD

➥ SHD *count*

Shifts right the DPM data by the number of bits specified by count.

> *count*—integer between 2 and 16 specifies the number of times to repeat the current pattern vector while shifting the data.

This micro-instruction is used on physical tester pins in groups of 16; consequently, be aware of the pin groupings when designing the AdapterBoard so you do not put serial pins in the same group of the 16 physical channels.

It is nearly identical to the RPT micro-instruction except the Data Pattern Shifter is enabled for the duration of the instruction execution. Serial shift (LSSD) applications with serial chains greater than 16 use this micro-instruction inside a CPM loop repeated a number of times.

With this micro-instruction, a DPM vector is parallel loaded into the Data Shifter block of the hardware and shifted from higher-to-lower order channels the specified number of times. In other words, data in the LSB of the shift register is shifted into the MSB (modulo 16), creating a circular shift. Shift is 2 to 16 places, specified by count.

The following example shows how a 40-bit serial chain is serially shifted in 16-bit DPM words onto a tester channel that connects to the DUT:

```
*...*; <SHD 16> "Do 16 cycles, shifting DPM data"
*...*; <SHD 16> "Do another 16 cycle shift, etc."
*...*; <SHD 8>  "This finishes the total 40 bits"
*...*;          "Apply parallel data to device"
*...*;
*...*; <SHD 16> "Do 16 cycles, shifting DPM data"
*...*; <SHD 16> "Do another 16 cycle shift, etc."
*...*; <SHD 8>  "This finishes the total 40 bits"
*...*;          "Apply parallel data to device"
*...*;
*...*; <SHD 16> "Do 16 cycles, shifting DPM data"
*...*; <SHD 16> "Do another 16 cycle shift, etc."
*...*; <SHD 8>  "This finishes the total 40 bits"
*...*;          "Apply parallel data to device"
*...*;
```

The following is a complete pattern file that shows several aspects of the DPM type syntax:

```
enVisionObject:"bl8:R5.6";
Pattern Z8600 {
Type DPM;
Default SignalHeader SH1;
$Funct_st
*2253333 ....... .2 . .... ................* _0 ;<RPT 010 >
*2252333 ....... .2 . .... ................* _0 ;<RPT 007 >
*2252333 HLHLHHL L2 L LLHL TTTTTTTTTTTTTTTT* _4 ;
*2252333 HLLLHHL L2 L LLHL TTTTTTTTTTTTTTTT* _4 ;
*2252333 HLHLHHH L2 L LLHL 0000000011101010* _6 ;
*2252333 HHHLHHH L2 L HHHH HHHHHHHHHHHHLLHL* _2 ;
*2252333 HLHLHHL L2 L LLHL TTTTTTTTTTTTTTTT* _4 ;
*2252333 HLLLHHL L2 L LLHL TTTTTTTTTTTTTTTT* _4 ;
*2252333 HLHLHHH L2 L LLHL 0000000001000000* _6 ;
*2252333 HHHLHHH L2 L HHHH HHHHHHHHHHHHLHLL* _2 ;
*2252333 HLHLHHL L2 L LLHL TTTTTTTTTTTTTTTT* _4 ;
*2252333 HLLLHHL L2 L LLHL TTTTTTTTTTTTTTTT* _4 ;
*2252333 HLHLHHH L2 L LLHL 1111010000000000* _6 ;
*2252333 HHHLHHH L2 L HHHH HHHHHHHHHHHHLHHL* _2 ;
```

```
*2252333 HLHLHHL L2 L LLHL TTTTTTTTTTTTTTT* _4 ;
*2252333 HLLLHHL L2 L LLHL TTTTTTTTTTTTTTT* _4 ;
*2252333 HLHLHHH L2 L LLHL 1111010011110100* _6 ;
*2252333 HLHLHHH L2 L LLHL TTTTTTTTTTTTTTT* _4 ;<RPT 004 >
*2252333 HHHLHHH L2 L LLLL LHLLLLLLLLLLLLL* _2 ;
*2252333 HLHLHHL L2 L LLHL TTTTTTTTTTTTTTT* _4 ;
*2252333 HLLLHHL L2 L LLHL TTTTTTTTTTTTTTT* _4 ;
*2252332 LLHHHHH HH L LLHL TTTTTTTTTTTTTTT* _4 ;<RPT 004 >
*2252332 LLLHHHH HH L HLHL HHHLHLLLHHHLHHHH* _2 ;
*2252332 LLLHHHH HH L LLHH HHHHLLLHLLLLHLL* _2 ;
*2252332 LLHHHHH HH L LLHH HHHHLLLHLLLLHLL* _2 ;<RPT 002 >
*2252332 LLLHHHH HH H HLHL HHHLHLLLHHHHLLL* _2 ;
*2252332 LLLHHHH HH H LLHH HHHHLLLHLLLLHLL* _2 ;
*2252332 LLHHHHH HH H LLHH HHHHLLLHLLLLHLL* _2 ;
*2252332 LHHHHHH HH H LLHH HHHHLLLHLLLLHLL* _2 ;
*2252332 HHHHHHH HH H LLHH HHHHLLLHLLLLHLL* _2 ;
*2252332 LLHHHHH HH H LLHH HHHHLLLHLLLLHLL* _2 ;
*2252332 LLLLHHH HH L LLLL LHLLLHLLHLLHLHL* _2 ;
*2252332 LLLLHHL HH L LLHL TTTTTTTTTTTTTTT* _4 ;
*2252332 LLHHHHL HH L LLHL TTTTTTTTTTTTTTT* _4 ;
*2252332 LLHHHHH HH L LLHL 1000100100100100* _6 ;
*2252332 LLHHHHH HH L LLHL TTTTTTTTTTTTTTT* _4 ;<RPT 004 >
*2252332 LLLHHHH HH L HLHL HHHLHLLLHHHLHHHH* _2 ;
$Funct_sp
*2252332 ....... .. . .... ..............* _4 ;<RPT 012 >
}
```

Keep Alive (KAP) Pattern Type

The Keep Alive Pattern (KAP) is a special pattern that loops until the next pattern is executed. It is very similar to the existing CPM/DPM patterns; however, it does have certain programming methods and restrictions. KAP runs between the execution of the threads or flow nodes, or both. The transition from a functional pattern to KAP, and vice versa, is seamless to the DUT; refer to KAL Pattern Properties.

The benefit of KAP is that a test program will take less time to run because a device is initialized once for the entire execution of a test program, instead of re-initialized each time a test is run. Also, the Pattern Sequencer is not busy while the KAP is running; thus, the next flow node can be setup while the device is kept in a known state.

Also, refer to Example: Small Keep Alive Pattern and Example: Keep Alive Waveform Cell.

Creating a KAP Using WaveTool

1. In WaveTool, create Keep Alive Pattern and Keep Alive waveform cells.

2. Add the pattern to a thread as the last entry of that thread.

3. Once the pattern is added to the thread, the action (icon) changes, showing the pattern is now a KAP.

4. When the thread executes, all patterns in the thread will run, then the KAP will start. Also, refer to Terminating KAL.

KAP Restrictions

a. KAPs must be equal to or less than 64 vectors (128 vectors in DVM). The end location signifies the length of the loop in vectors.

b. Low-speed mode switching between pattern executions joined by KAP is not supported. These modes include: `MuxMode`, SyncClock, SVM, DVM, and Differential.

c. Base period switching between pattern executions joined by KAP is not supported.

d. No strobes in the KAL timeset.

e. No labels, or micro-instructions in KAL except for the DPM `RPT` micro-instruction, maximum 2047 repeats.

f. Must use the same waveform reference on all vectors.

g. Each pin can only use up to 2 aliases because the KAP has up to 2 user drive-only timesets. enVision constructs a single system timeset from these user timesets.

h. KAP vectors that specify alias `k` (data 0) will contain vector mask data as TG2, that is, masking TG2. Vectors containing alias `K` (data 1) will mask TG3. This technique provides (1) a single user timeset using all four TGs or (2) two user timesets sharing the four TGs. For example: you specify alias **`k/K`** as `DriveOn`, `DriveData`, enVision creates the one system time set:

- `TG1:DriveOn`
- `TG2:DriveHigh`

- TG3:DriveLow
- TG4:NoAction

KAL Pattern Properties

a. When the thread executes, all patterns in the thread will run, then the KAP will be started.

b. enVision verifies the specified waveshapes of the KAP cells are within the capabilities of the Fusion hardware.

c. KAL pattern continues to run until the next thread is executed. Next pattern execution causes the KAP to complete its current loop, then the new pattern controls the test program. Also, refer to Terminating KAL.

d. All fails are ignored.

e. Pattern Fail logs are disabled.

Terminating KAL

- KAP exits after the next execution of the last vector, and it clears the KAL flag.
- When exiting from KAL, all logs and cycle counter initialize.

Example: Small Keep Alive Pattern

```
Pattern MyKeepAlive {
Type KAP;
Default SignalHeader Hd1;
Default WaveformTable KAFmt;

* k000 1 Ar11 xxxxxxxx *
* k100 1 Ar11 xxxxxxxx *
* k001 1 Ar01 xxxxxxxx *
* K000 1 Ar11 xxxxxxxx *
* K000 1 aR11 xxxxxxxx *
* k100 1 aR10 xxxxxxxx *
* k100 1 Ar11 xxxxxxxx *
* K000 1 Ar01 xxxxxxxx *
* K100 1 ar11 xxxxxxxx *
* k000 1 aR11 xxxxxxxx *
* K100 1 AR10 xxxxxxxx *
}
```

Example: Keep Alive Waveform Cell

```
Cell "clk_out+data_A" K/A KeepAliveWaveform {
    Data 6/7 KeepAlive;
    Drive {
      Waveform { DriveOn @ "0nS"; DriveData @ "10nS";
}
    }
}
```

Fast Data Mode Pattern Type (FDM)

This optional type executes patterns at twice the frequency of the standard mode. It is described in the *Fast Data Mode Option* chapter in the *enVision Digital Programming* manual.

The FDM pattern type typically programs the DPM pattern memory, like a truth table, internally set in FDM mode. Each vector in the pattern file programs a unique memory address in the memory and is executed as a single pattern vector; refer to Data Pattern Type (DPM).

Serial Scan Patterns

Devices use serial scan testing as a method for increasing fault coverage during testing without requiring a tester to generate very large sequential patterns. Scan testing is usually included in the design of a device to support Built-In Sequential Test (BIST).

Scan patterns consist of a:

- DPM pattern containing the scan chain vectors and parallel (normal) vectors.
- CPM pattern that controls the execution of the scan pattern.

Serial Scanning Method

Scan testing places a small scan blocks of digital logic around the boundaries of the core digital logic of a device. These blocks are connected serially, forming a chain in the device that is tied to a single input and a single output pin; refer to Creating a Serial Scan Pattern. Devices with very large blocks of digital logic may have several unique scan chains.

Scan testing shifts a serial stream of bits synchronous to an input clock and simultaneously strobes a serial stream of output data. The serial scan bits are referred to as a *scan vector*, which is usually, but not always, fixed in length.

To start serial testing, the device is put into the proper state by running parallel vectors. Parallel vectors that run previous to and between scan vectors are defined normally. Once the device is in the proper state, the rest of the device inputs and outputs are held static, while the scan vector is clocked into the device. Parallel vectors are then run, and the data in the scan chain is passed to the digital core logic. The resulting output of the logic is captured by the scan chain. The output is clocked out serially as the next input scan vector is clocked in.

Creating a Serial Scan Pattern

You must write a DPM pattern that runs the scan vectors and scans parallel vectors; refer to Creating a Scan DPM Pattern. EnVision uses a special `SignalHeader`, `Scan_HDR`, to define the scan vector and to organize and pack the data in the DPM memory used by a CPM pattern. You must also create a CPM pattern to control the execution of the scan pattern.

LTX is currently working on having the CPM pattern auto generated, like normal stand-alone DPM patterns.

Capture Tool and Pattern Tool can help you create and debug a scan pattern. Pattern Tool can display DPM scan vectors, so you can easily modify the bits in the scan vector. All pattern data in the scan vector is accessible for review, and the position of the data in the scan chain is displayed. Capture Tool can display the scan vectors position in the pattern and the bit position also in the header information.

Creating a SignalHeader Scan_HDR

When writing a serial scan pattern, you must use the `SignalHeader`, `Scan_HDR`, which specifies the pin name, number of elements in the scan chain, fill alias to be used, and fill mode. It enables you to write scan vectors in a DPM pattern:

```
SignalHeader Scan_HDR {
   Rate 1;
   Default Format Fixed 1;
   Signals {
    Pin1 (Scan, ChainLength = 80, Fill = X, PostFill;}
```

```
    %Pin2 { Scan, ChainLength = 80, Fill = X, PostFill;}
 }
}
```

where:

`ChainLength`—an integer representing the maximum length of the scan vector used by this pin.

`Fill`—an alias character to fill the scan vector to the size specified by the `ChainLength` parameter.

A fill mode is required:

- `PreFill`—the `Fill` alias is inserted at the beginning of the scan chain until the scan vector equals the size specified by the `ChainLength` parameter.
- `PostFill`—the `Fill` alias is appended to the end of the scan vector.

`Fill` alias characters are not necessary if all scan vectors in the pattern are always the same length, as defined by `ChainLength`; however, the `Fill` alias and the `Fill` mode must still be specified. In addition to manually adding the `SignalHeader Scan_HDR` to the .eva file, you can also use the `SignalHeader` Editor of Pattern Tool to create the `SignalHeader Scan_HDR`.

Creating a Scan DPM Pattern

Due to the amount of data in a scan pattern, running a scan pattern in CPM may be impractical; thus, a full scan pattern, including the scan vectors can be loaded in DPM memory. The DPM scan pattern follows the same constructs and restrictions as normal DPM patterns. The only difference is that the actual serial scan vectors use the `Scan_HDR`, and the parallel vectors use a `SignalHeader` that defines the pins used in the parallel vectors; refer to Sample DPM Scan Pattern.

Sample DPM Scan Pattern

```
enVisionObject:"bl8:R10:S3.0";

Pattern scan_ck {
Type Dpm;
Default WaveformTable scan_0;
Default SignalHeader SH0;
```

```
$Scan_dpm_start
*MMMMMMMMMM L MMHHHMM MMM M MMMML*; < RPT 2 >
*MMMMMMMMMM L MMMMMMM MMM M MMMML*;
*MMMMMMMMMM L MMHMHMM MMM M MMMML*;
"First Scan Vector starts here:"
$Scanvec1
*HHLLHHLL HHLLHHLL HHLLHHLL HHLLHHLL HHLLHHLL
 HHLLHHLL HHLLHHLL HHLLHHLL HHLLHHLL HHLLHHLL;
 MMMMMMMM MMMMMMMM MMMMMMMM MMMMMMMM MMMMMMMM
 MMMMMMMM MMMMMMMM MMMMMMMM MMMMMMMM MMMMMMMM;
* scan_0 Scan_HDR;
*MMMMMMMMMM L MMHHHMM MMM M MMMML*; < RPT 2 >
*MMMMMMMMMM L MMHHHM1 MMM M MMMML*;
$ScanVec2 "
Second Scan Vector starts here:"
HHLLHHLL HHLLHHLL HHLLHHLL HHLLHHLL HHLLHHLL
HHLLHHLL HHLLHHLL HHLLHHLL HHLLHHLL HHLLHHLL;
10011001 10011001 10011001 10011001 10011001
10011001 10011001 10011001 10011001 1001100M;
* scan_0 Scan_HDR;
*MMMMMMMMMM L MMHHHMM MMM M MMMML*; < RPT 3 >
}
```

In the sample DPM scan pattern, note the actual scan vectors start the same as the parallel vectors with the * character. The scan vector contains the scan data for the pins in the order defined by `Scan_HDR`. The number of scan data bits in a vector does not have to equal the number defined by the `ChainLength` parameter because the compiler uses the `Fill` alias character to make the vector the proper length.

Also, note the scan data for each pin is separated by a semicolon (;); the end of the scan vector is denoted by the second * character. The scan vector must be tied to the `SignalHeader Scan_HDR`. header. You can tie scan vectors to different scan `SignalHeaders`, if necessary.

Creating a Scan CPM Pattern

As with all DPM patterns, a CPM pattern is executed to control the execution of the DPM pattern. Usually, enVision generates a CPM pattern to execute the DPM pattern specified in the pattern sequence; however, scan patterns are not supported by this feature. As a result, you must write the CPM pattern as an `.evo` file so it becomes part of the pattern sequence for the scan pattern. Also, refer to Scan Patterns Requiring Large CPM.

In the Sample CPM Scan Pattern, a sample CPM pattern executes a scan DPM pattern listed in the Sample DPM Scan Pattern.

Sample CPM Scan Pattern

```
enVisionObject:"bl8:R10.0";

Pattern scan_chk_Cpm {
Type Cpm;
Default WaveformTable scan_0;
Default SignalHeader SH0;

$scan_chk_Cntl_st
*MMMMMMMMMM L MMMMMMM MMM M MMMML* hskp; < LDA DPM_scanpat, SWCCPM >
*MMMMMMMMMM L MMMMMMM MMM M MMMML* hskp; < LC1 1 >
*MMMMMMMMMM L MMMMMMM MMM M MMMML* hskp; < RPT 127 >
*MMMMMMMMMM L MMMMMMM MMM M MMMML* hskp;
*ddddddddddd d ddddddd ddd d ddddd* scan_0; < SWCDPM, SDP, RPT 3 >
*ddddddddddd d ddddddd ddd d ddddd* scan_0; < SWCDPM >

$rep_loop_sc
*MMMMMMMMMM L MMHHHMM MMM M MMMML* hskp; < SWCCPM, LC2 78 >
*MMMMMMMMMM L MMHHHMM MMM M MMMML* hskp; < COND NZC2 >
*MMMMMMMMMM L MMHHHMM MMM M MMMML* hskp; < SDP, JSR scan_mode_sc >
*MMMMMMMMMM L MMHHHMM MMM M MMMML* hskp;
*ddddddddddd d ddddddd ddd d ddddd* scan_0; < SWCDPM, SDP, RPT 2 >
*ddddddddddd d ddddddd ddd d ddddd* scan_0; < SWCDPM >
*MMMMMMMMMM L MMHHHMM MMM M MMMML* hskp; < COND NZC1, SWCCPM >
*MMMMMMMMMM L MMHHHMM MMM M MMMML* hskp; < CJMP rep_loop_sc, DC1 >

$scan_chk_Cntl1_sp
*MMMMMMMMMM L MMMMMMM MMM M MMMML* hskp; < STOP >
*MMMMMMMMMM L MMMMMMM MMM M MMMML* hskp; < RPT 128 >

$scan_mode_sc
*MMMMMMMMMM C HDHHHMQ MMM M MMMML* scan_md; < SDP, CJMP ., DC2 >
*MMMMMMMMMM C HDHHHMQ MMM M MMMML* scan_md; < SDP, RET >
}
```

In the example, the first vector's micro-code loads the DPM pointer with the start of the DPM scan vector. The `SWCCPM` command, which gets the pattern data from the CPM, is not necessary in this case, but including it is good programming practice.

The micro-code in the second vector is a counter load. It specifies the number of times the `RPT` loop (starting at the `rep_loop_sc` label) is repeated. The number in this example is not important: it is usually the number of actual scan vectors in the DPM pattern.

The micro-code in the third vector specifies enough time to load the DPM pointer, while the fourth vector completes the repeats and gets ready to begin executing the DPM vectors. None of these vectors executes a DPM scan vector—they only set up the pattern execution.

In the example, these vectors are assigned to a special `WaveformTable`, `hskp`. They are run a very fast rate (usually at the fastest tester rate), and the pins are set to a NRZ format to keep the levels static. LTX recommends keeping the actual DUT inputs in a static condition while these vectors are executed.

The fifth and sixth vectors use the `SWCDPM` micro-code to instruct the pattern engine to source its pattern data from the DPM memory. This vector contains the `d` alias, which in this example, is the alias of the data source defined as `Other` in the `Data Source` block of the Waveform Cell Tool for the `scan_0 WaveformTable`. Thus, the DPM pattern data from the initial parallel vectors of the scan pattern is executed.

The fifth vector also contains the `SDP` micro-code that increments the DPM pointer in synchronization with the execution of the CPM pattern. `SDP` is not specified in the sixth vector so that the correct DPM vector for the upcoming repeat loop is pointed to by the DPM pointer. The repeat loop must increment the DPM pointer.

The next eight vectors comprise the repeat loop that executes most of the DPM scan pattern. The first vector of the loop switches the pattern data source to CPM and loads Loop Counter 2 with a value that is two less than the desired length of the scan chain. The next vector sets up a non-zero count for Loop Counter 2. These two vectors use the `WaveformTable hskp` and keep the levels on the DUT pins static. The next vector also keeps the DUT pins static, but the micro-code on this vector increments the DPM pointer and then jumps to the subroutine at the `scan_mode-sc` label.

The subroutine starting at the `scan_mode_sc` label is where the actual scan vector gets executed. In the first vector of the subroutine, the `SDP` instruction normally advances the DPM pointer to the next vector, but this instruction in a scan vector steps through the scan vector as if the scan vectors were a set of parallel vectors.

Also in the first vector, the `CJMP` and `DC2` instructions control the number of times the vector is executed. The number of times this sample vector is executed equals one less than the length of the scan chain. This vector continues executing until Loop Counter 2 is zero.

Loop Counter 2 decrements by one each time this vector is executed. The next vector finishes the execution of the scan vector, and the subroutine exits normally.

The next vector executed is the next one in the CPM pattern (tenth vector in example). Be aware the vectors in the scan subroutine are tied to a unique `WaveformTable` (`scan_md` in example), which is set up to drive the scan clock, usually with a RZ format (`C` alias in subroutines vectors). The driven scan pins have their timing data defined in this `WaveformTable`. Also, the Data Source field of the Waveform Cell must be set to Scan so the scan data is sent to the driver.

The vector after the call to the scan subroutine (tenth vector) is a convenient place for the subroutine to return to. The following two vectors execute the parallel vectors that must be executed between scan vectors. They act like the initial CPM vectors that execute the first parallel vectors of the scan DPM pattern. The final two vectors in the loop set up the condition for a non-zero count of Loop Counter 1, which controls the number of times the loop is executed.

In this example, the number of parallel vectors between scan vectors is constant. If the number of vectors is not the same, you will have to modify the CPM pattern so it has different control loops that execute the required number of parallel vectors.

Scan Patterns Requiring Large CPM

The worst-case CPM pattern is having two vectors to switch between a parallel vector block and a scan chain. The first vector executes the parallel vectors (source select is set to DPM) with a repeat for the number of parallel vectors to execute. The second vector executes the scan chains (source select of the serial pins is set to serial), and the repeat count is the size of the scan chain. For example, if your scan pattern has 1000 scan chains, over 2000 vectors of CPM may be required for the scan control pattern.

Scan patterns have limited amount of CPM available for a thread execution on the VX IV tester. A test program may run out of CPM if a thread contains scan patterns requiring large CPM patterns to control them; refer to Workarounds for Scan Patterns with Limited CPM.

Workarounds for Scan Patterns with Limited CPM

A user workaround to the CPM hardware limitation may be successful if your scan pattern has a certain structure. Consequently, your scan pattern may still run out of CPM if it does not meet certain requirements. Depending on the scan pattern, you can:

- create a uniform scan pattern (if possible), or
- control the number cycles between scan vectors.

Uniform Scan Pattern Method

To create a uniform scan pattern (number of parallel vectors between scan vectors is the same in the entire pattern), the CPM patterns should be less than 20 vectors. Create a uniform scan pattern by using loops, counters and subroutines to control the scan pattern execution. With this software approach a large number of scan patterns are in the same thread, without using up the entire CPM.

Sequence Table Method and Scan Patterns

The second method uses the hardware Sequence Table to jump between the parallel vectors and scan chain vectors. This approach to controlling the number of cycles between scan vectors requires at least 64 cycles of parallel vectors between the scan chains (pattern size limit of the Sequence Table). It needs only one or two CPM vectors for each unique scan `SignalHeader` in the thread. For example, assume:

- Thread consisting of a three-scan pattern used as `SignalHeader` Scan1, with scan pins **`TDI`** and **`TDO`**. Chain length is 1000.
- Two- scan pattern used as `SignalHeader` Scan2, with **`RD0`** and **`RDI`** scan pins. Chain length is 2000.

Using the Sequence Table to control the scan patterns requires only two CPM vectors for the entire thread. The Sequence Table for this thread alternates calls to the DPM pattern memory for executing the parallel vectors, and to the CPM vector (one of these two CPM vectors depending on which scan chain was executed) vectors for executing the scan chains.

Offline Pattern Compiler

The enVision pattern compiler (epc) is a stand-alone program that compiles ASCII pattern files (.evo files) into binary pattern files (.epf files); refer to Supported File Formats and Compiler Syntax.

The compiler runs in the standard enVision shell environment; refer to Compiler Examples. You can open a shell by creating an xterm from Operator Tool.

The epc provides three ways to compile your patterns: two for compiling patterns in existing test programs, and one for compiling patterns for new programs:

- -c: Compiling Existing Patterns Serially
- -g: Compiling Existing Patterns in Parallel
- Compiling Patterns for New Programs

Be aware of the compiler limitations. Also, epc has restrictions for sharing pattern files.

Supported File Formats

This release of enVision supports two pattern file formats: .epf and .eva. The default is .epf.

For an .epf file, patterns are not compiled when the test program is loaded, the existing .epf files are loaded. If a pattern has not yet been compiled, an error message tells you to recompile the pattern, and that the Pattern object will not be loaded.

In the Flex/CPG mode, enVision will compile patterns when the .eva file is loaded (if they need to be compiled), and will create the .flex and .cpg files. To force enVision load the old flex/cpg pattern files, execute the following command before starting the enVision launcher:

```
$ setenv PATTERN_FORMAT FlexCpg
$ launcher
```

NOTE R10.4 and below support the .flex/.cpg format for pattern files; R10.8 and above do not support these formats.

Compiler Syntax

Use epc for compiling pattern files with existing .eva files:

```
epc [-svm] [-I include_file]* [-bp binary_path]
[-sp source_path] <evo_files>*
```

svm—specifies compile for SVM (Single Vector Mode) only. Use for scan patterns and any pattern executed only in SVM; refer to Compiling Scan Patterns.

I include_file—specifies at least one file defining the following objects: PinGroups, (AliasMap or WaveformTable), and SignalHeaders. Multiple include_file options to the compiler are allowed. Example: a file with PinGroups, a second file with the AliasMap, and a third file with the SignalHeaders:

```
-I pins.evo -I AliasMap.inc -I signal_headers.evo
```

These file can have any name; extensions are not required.

bp binary_path—specifies the path of the directory where the binary files will be stored. If this optional parameter is not specified, the .epf files will be stored in the current directory.

sp source_path—path to the directory of the source pattern files. If this optional parameter is not specified, enVision expects the files to be in the current directory.

evo_files—wild card expression or list of files to be compiled, including normal or compressed files. Filenames can contain a path if you did not specify the **source_path**.

Also, refer to Compiler Examples.

Compiling Existing Patterns Serially

The -c option compiles all patterns in the program (.eva file and any external files) with one command. If you do not have access to multiple CPUs, or you do not want to compile the patterns in parallel, LTX recommends you use this option to create the .epf files so these files are created in the locations specified by the ExternalRefs for each pattern:

```
epc [-svm] -c eva_file
```

where:

eva_file—path to the .eva file (including the .eva file) for the test program to compile.

svm—optional flag, forces all patterns to be compiled only for the Single Vector Mode (default compile for SVM and DVM). If your program does not use Dual Vector Mode (DVM) patterns, use this option to speed up the pattern compile.

For example, assuming the pattern file C8086.eva is in the current working directory, use the following command:

```
$epc -svm -c C8086.eva
Compiling .//./pat_sources/Z86FREQ.evo
PatternType = VX250 CPM Pattern, NumVecs = 436, Count
       = 635
Compiling .//./pat_sources/Z86DC.evo
PatternType = VX250 CPM Pattern, NumVecs = 451, Count
       = 849
Compiling .//./pat_sources/Z8600.evo.gz
PatternType = VX250 DPM Pattern, NumVecs = 14150,
       Count = 20202
Compiling .//./pat_sources/Z86TMU.evo.Z
PatternType = VX250 CPM Pattern, NumVecs = 10, Count
       = 20257
```

Compiling Existing Patterns in Parallel

The -g option creates the files required for parallel compilation of the pattern files. A compile script (**<group_name>_CompilePats**) and pattern include (**<group_name>_epc.epi**) file are generated for each pattern group used by the test program. The pattern include file contains the compile information required by epc; refer to Compiling Patterns for New Programs. You can then execute each of these compile scripts on a different computer or CPU to compile the patterns in parallel.

NOTE The .epf file does not contain information about the PatternGroup. This compiler option is a convenience. It creates scripts that can be run in parallel. You can split the epc commands from these scripts, or combine them to create other compile scripts.

This option also creates a script called CompilePatterns that calls all group compile scripts; thus, it compiles all patterns:

```
epc [-svm] -g eva_file
```

where:

`eva_file`—is the path to the .eva file (including .eva file) for the test program to compile.

`svm`—is an optional flag, forcing all patterns to be compiled only for Single Vector Mode (default is compile for SVM and DVM). If your program does not use Dual Vector Mode (DVM) patterns, use this option to speed up the pattern compile.

For example, assuming a pattern file `C8086.eva` in the current working directory, use the following command:

```
$epc -g ../C8086_EPF.eva
Generating epc compile files for ../C8086_EPF.eva:
Creating CompilePatterns script
Creating Z86DC_pats_epc.epi pattern include file
Creating Z86DC_pats_CompilePats compile script
Creating Z8600_pats_epc.epi pattern include file
Creating Z8600_pats_CompilePats compile script
File Generation Complete
```

Compiling Patterns for New Programs

You can compile patterns with epc before the .eva file is created by creating one or more files with the extra information needed by epc:

1. Pattern compile `include` files—contain `SignalHeader` objects, alias character mapping information, and the pin groups required by the alias map. Put this information in one or more ASCII files; refer to the fourth example in Compiler Examples.

   A pattern `include` file must be created for each different set of alias mappings in your patterns; refer to Signal Direction and Alias Symbols.

2. SignalHeader object.

3. `pin_groups`—are required for the `AliasMap` or the default `WaveformTable` object. The syntax is the same as the `pin_groups` in the enVision ASCII syntax.

4. alias character mapping to the compiler—are lists of pairs or individual aliases of 0/1 data a pattern can use for a specified pin group; refer to Binary Data Characters. The compiler uses this

information for pattern error checking and for supporting the hexadecimal notation in the pattern. The zero data alias is first followed by a slash and the 1 data alias; for example, L / H, 0 / 1. An individual alias can also be specified by the alias name for a data 0 alias, or a slash followed by the alias for a data 1 alias. enVision supplies the compiler with the alias character with one of two methods:

NOTE Each pattern `include` file supports only one `AliasMap` or `WaveformTable` object.

a. `AliasMap` object.

It is a compact method of creating objects for a new program. It is a condensed version of the `WaveformTable` object: it contains one or more rows that specify a pin group and a list or comma-separated alias pairs or alias characters, and an optional pattern select. Use the following `AliasMap` syntax:

```
AliasMap {
 pin_group {data0_alias}{/data1_alias}{:pat_select},
        ... ;
 pin_group {data0_alias}{/data1_alias}{:pat_select},
        ...;
  pin_group {data0_alias}{/data1_alias}{:pat_select}
        ... ;
}
```

where:

`pin_group`—group name in quotes or a pin name expression in quotes. Pin name expression is **`pin_names`** separated by **`+`** or **`-`**.

`data0_alias`—character representing the data 0 value.

`data1_alias`—character representing the data 1 value.

`pat_select`—one of the following (`Cpm` | `Dpm` | `Apg` | `Spg`). If you do not supply `pat_select`, default is the current pattern.

Example:

```
AliasMap {
  "CLKOUT1m" /3, 0;
  "ALLOUTS" 0, /1, 2, /3, 4, /5, 6, /7;
  "CLKOUT1+CLKIN1" /9;
```

```
    "ALLINS_INC_MUX" L, h, /l, /H, A, /B, C:Spg, /D, J,
          /K, R, /S;
}
```

b. default `WaveformTable` object.

To compile patterns from an existing program, you can use the enVision `WaveformTable` object to define the alias mapping. The parser will ignore all extra information; refer to Figure 2.6.

If you use the `WaveformTable` object containing a pattern_group pin group (**`pattern_group_name.Pins`**), you must also specify the `PatternGroup` object, so that **`Pins`** is inserted in the **`pattern_group_name`**. The **`Pins`** `PinGroup`, the pattern, is not being compiled only for this pattern group.

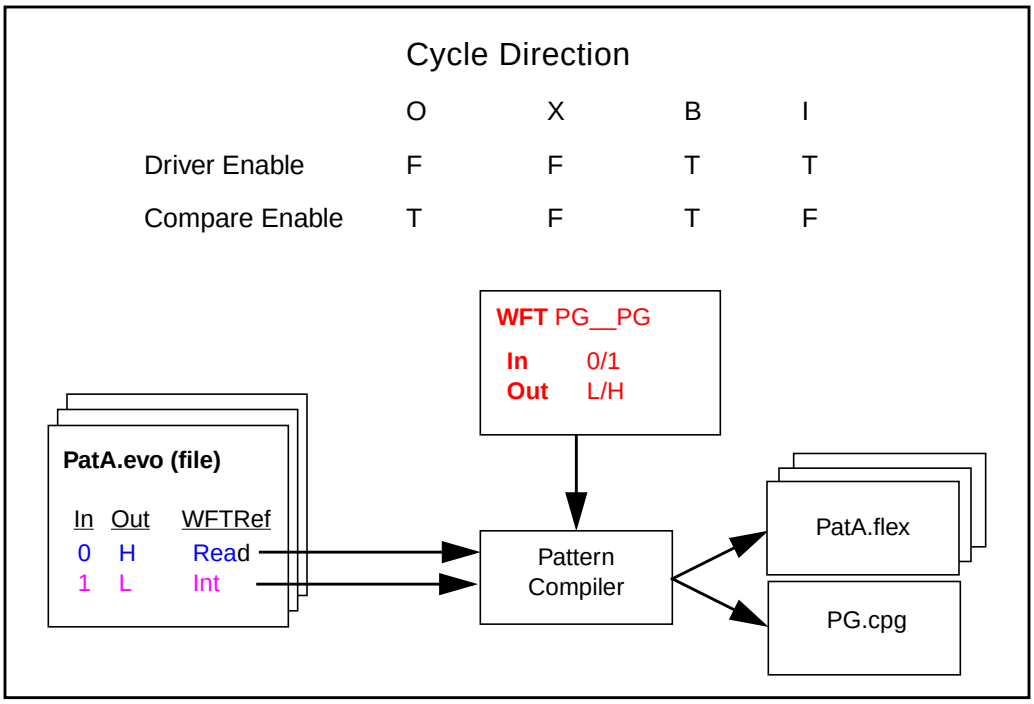


Figure 2.6: Pattern Compile Using Default PatternGroup WaveformTable

Compiler Examples

- Compile all compressed files with extension `.evo.gz` in `evos2/atpg` directory and put the `.epf` files in the `epfs/atgp` directory.

```
$ epc -svm  -I pattern_inc.evi -bp epfs/atpg evos2/
atpg/*.evo.gz
```

- Compile the r30_write_on.evo pattern in the ./Pats/VEGA directory and put the .epf file in the ./epfs/VEGA directory. Compile the pattern for both SVM and DVM execution modes.

```
$ epc -I VegaPatGrp125.evi  -bp ./epfs/VEGA ./Pats/
VEGA/r30_write_on.evo
```

- Compile the Z8600.evo and Z86TMU.evo files in the pat_sources directory and put the .epfs binaries in the pat_epfs directory. Compile the patterns in the SVM execution mode only.

```
$ epc -svm -I Z8600.inc -sp pat_sources -bp pat_epfs
Z8600.evo Z86TMU.evo
```

- Pattern include File. The contents of the Z8600.inc data file is used by the epc to compile the Z8600.evo and Z86TMU.evo pattern files.

```
PinGroup Stats {Group = Expr { String = "S0+S1+S2"; }}
PinGroup BHE_stats {Group = Expr { String =
    "BHE_S7+A19_S6+A18_S5+A17_S4+A16_S3"; }}
PinGroup AD_bus {Group = Expr { String =
    "AD15+AD14+AD13+AD12+AD11+AD10+AD9+AD8+AD7+AD6+
    AD5+AD4+AD3+AD2+AD1+AD0"; }}
PinGroup RQ_gnt {Group = Expr { String =
    "RQ_GT1+RQ_GT0"; }}
PinGroup All_bids {Group = Expr { String =
    "AD_bus+RQ_gnt"; }}
PinGroup Async {Group = Expr { String =
    "NMI+INTR+Test0"; }}
PinGroup Ins {Group = Expr { String =
    "Clk+RES+Ready+MN_MX"; }}
PinGroup All_ins {Group = Expr {String="Ins+Async";}}
PinGroup QS {Group = Expr { String = "QS1+QS0"; }}
PinGroup All_outs {Group = Expr { String =
    "QS+Stats+BHE_stats+ock+RD"; }}
PinGroup All_pins {Group = Expr { String =
    "All_ins+All_outs+All_bids"; }}
AliasMap { "All_pins" c/C, 0/1, L/H, T, .; }
SignalHeader SH1 {
    Rate 1;
    Default Format Fixed 1;
    Signals {
       %NMI INTR Clk RES Ready Test0 MN_MX %QS1 QS0
```

```
                S0 S1 S2 Lock RD %RQ_GT1 RQ_GT0 %BHE_S7 %A19_S6
                A18_S5 A17_S4 A16_S3 %AD15 AD14 AD13 AD12 AD11
                AD10 AD9 AD8 AD7 AD6 AD5 AD4 AD3 AD2 AD1 AD0 }
        }
```

Compiling Scan Patterns

Scan patterns are run in SVM; thus, use the -svm switch when compiling scan patterns. If you do not use this switch, a warning dialog tells you the epc will compile with the -svm switch. If you compile without the -svm switch, the compiler does not know the pattern is a scan pattern until it starts compiling the first scan vector. After epc determines the pattern is a scan pattern, it stops and then restarts compiling with the -svm option. Consequently, if you do not start compiling with the -svm option, your compile times will be longer than if you first used the -svm switch.

Also, refer to Serial Scan Patterns.

Compiler Limitations

The pattern compiler has two limitations with the -g or -c options. Both limitations involve parsing an .eva file with a PatternMap object:

1. If the .eva file contains a PatternMap object, the active PatternMap must be named DefaultPatternMap; otherwise, the parser will issue error messages, and the patterns will not compile, or the compiled files will not be generated.

2. If the .eva file contains a PatternMap object using expressions for the PatternName, SourcePath, CachePath or Pattern Group, the parser issues a syntax error, and the patterns will not compile, or the compiled files will not be generated.

Guidelines for Sharing Pattern Files

When sharing a pattern file between different test programs, be aware of the following guidelines:

1. Both test programs must contain the same pin_names and SignalHeader objects. The order of the pins in the Adapterboard object and assigned tester channels does not have to be identical; only the names of the pins in the pin_names object have to be identical to the pin names in the Adapterboard object.

2. This second requirement also applies to changing the pattern group a pattern belongs to.

   When a `Pattern` object is defined in the .eva file by using the `ExternalRef` statement, the object name, file path and name, and pattern group are specified. Each pattern group has a `PatternGroup WaveformTable` (sometimes called the default `WaveformTable`) associated with it. This waveform table defines the alias characters valid for a pattern in this group and the mapping of these alias characters (L/H, 0/1 etc.). During compilation, enVision verifies this waveform information of the pattern has valid aliases, and uses the alias character mapping for processing the hexadecimal notation in the pattern .evo file.

To use a pattern with a different pattern group or a different test program, the pattern group waveform tables must be compatible: they must contain the same alias definitions and groupings; however, the timing for these waveform tables (if defined) does not have to be the same. For example, the following `PatternGroup WaveformTables` for the C8086 test program are compatible:

```
WaveformTable Z86DC_pats Z86DC_pats {
  Comment =3D "Alias characters for DC patterns.";
  Cell "Z86DC_pats.Pins" c/C _C {
    Data 6/7;
  }
  Cell "Z86DC_pats.Pins" 0/1 _0_1 {
    Data 6/7;
  }
  Cell "Z86DC_pats.Pins" L/H/. _L_H {
    Data 0/1/2;
  }
  Cell "Z86DC_pats.Pins" T _T_ {
    Data 0;
  }
}

WaveformTable Z8600_pats Z8600_pats {
  Comment =3D "Functional patterns.";
  Cell "Z8600_pats.Pins" c/C _C {
    Data 6/7;
  }
  Cell "Z8600_pats.Pins" 0/1 _0_1 {
    Data 6/7;
      }
  Cell "Z8600_pats.Pins" T _T_ {
    Data 0;
```

```
    }
    Cell "Z8600_pats.Pins" L/H/. _L_H {
      Data 0/1/2;
    }
  }
```

3. If the pattern uses hexidecimal notation, the PinGroups for the buses in the pattern must be the same in both test programs. Also, the group_name and pin names must be defined in the same order in both test programs.

4. If the pattern is an APG pattern, both test programs must have the same memory descriptor objects defined.

Waveforms

The waveform formatting and timing of the pattern data sent to the DUT is defined by the WaveformTable object. These objects define the waveforms for each alias character (also known as a *symbol*) of a pattern vector; refer to Types of Waveform Tables.

Types of Waveform Tables

The WaveformTable is either a stand-alone object or part of a PatternGroup; see Figure 2.7:

Default WaveformTable—is mainly for pattern compilation only; refer to Creating a WaveformTable Object. The name of this table includes the PatternGroup name; refer to WaveformTable Inheritance. You must specify a default PatternGroup-related WaveformTable so users may select the default definitions for all time set names within the group.

You usually create it for a new PatternGroup. All alias symbols used for all pins must be defined before they can be used in a new Pattern object. A default WaveformTable usually resolves the pattern symbols, while an exception mode lets you specify a unique exception on each WaveformTable name.

- PatternGroup WFT Reference WaveformTable—defines the table objects for each WaveformTable Reference; refer to Creating a WaveformTable Object. They are defined by the specified PatternGroup. All data in the WaveformTable defines

the format, timing, and data source for the patterns in the group. At a minimum, the data level (0/1) state of each alias character is defined.

- Zipper `WaveformTables`—are defined for each `WaveformTable` Reference and include timing and formatting. They can be referenced by one or more of the `WaveformTable` References defined in the `Pattern` object; refer to Zipper Table.

  If all other waveform data is not entered, use a stand-alone `WaveformTable` to resolve the remaining timing and waveform information: either the Zipper Table of a `PatternSequence` or the waveform cell libraries. This method is useful because `WaveformTables` may inherit data from other `WaveformTables`; thus, you can define common waveform information in one table and reuse this information in other `WaveformTables`; refer to WaveformTable Inheritance.

WaveformTable Entities

A `WaveformTable` consists of the following entities:

- Table name—either as a default `PatternGroup` `WaveformTable` or a normal table name; refer to WaveformTable Naming Rules.
- Period—Timing expression defines the global time value for the period represented by this `WaveformTable`. Typical forms: **`Per`**, **`Per+5nS`**, **`1/Freq`**, and **`100nS`**; also, refer to Period Definition.

NOTE When inheriting a WaveformTable, the period value does not get inherited; thus, it must be defined locally before performing the inheritance.

- Inheritance specification (optional)—refer to WaveformTable Inheritance.
- Cell data—refer to WaveformTable Cells.

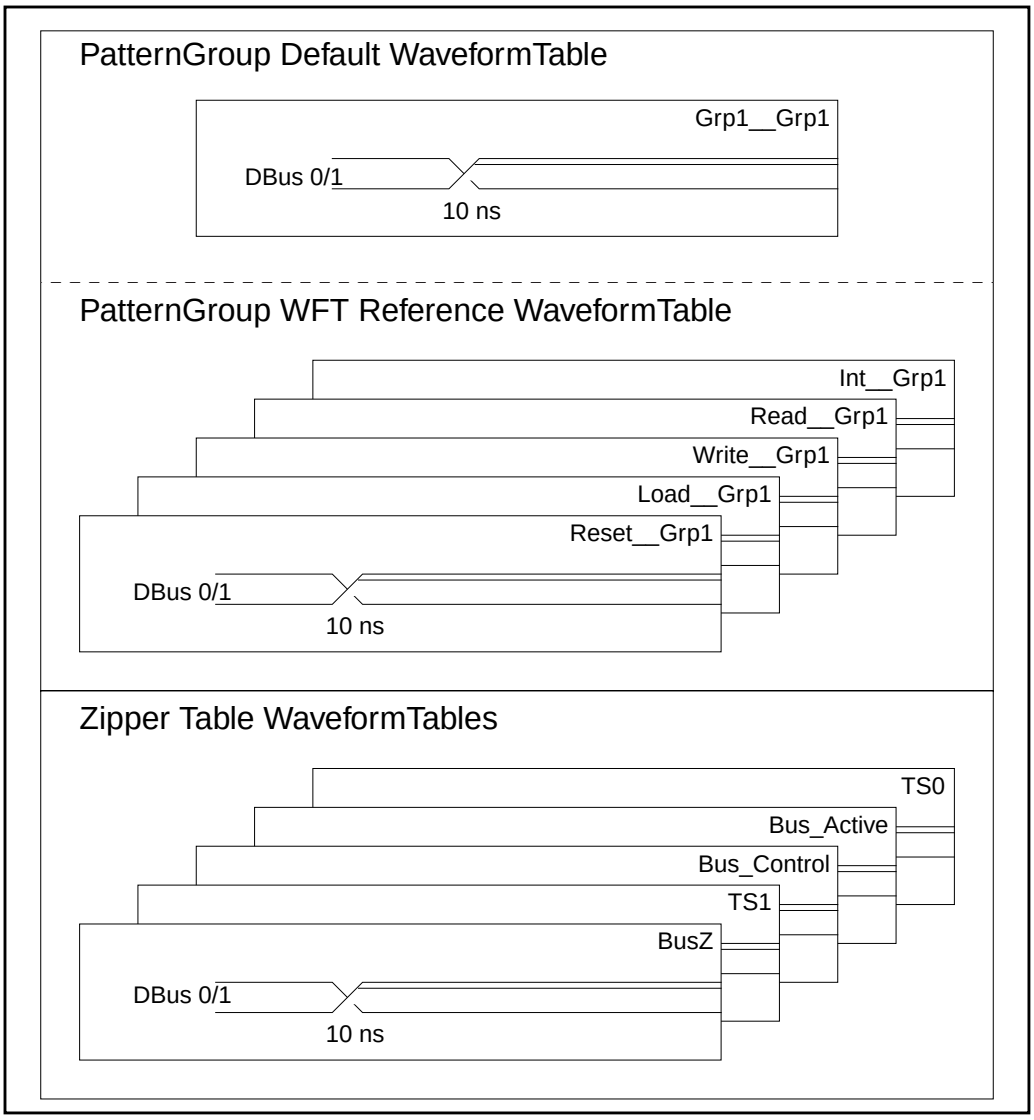


Figure 2.7: WaveformTable Types

WaveformTable Naming Rules

A `WaveformTable` name is two names concatenated by a double underline delimiter (double space delimiter also supported). Second name is always the `PatternGroup` to attach the `WaveformTable` to:

- Default `WaveformTable`

  `pat_group_name__pat_group_name`

- Exception `WaveformTable`

  `waveform_table_name__pat_group_name`

Using WaveformTool to Develop the WaveformTable Object

WaveformTool helps you develop the `WaveformTable` object and add other waveform definitions, including `WaveformTable` inheritance and `WaveformCell` inheritance.

Even though you can add waveform and timing information by manually entering the appropriate code into the .eva file, WaveTool does this for you, making your test program code must easier to read and understand, as well as saving time developing a test program.

The remaining information in this section includes a WaveformTool tutorial and a brief description of certain aspects of WaveformTool; refer to WaveformTable Cells and see Figure 2.8. For more information about this tool, refer to the *WaveformTool* chapter in the *Tools* manual.

WaveformTable Cells

Each `WaveformTable` consists of cells, which correspond to the rows in the WaveTool view of the waveforms; refer to Figure 2.9. A cell specifies sets of signal names, data, and alias characters:

- Timing Display
- Period Definition
- Device Pin Definition
- Cell Naming Convention
- Signal Direction and Alias Symbols
- Binary Data Characters
- MuxMode and MultiState Cells

You can use the mouse buttons to quickly accomplish certain WaveTool cell operations; refer to WaveformTool Mouse Conventions.

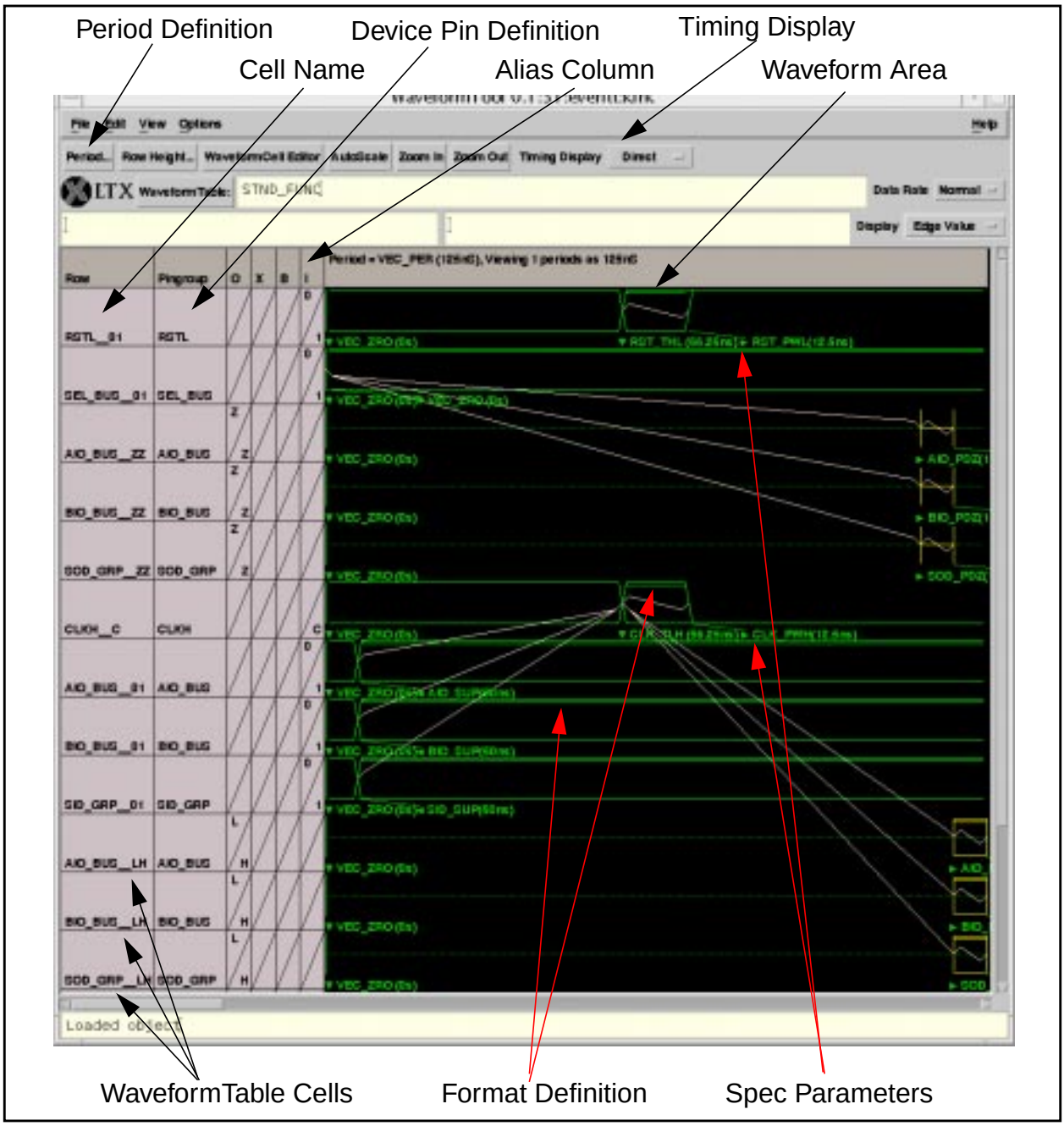


Figure 2.8: Waveform Tool, Main Window

Timing Display

WaveformTool displays timing relations in one of three ways:

- **None**—not display any lines between timing relationships.

- **Direct—**displays the point-to-point connections for timing relationships. When the *from* edge is not visible a short horizontal line is displayed.
- **DataBook—**displays the relationhips on individual rows by using horizontal and vertical lines.

Period Definition

Clicking Period displays a dialog box for entering the `WaveformTable` cycle time. If a `WaveformTable` does not have a defined period, the timing of the displayed waveforms may be inaccurate.

NOTE When inheriting a WaveformTable, the period value does not get inherited, hence it must be defined locally before performing the inheritance.

Device Pin Definition

The Pingroup field may contain one or more pin names or `PinGroup` names. You can name these pins by using the *Add Row* feature. You can also re-program the pin by using the mouse button M3 and selecting one or more pins or `PinGroup` names from the enVision Pin Finder popup; refer to Cell Naming Convention.

Cell Naming Convention

Each waveform cell is identified by its cell name (also known as *row name*). When you create a row, use the following naming convention:

```
<pin or pingroup name>__<alias_symbols[alias_symbol]>
```

Examples: **`Clk__cC`** and **`DATA_IN__01`**

where **`cC`** and **`01`** are alias symbols.

WaveformTool Mouse Conventions

Certain WaveformTool operations can accomplished by clicking a particular mouse button on a cell column. For a list of these operations, refer to the *Wavetool* chapter in the *Tools* manual. WaveformTool use the following mouse conventions:

- Left mouse button—M1

- Center mouse button—M2
- Right mouse button—M3

Signal Direction and Alias Symbols

WaveTool signal direction field consists of four columns—**O**utput, **X**care, **B**i-directional, and **I**nput—that define the direction of the pins or `PinGroups` and define the alias symbols representing the data, timing, and formatting.

Alias symbols are alphanumeric characters, case sensitive. An alias symbol defined in the upper triangle represents data logic 0, and an alias symbol defined in the lower triangle represents data logic 1; refer to Figure 2.9 and Binary Data Characters.

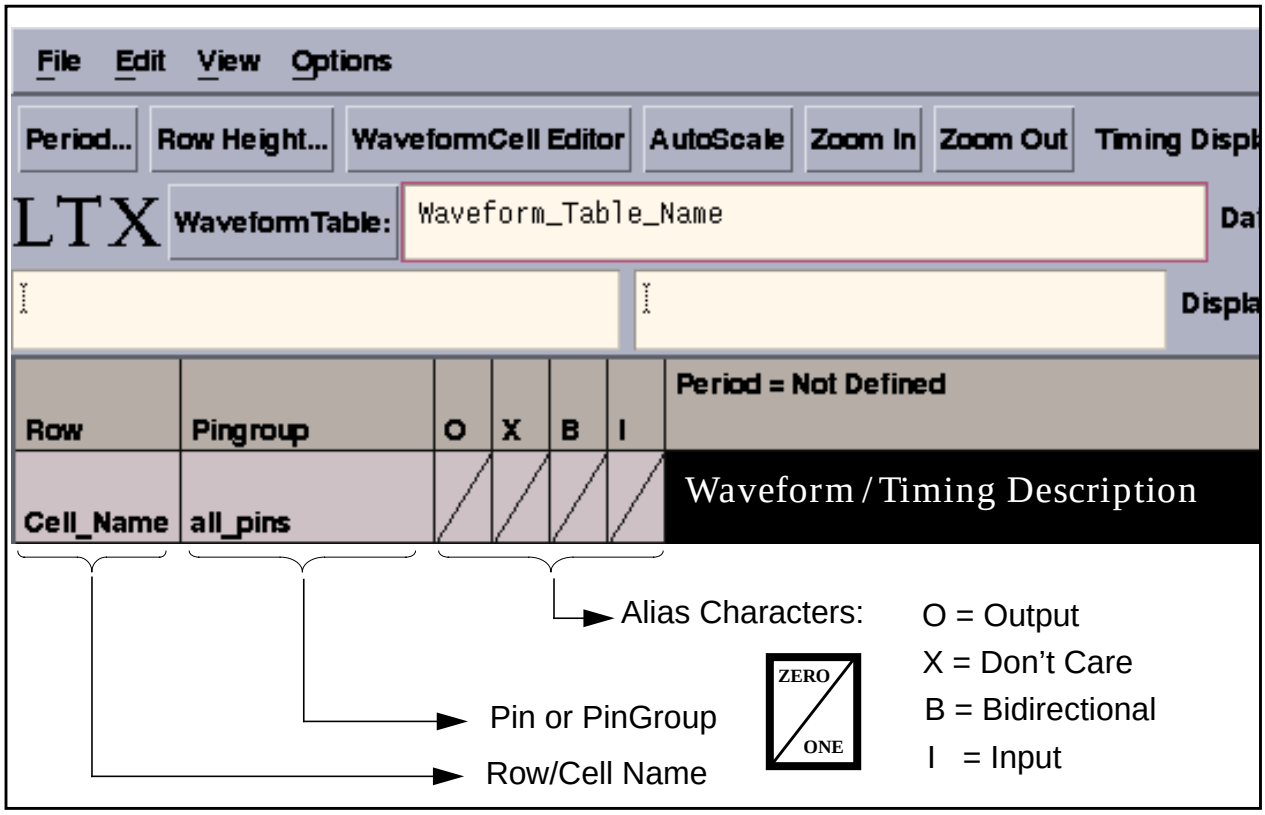


Figure 2.9: Defining Data in WaveformTable Cells in WaveTool

Binary Data Characters

Binary data and alias characters are associated by their list order: **L/H** means the L alias represents 0 data, and the H alias is 1 data. enVision encodes the data into a single number from 0 to 7. These numbers are defined in Table 2.6.

Table 2.6: Alias Data Definition

Value	Usage
0	Output (drive off, compare care), data = 0
1	Output (drive off, compare care), data = 1
2	Mask (drive off, compare mask), data = 0
3	Mask (drive off, compare mask), data = 1
4	Both (drive on, compare care), data = 0
5	Both (drive on, compare care), data = 1
6	Input (drive on, compare mask), data = 0
7	Input (drive on, compare mask), data = 1

Waveform and Timing Definitions

The Waveform area in Figure 2.8 displays the driver format, drive data format and comparator format of a waveform; refer to Formatting the Waveform. Each transition edge of a waveform has a define timing value, which can be programmed as a:

- timing offset
- timing spec parameter
- expression of a timing spec parameter and a timing offset.

MuxMode and MultiState Cells

In this enVision software release, MuxMode and MultiState vectors are supported. These cells are assigned per-pin time sets and the proper timing so the patterns in the PatternSequence object are executed. For MuxMode and MultiState operation, waveform cells are combined according to the mode of operation: MultiState, MuxMode, or MuxMode / MultiState.

Waveforms for MultiState Operation

For example, assume a `PatternMode` X3, which is a 3-state pattern with 3 separate and unique user cells, each with a unique group identifier. In this mode, a waveform cell is created that combines these three cells. The new cell contains all events of the 3 user cells represented by the unique group. The timing for the second and third cells is offset by the periods of the previous cells. For example, assume the following definitions:

- Cells 0 and 1 in a `WaveformTable` whose period is 4 ns.
- Cell 1 drives high at 1.5 ns
- Cell 0 drives low at 3 ns

Thus, the new cell representing these 3 cells would have the following events and timing:

- TG1 Drive High @ 1.5 ns
- TG2 Drive Low @ 7 ns
- TG3 Drive High @ 9.5 ns
- Period is 12 ns.

Waveforms for MuxMode Operation

In this mode, a list of even and odd pin pairs in a `MuxMode` pattern is used to create a waveform cell for the odd pins. Although the test program contains only the even-numbered tester channels, the new waveform cell for the odd-numbered channel includes the events of the user waveform cells, but its timing is offset by the period of the even pin.

Waveforms for MuxMode/MultiState Operation

In this combined mode, the waveform cells and multiplexed pin waveform cells are created by using a list of odd- and even-channel pairs.

To create the period time sets for these cells, a list of all vectors in the `MultiState/MuxMode` pattern is used to determine the periods required for the pattern. The cells are then assigned the period time sets, which are reported to the `PatternGroup` object.

Formatting the Waveform

Formatting shapes the waveforms sent to the DUT. It describes the type of waveform: Drive, Compare, and Expect fields specify the mask applied to the waveform from the pattern memory:

- Drive—describes the input drive waveform the tester is generating to the DUT; refer to Driver Data Formats and Standard Driver I/O Formats.
- Compare—describes the pass/fail compare strobing of the DUT when the tester driver is turned off and the device is driving; refer to Compare Events.

The enVision drive and compare symbology is different than that for a conventional device; refer to Drive Event Symbols and Compare Data Symbology.

Figure 2.10 shows an example of the driver events needed to create a waveform.

Driver Data Symbology

enVision uses a different drive data symbology than that documented for conventional devices; see Figure 2.11 and Figure 2.12.

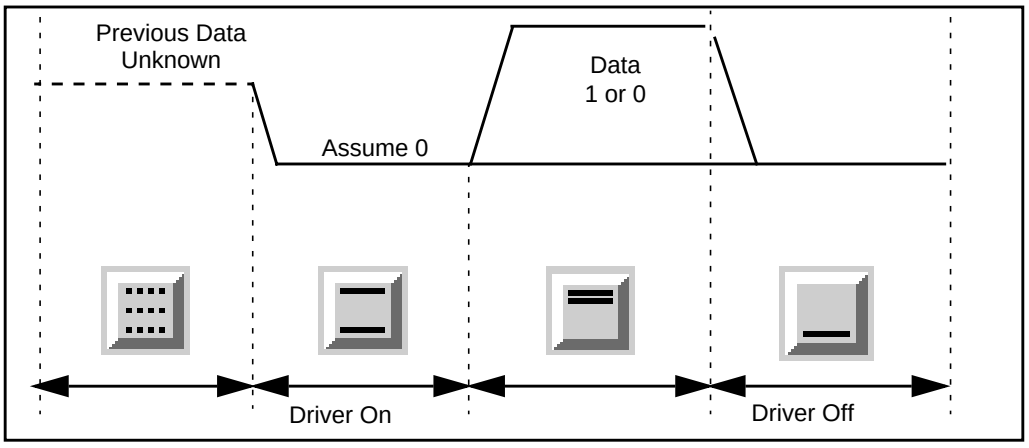


Figure 2.10: Driver Events to Create a Waveform

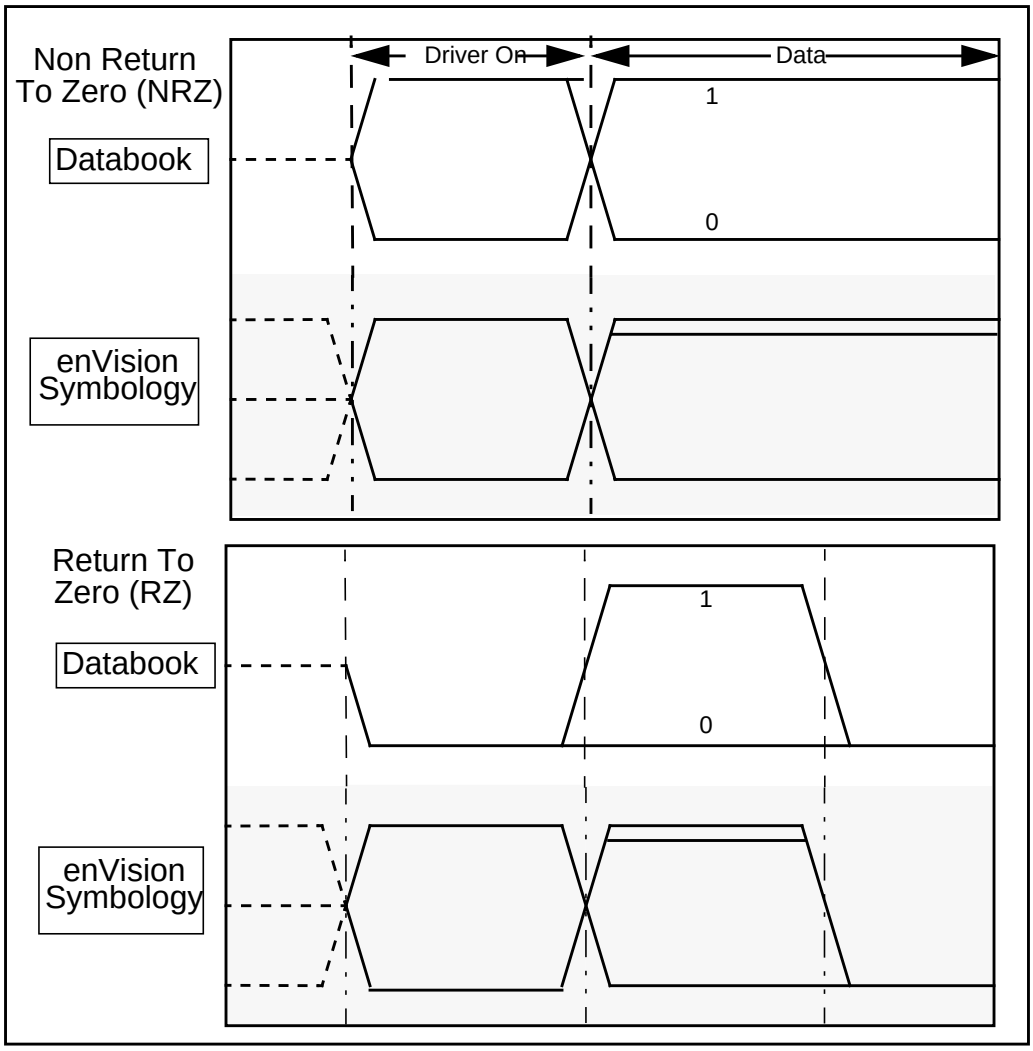


Figure 2.11: enVision NRZ and RZ Formats

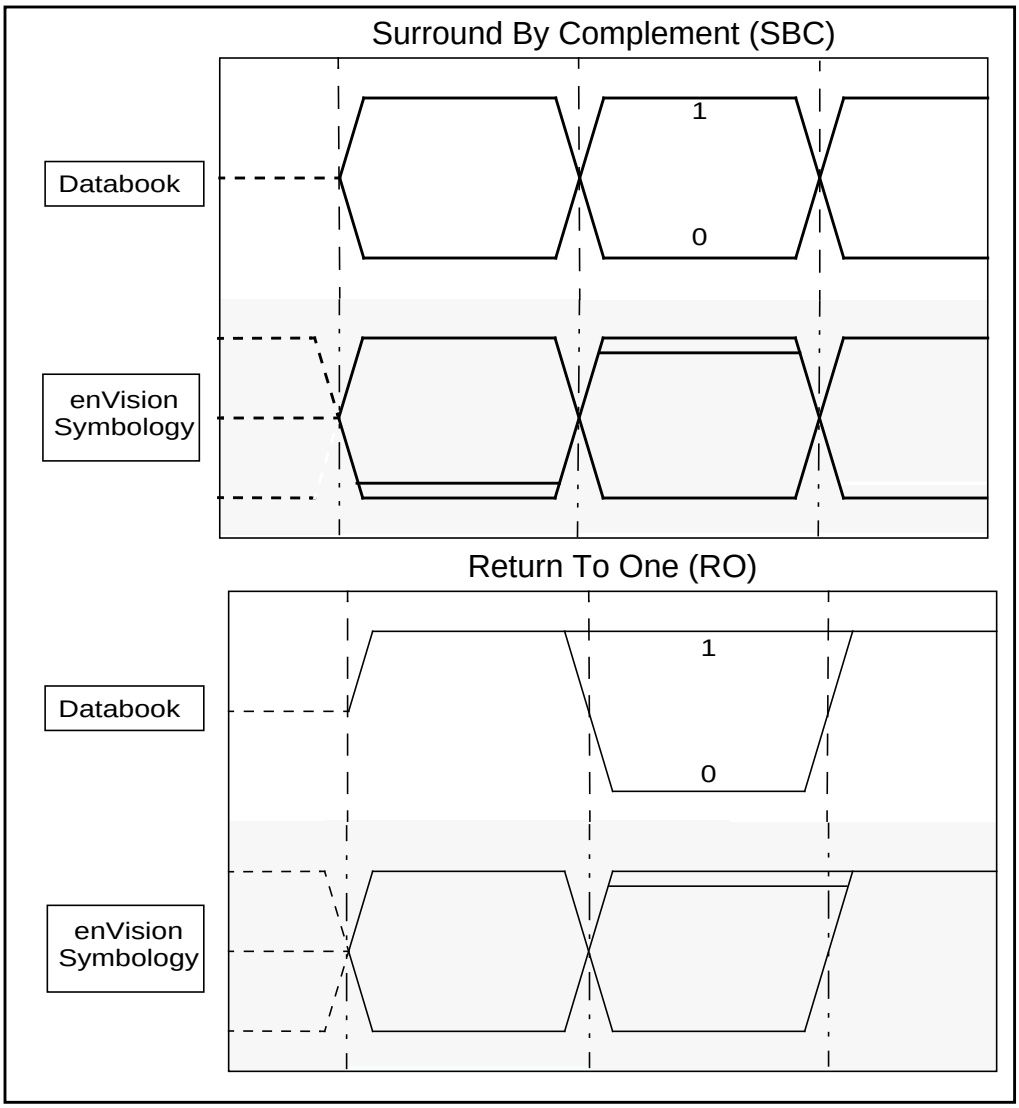


Figure 2.12: enVision SBC and RO Formats

Standard Driver I/O Formats

enVision supports the following I/O formats; also, refer to Figure 2.13.

- OFF—ON

  Turns the driver from off to on, leaving it on until the end of the test cycle. Requires only one timing edge to be programmed.

  Purpose: Programming pure input pins or pins that are tested at the beginning of the cycle and drive data in the second half of the cycle.

- ON—OFF

  Turns the driver from on to off, leaving it off until the end of the test cycle. Requires only one timing edge to be programmed.

  Purpose: programming pure output pins or pins that drive data at the beginning of the cycle and are tested in the second half of the cycle.

- OFF—ON—OFF

  Turns the driver from off to on to drive data and turns the driver off until the end of the test cycle. Requires two timing edges to be programmed.

  Purpose: bi-directional pins requiring testing, driving data, followed by testing, if needed.

- ON—OFF—ON

  Turns the driver from on to off for testing and turns the driver back on until the end of the test cycle for data driving. Requires two timing edges to be programmed.

  Purpose: bi-directional pins that drive data, then testing, followed by driving data, if needed.

- OFF

  Requires one timing edge to turn the driver off, allowing for testing.

  Purpose: pure output pins.

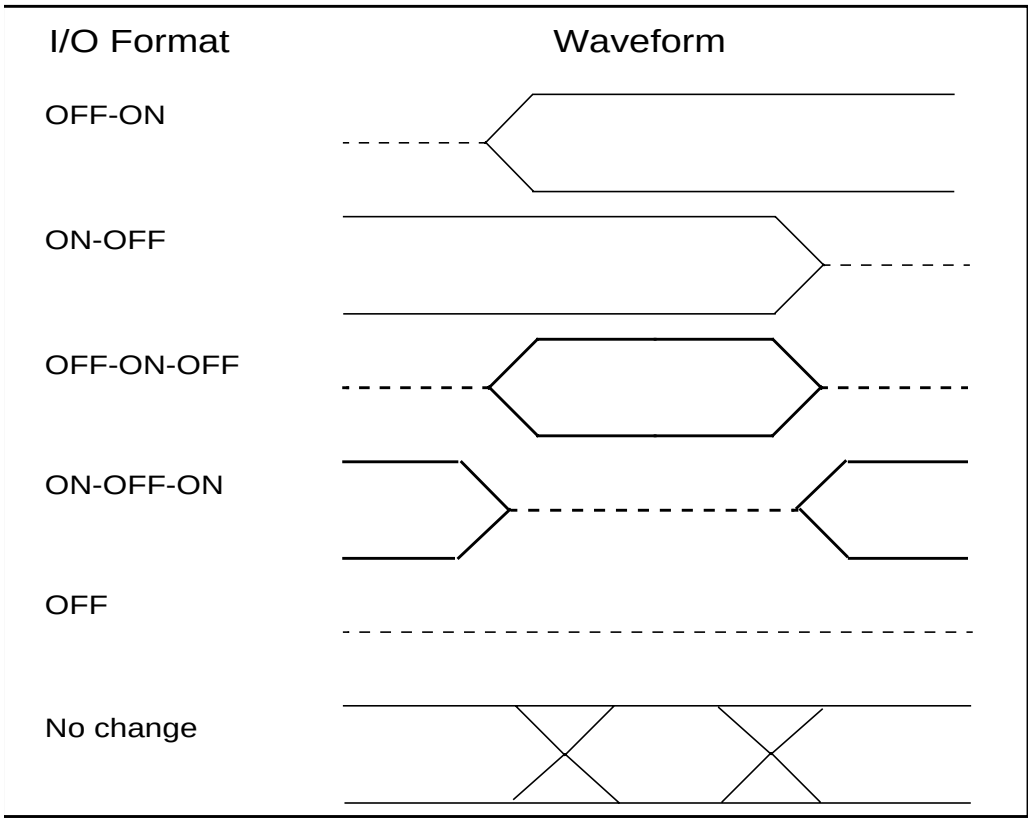


Figure 2.13: Standard Driver I/O Formats

Driver Data Formats

enVision uses a drive data symbology that is different than that used by most device manufacturers; refer to Driver Data Symbology.

enVision has six driver data formats; see Figure 2.14**:**

- NRZ—Non Return to Zero

  Only one timing edge is needed. At most one transition edge exists if the present and the previous cycles have different data. No transition edge if the data in the present and the previous cycles is the same. Purpose: data or address pins.

- RZ—Return to Zero

  Two timing edges must be defined. If the data in the present cycle is a 1, two transitions by two timing edges form a positive pulse. Or, if the data in the present cycle is a 0, data is low for the entire cycle: no transitions. Purpose: clock or strobe pins.

- RO—Return to One

  Two timing edges must be defined. If the data in the present cycle is a 0, two transitions by two timing edges form a negative pulse. Or, if the data in the present cycle is a 1, data is high for the entire cycle: no transition. Purpose: negative clock or enable pins.

- SBC—Surround By Complement

  Data is surrounded by its complement requiring a minimum of two timing edges: one for the transition from the complement to data and another for the transition from the data to its complement. Purpose: data pins needing setup time and hold time.

  An additional timing edge is required for starting the complement of a cycle if the data in the present and previous cycles is different. This timing edge for starting the complement is the same as the timing edge to turn on the driver for the tester.

- RTC—Return To Complement

  Two timing edges if the data in the present and previous cycles is the same. At least one timing edge to transit the data to its complement. Purpose: data pins needing hold time.

- PBC—Precede By Complement

  At least one timing edge to transit from the complement data to the data.

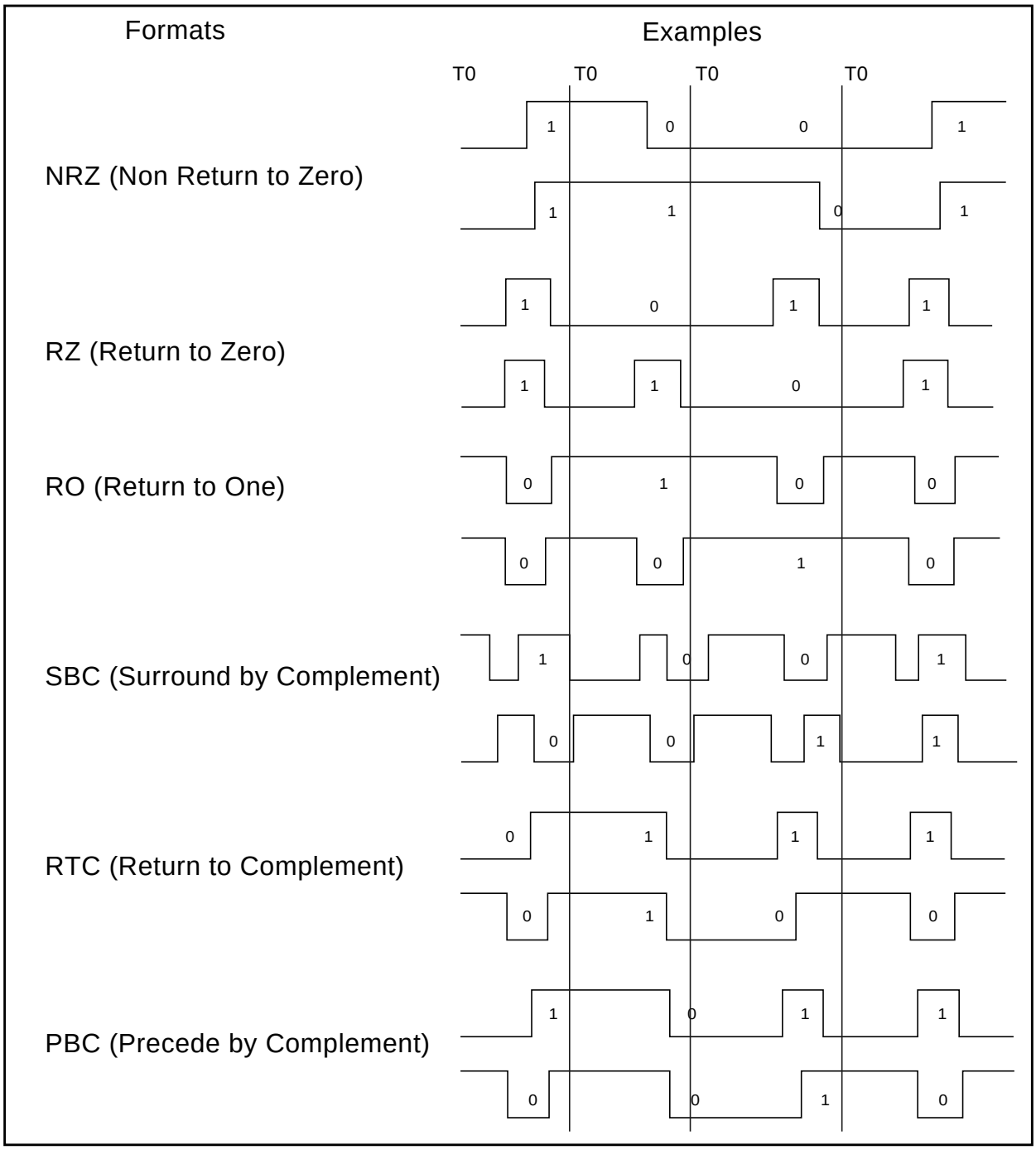


Figure 2.14: Standard Driver Data Formats

Drive Event Symbols

 Unknown Previous (driver)

 Drive High if One (data). `DriveHighIfOne`.

 Drive Low if Zero (data). `DriveLowIfZero`.

 Driver Off (driver). `DriveOff`

 Driver On (driver). `DriveOn`

 Drive High (data). `DriveHigh`

 Drive Low (data). `DriveLow`

 Drive Data Not State (complement data). `DriveDataNot`.

 Drive Data State (data)

Compare Events

Each event is specified with a certain timing value. Several forms of event marker timing can be specified; refer to Timing. The syntax is a written representation of the timing as it appears in WaveformTool.

enVision uses a different compare data symbology that is different than that used by most device manufacturers; see Figure 2.15 and Figure 2.16.

For example, two compare events: Compare High and Float Edge, are needed to test a waveform; see Figure 2.17.

Compare Data Symbology

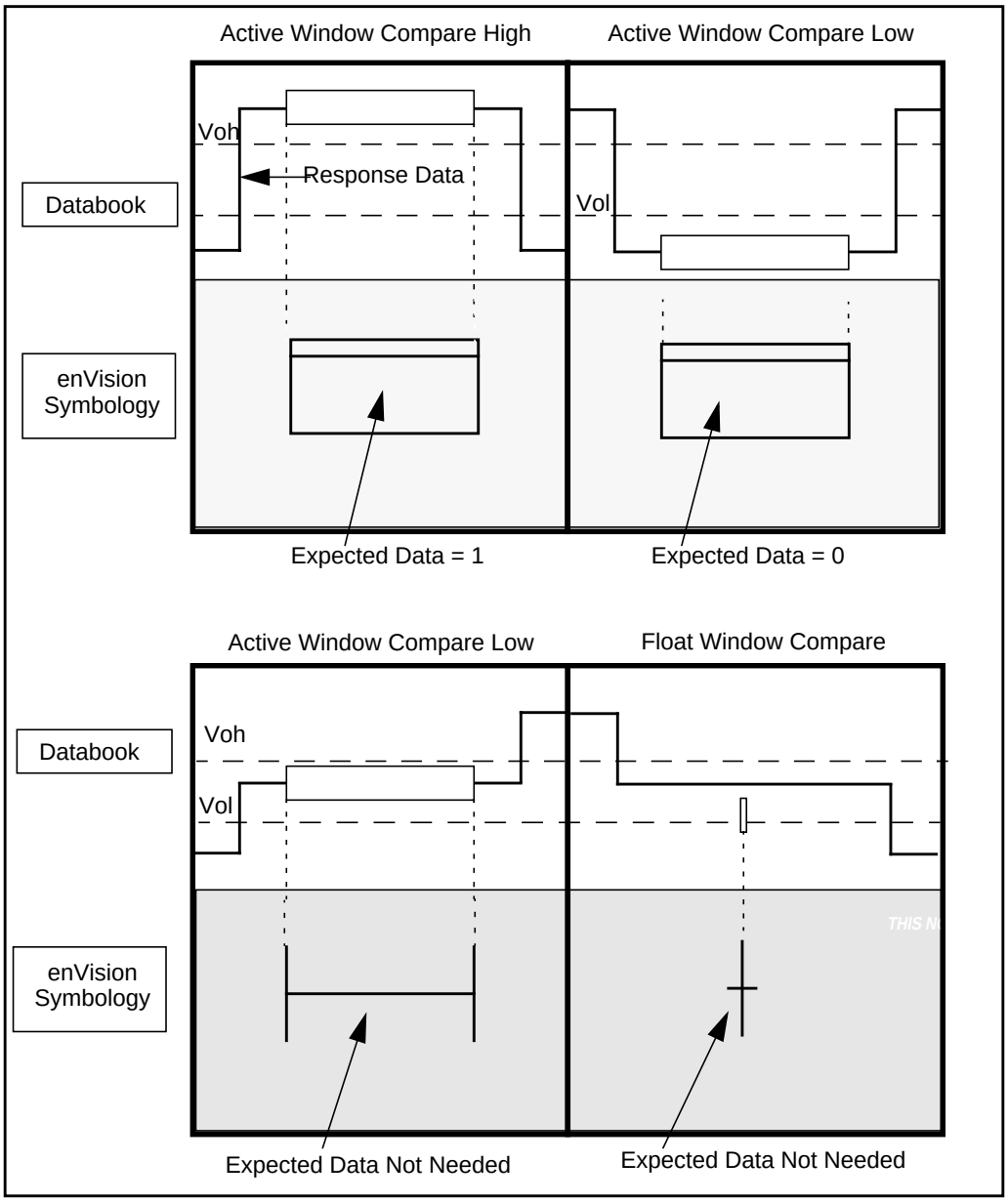


Figure 2.15: enVision Compare Symbology, Active Windows and Float Window

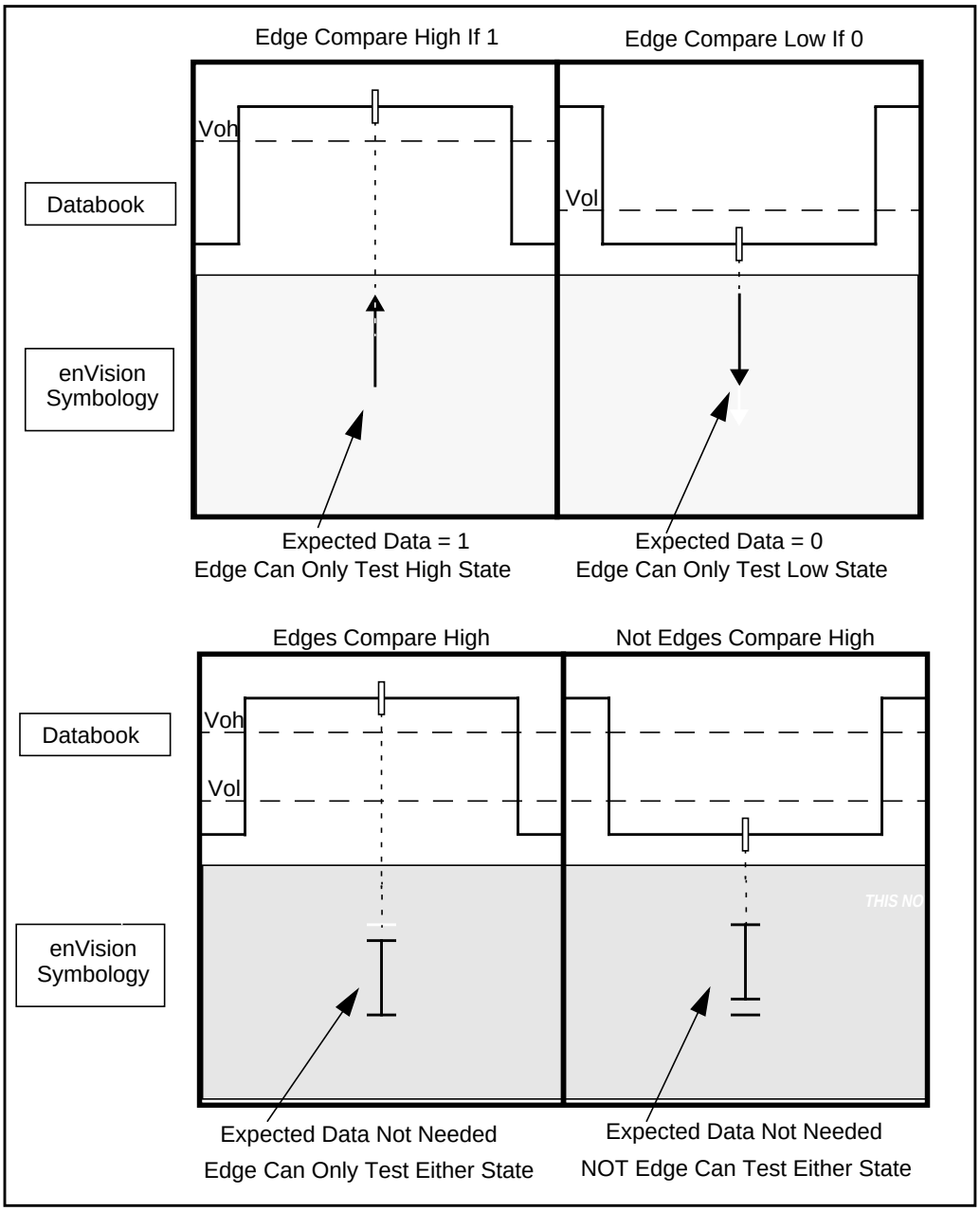


Figure 2.16: enVision Compare Symbology, Edge Compare and Not Edge Compares

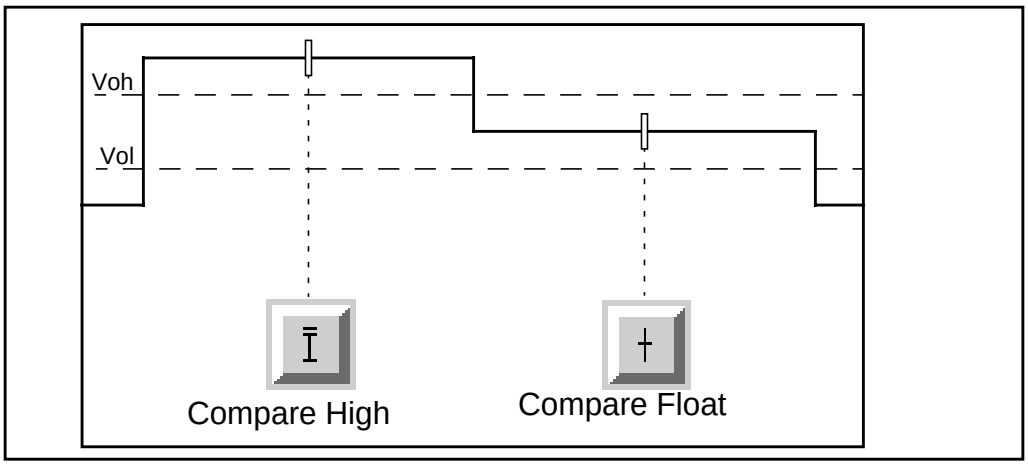


Figure 2.17: Compare Events to Test Waveform

enVision Compare Events

enVision uses a compare data symbology that is different than that used by most device manufacturers; refer to Compare Data Symbology.

 No State Marker (Place Holder)

 Edge Compare Low (unqualified). `CompareLowIfOne`

 Edge Compare High (unqualified). `CompareHighIfOne,CompareHighIfZero.`

 Open Window Compare Low if Zero. `CompareLowIfZero`

 Open Window Compare High if One

 Close Any Window. `CompareClose`

 Open Float Window. `CompareOpenFloat, CompareOpenFloatNo`

 Open Active Window

 Float Edge. `CompareFloat, CompareFloatNot`

 Edge Compare High if One

 Edge Compare Low if Zero

 Edges Compare

 Period Marker

WaveformTable Inheritance

Inheritance allows one `WaveformTable` to derive its information from another `WaveformTable`; refer to Specifying the Forms of Inheritance. The benefit of this feature is reducing the amount of information you must create or re-enter rather than reusing existing information. For example, if a set of timing marker values is common to several different waveforms, you need only define these values once in a base `WaveformTable` so other `WaveformTables` or cells can inherit them.

You can define inheritance by using the Waveform Cell Tool; refer to Using WaveformCell Tool to Define Inheritance and Inheritance Rules for Waveform Tables.

Specifying the Forms of Inheritance

Inheritance code begins with the `Inherit` keyword, followed by either a `WaveformTable` name (Table inheritance), a cell name (local cell inheritance), or the combination of the two (other table cell inheritance), separated by a period (.):

- `WaveformTable` inheritance—derives all information from the named waveform table. Do inheritance before defining any cells. Any waveform definitions in the table cells override the inherited information.

- Local cell inheritance is from another cell in the current waveform table, and must occur within a cell definition. If data from more than one cell is inherited, the first inheritance that defines data for a section will be used; refer to Data Inheritance by Section.

- Cell inheritance from another `WaveformTable`—references a cell in another `WaveformTable`; otherwise, it has the same properties as local cell inheritance.

Data Inheritance by Section

Data is inherited section by section; thus, if anything is defined locally in a section, all data for that section must be defined locally. Sections not defined locally can use inherited information. enVision has the following sections:

- Drive Format
- Drive Timing
- Drive Control
- Compare Format
- Compare Timing
- Compare Control
- Expect Format
- Expect Timing
- Expect Control
- Pattern Source
- Calibrate

Inheritance Rules for Waveform Tables

Be sure to follow the inheritance rules for Data, Drive Enable, and Compare Enable:

- Use the `WaveformTable` whose name agrees with the name of the un-zipped `WaveformTable` Reference on the vector.

- Use the `WaveformTable` with the same name as the group name.
- Default states

Be sure to follow the inheritance rules for cells:

- First cell inheritance resolves a given section.
- First `WaveformTable` inheritance resolves a given section
- Group `WaveformTable`
- Default states
- Inherited waveformcells are not directly editable.
- Inherited waveformcells show the referenced waveformcell's *full name.WFT.RowName*.

Using WaveformCell Tool to Define Inheritance

The WaveformCell Tool (WFCellTool) lets you create, view, and edit the `WaveformTable` objects of a single cell; see Figure 2.18. It is especially useful when defining `WaveformTable` inheritance and `WaveformCell` inheritance:

This section presents a brief WFCellTool tutorial:

- Launching WFCellTool
- Loading a Waveform Cell into WaveformCell Tool
- Displaying WaveformCell Information

For more information about this tool, refer to the *Waveform Cell Editor Tool* chapter in the *Tools* manual.

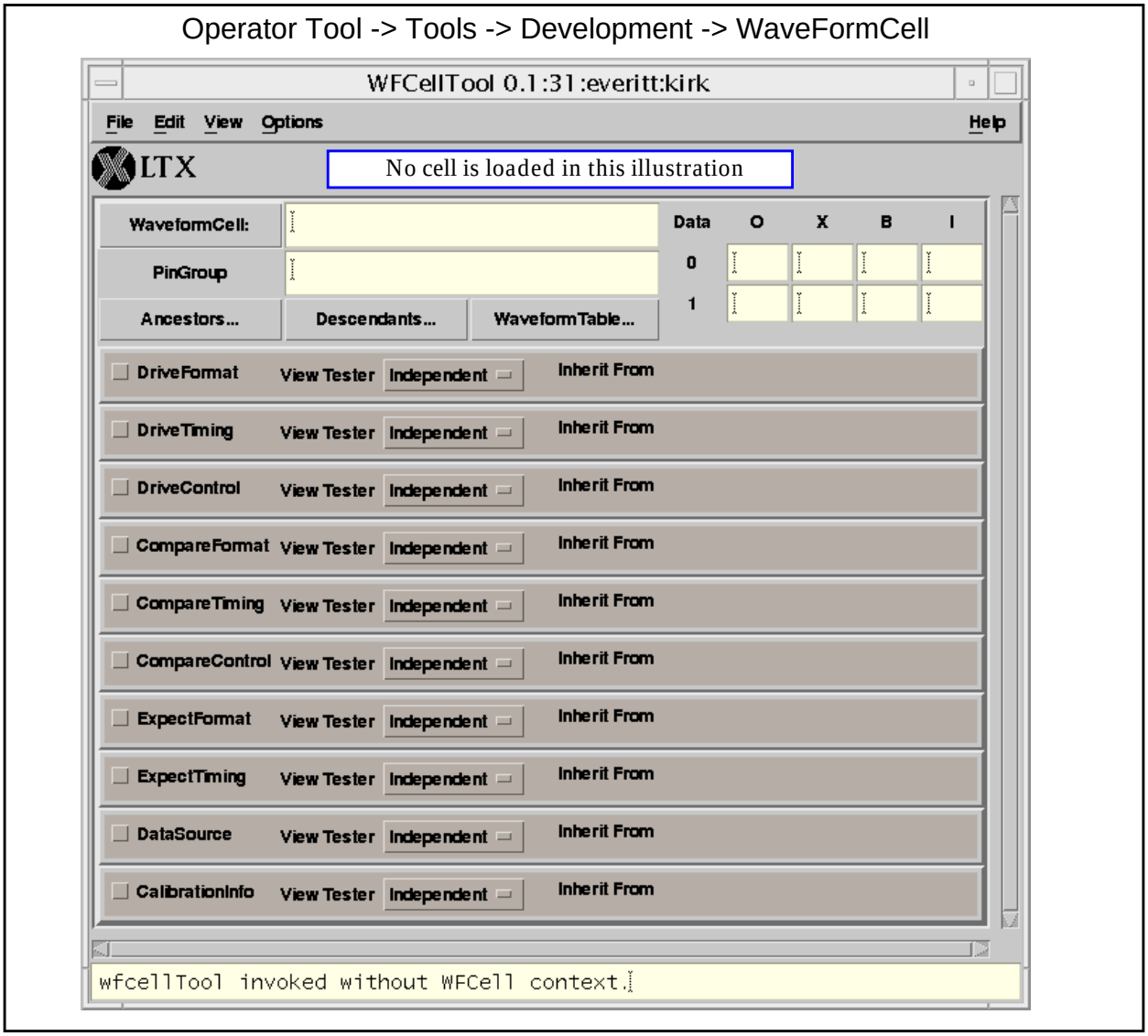


Figure 2.18: WaveformCell Tool

Launching WFCellTool

You have three options for launching this tool:

- Operator Tool -> Tools -> Development -> WaveFormCell
- In WaveformTool, place the cursor in the desired Row field of the cell. Double click mouse button 1 (M1). WFCell Tool is spawned with the selected cell loaded.
- WFCellTool may be spawned from the WaveformTool by clicking M1 on the WaveformCellEditor button.

Loading a Waveform Cell into WaveformCell Tool

1. In WFCellTool, click M1 on the WaveformCell: button to open Find a WaveformCell window; see Figure 2.19.

2. In WFCellTool window:

   a. Click on the required `WaveformTable` name.

   b. Use M1 to select the required cell name and double click on it. Alternatively, click on cell name and click *OK* button.

   Selected cell is loaded into WFCell Tool.

Displaying WaveformCell Information

The following examples assumes the (1)

- Drive format and timing and compare format and timing of the WaveformCell has been defined.
- WaveformCell is loaded into the WFCell Tool.

Drive Format. After clicking the DriveFormat button, a row of buttons associated with various drive format symbols appear with the defined waveform; see Figure 2.20. The waveform is formed with a combination of various drive formats. You can alter the existing waveform or re-create a waveform by using this tool.

Drive Timing. After clicking the DriveTiming button, the following information is displayed; see Figure 2.20:

- Drive timing information, such as timing edges, timing values, timing reference edges and directions, if any for the relative timing edge.
- Timing spec parameter values or expressions, depending on whether the Display mode is set to Value or Expression.
- Timing offsets, if any.

NOTE Any changes done in the WFCellTool will be reflected in the Waveform Tool immediately.

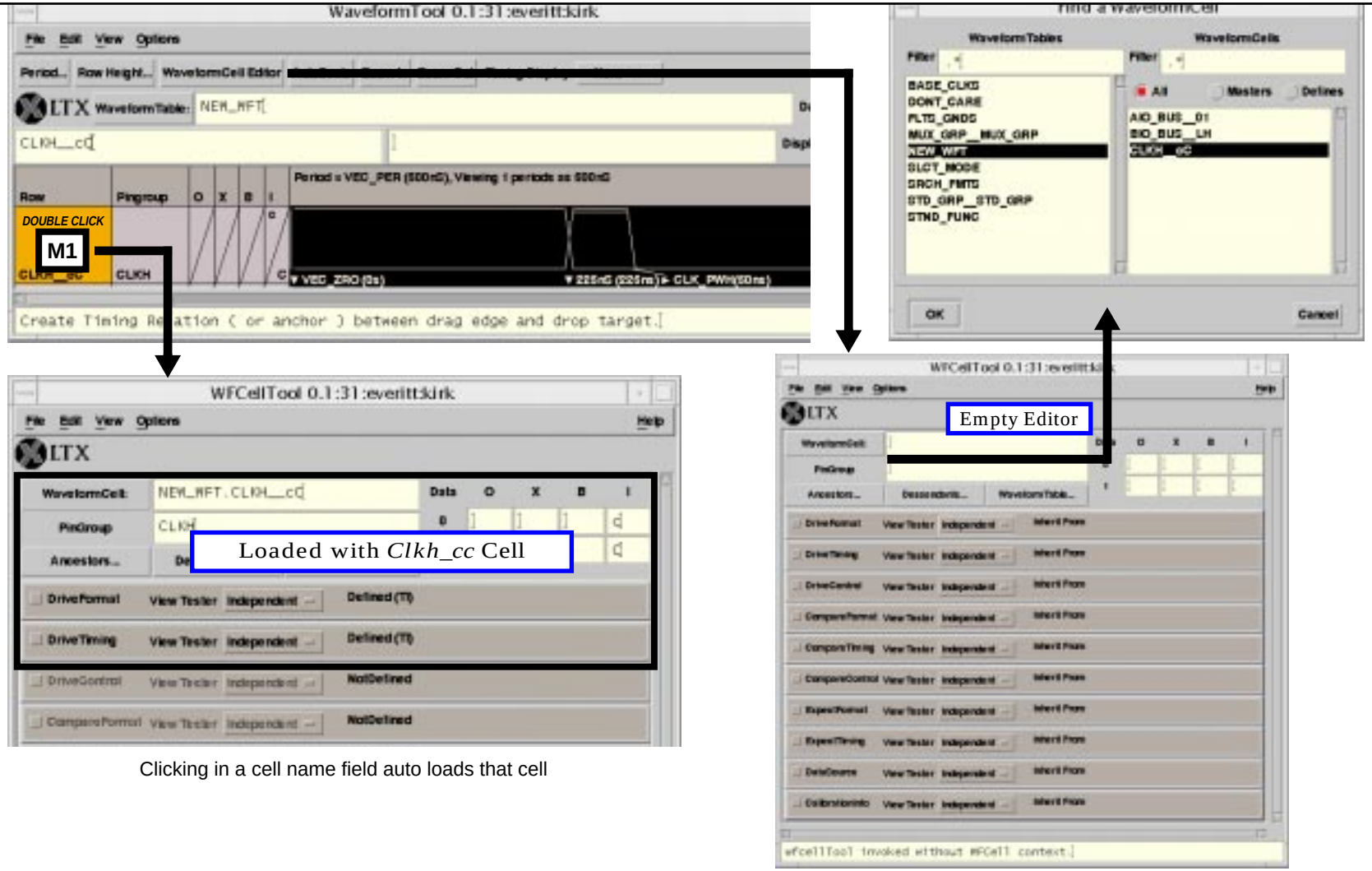


Figure 2.19: Loading a WaveformTable Cell into the WFCell Tool Editor

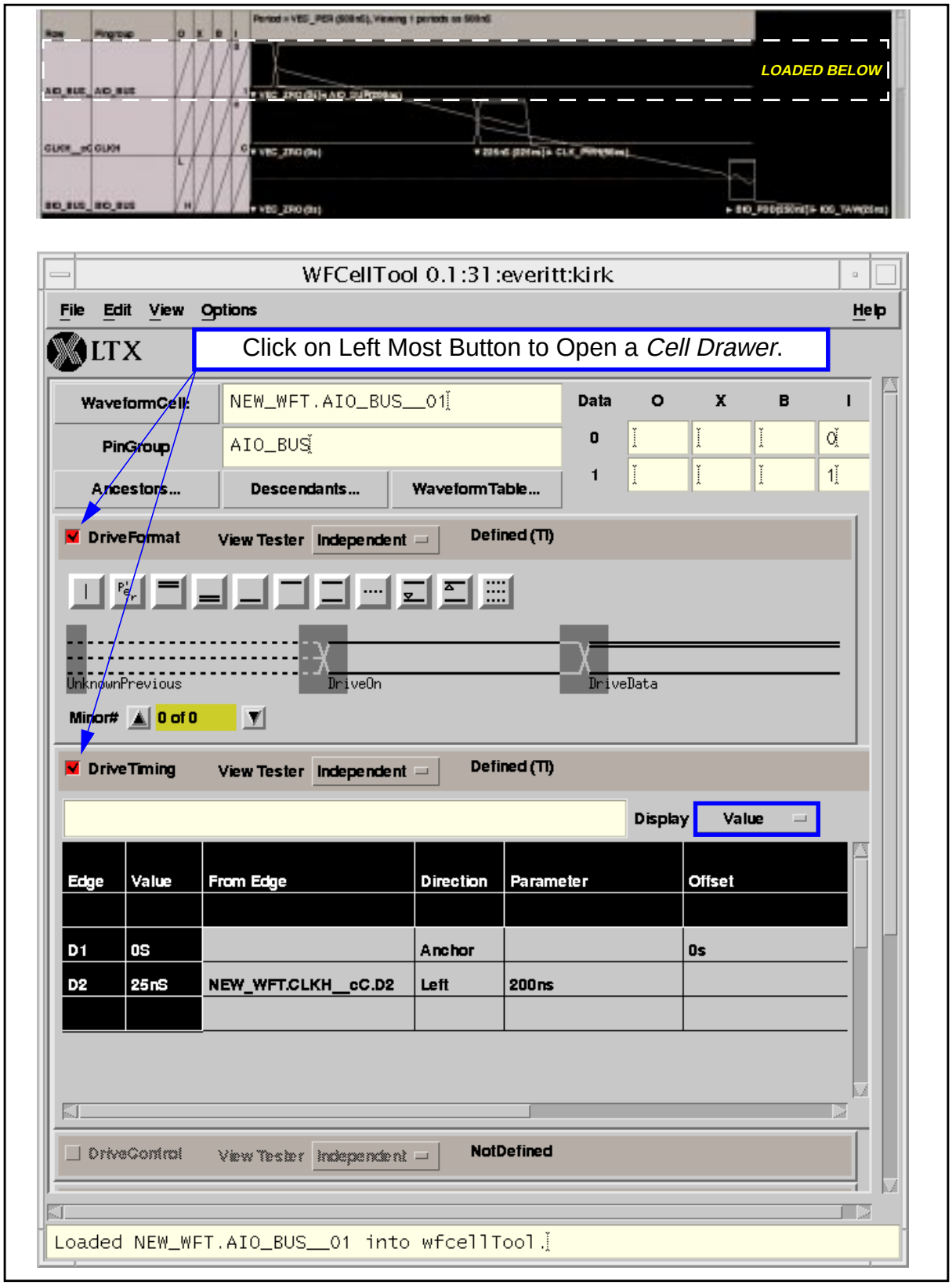


Figure 2.20: Displaying Drive Format and Timing of Waveform Cell

Compare Format. After clicking the CompareFormat button, a row of buttons associated with various compare format symbols appear with the defined waveform; see Figure 2.21. The waveform is formed with a combination of various compare formats. You can alter the existing waveform or re-create a waveform by using this tool.

This example shows the waveform cell with its driver turned off and the compare format and compare timing defined.

Compare Timing. After clicking the CompareTiming button, the following information is displayed:

- Compare timing information, such as timing edges, timing values, timing reference edges and directions, if any for the relative timing edge.
- Timing spec parameter values or expressions, depending on whether the Display mode is set to Value or Expression.
- Timing offsets, if any.

NOTE Any changes done in the WFCellTool will be reflected in the Waveform Tool immediately.

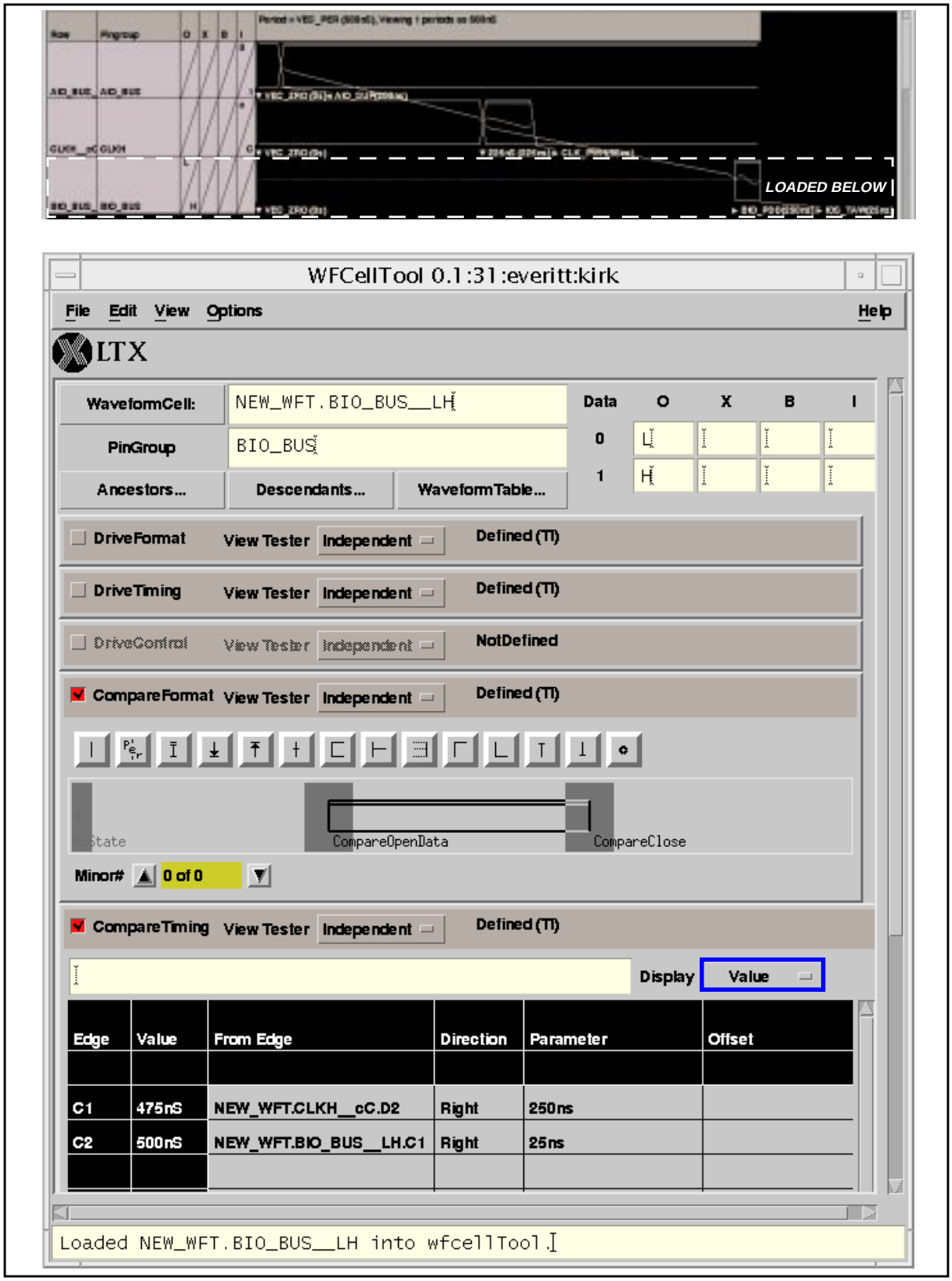


Figure 2.21: Displaying Drive Format and Timing of Waveform Cell

Inheritance Examples

WaveformTable Inheritance

To inherit all attributes (except Period) from another `WaveformTable`; see Figure 2.22:

- Create a `WaveformTable` object, program the period value, select File -> Inherit -> Ancestor WaveformTable.
- `WaveformTable` **`First`** is a *Descendent* of **`Main`**.
- `WaveformTable` **`Main`** is an *Ancestor* of **`First`**.

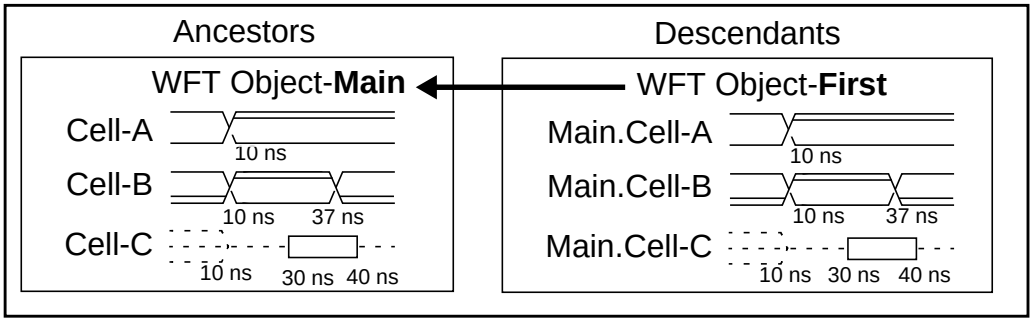


Figure 2.22: Inherit Entire WaveformTable

Cell Inheritance

To inherit all attributes (including name) from another `WaveformTable` cell; refer to Figure 2.23:

- In WFCell Tool, click M3 -> Inherit Cell in an unused (black) area, select the WaveformTable and Cell to inherit.
- You can not edit *Cell-B*. Must edit **`Main.Cell-B`** or make **`Cell-B`** local to `WaveformTable` object **`Second`** (M3 -> Make Local Cell).

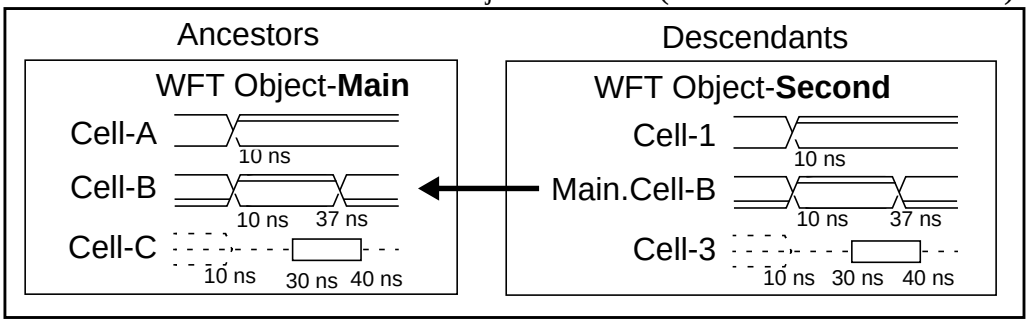


Figure 2.23: Inherit Entire Cell

Cell Content Inheritance

To inherit only the contents of a cell (timing, format, among others) from another cell or cells; refer to Figure 2.24 and Figure 2.25:

- Cell pin definition, name, direction and alias symbols are local to the `WaveformTable` object.
- In WFCellTool, select Ancestor. In the Ancestor popup, press and hold M3, select Inherit -> WaveformTable and Cell to inherit; see Figure 2.26.

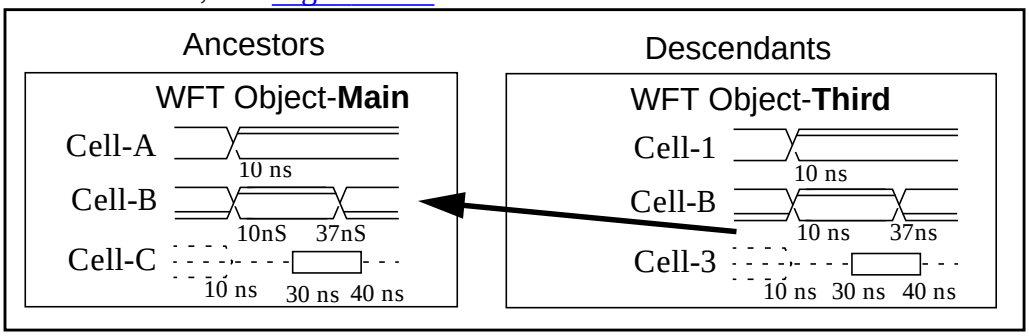


Figure 2.24: Inherit Contents of Cell

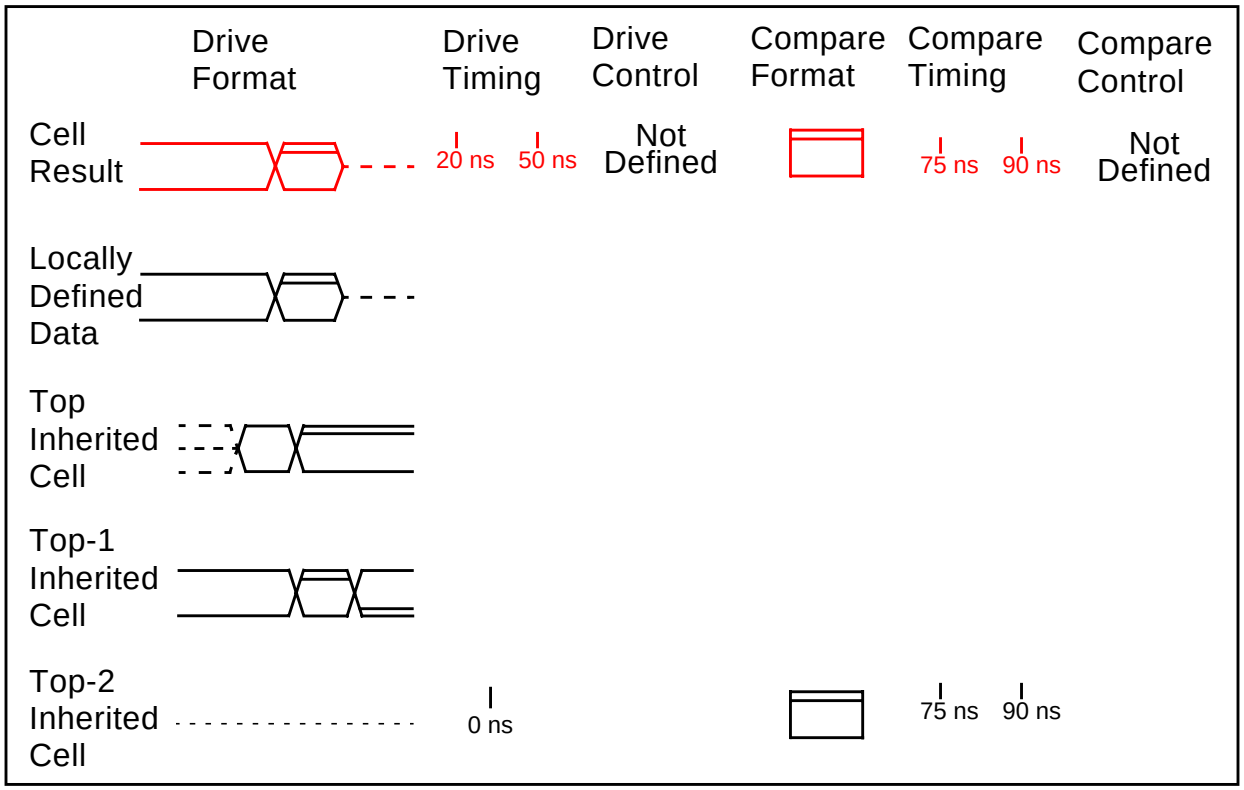


Figure 2.25: Inheriting Cell Contents

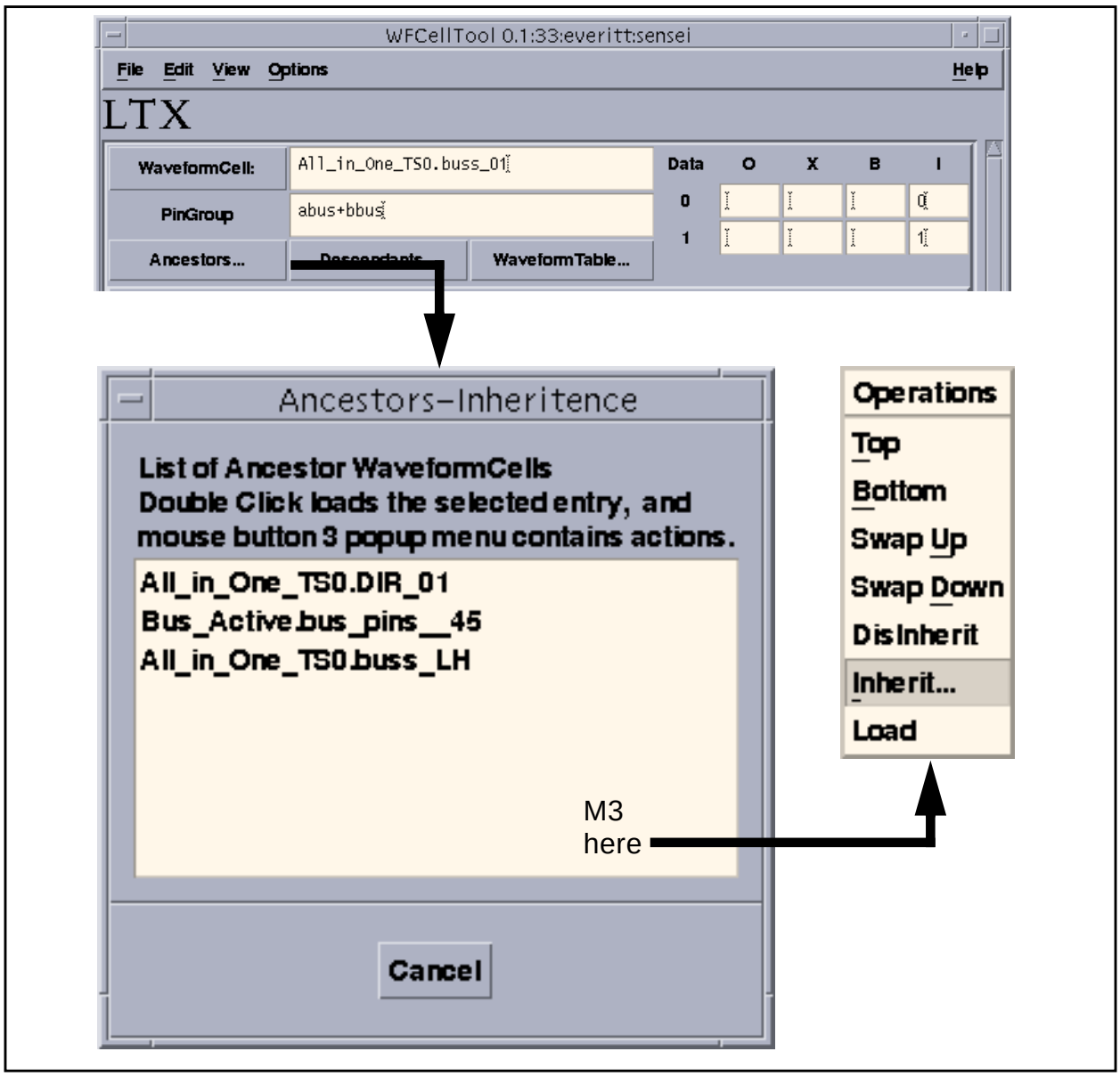


Figure 2.26: Inherit Contents of Cell Using WFCellTool

Using WFCellTool to Create Customized Drive Waveform

In the following example, the device's mode selection is handled by `PinGroup` **`SEL_GRP`** (**`SEL1`** .. **`SEL4`**). It is a completely asynchronous (no clock). However, before a mode can be selected, all four **`SEL`** lines must be driven low for a predetermined time and then driven to the desired bit combination for the mode selection (all in one vector).

The waveform format, `Precede By Zero`, for this operation is not available in WaveformTool; however, you can create this format in WFCellTool by combining **`Driver On`** and **`OnLow`**; see Figure 2.27 and Figure 2.28.

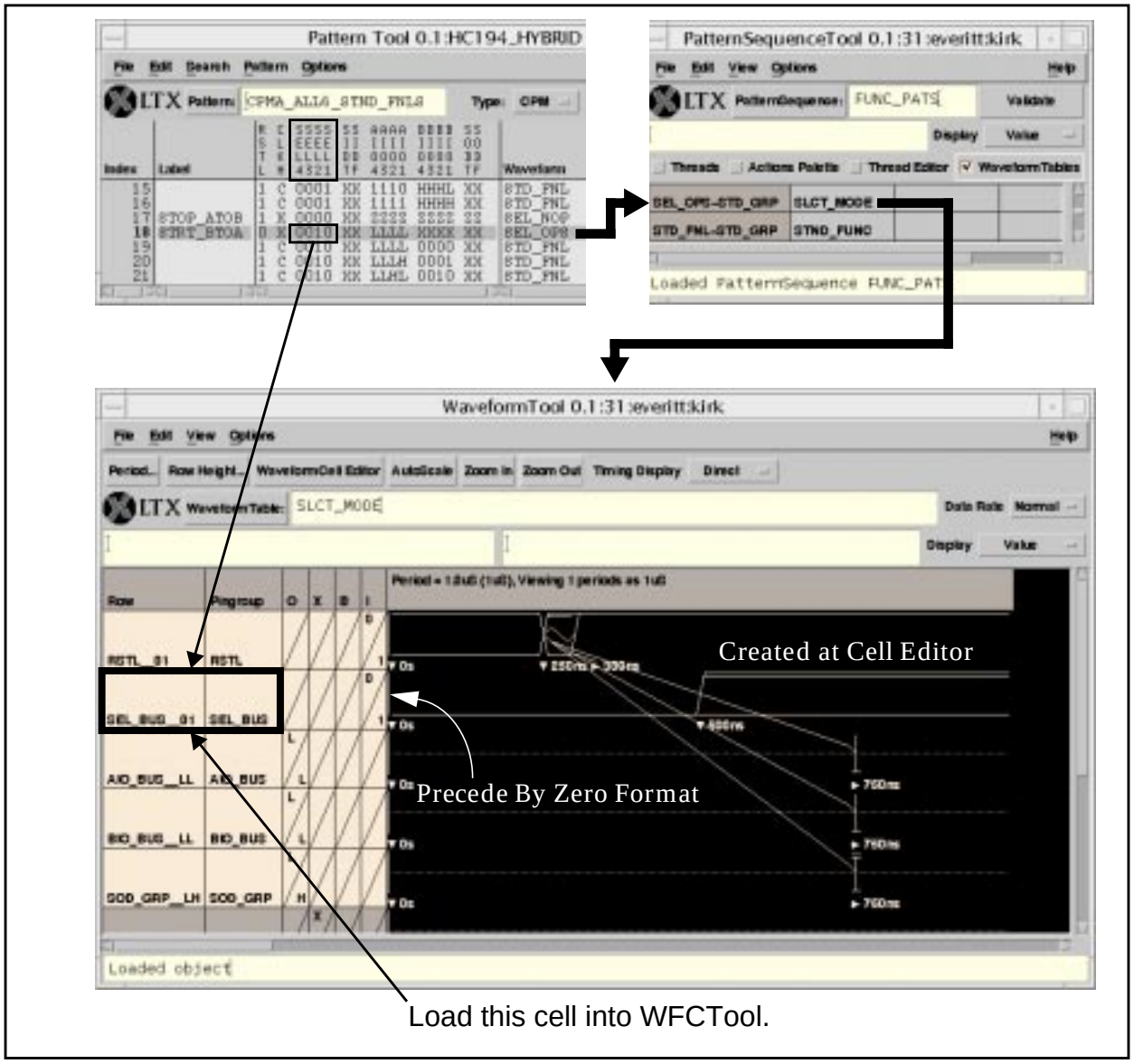
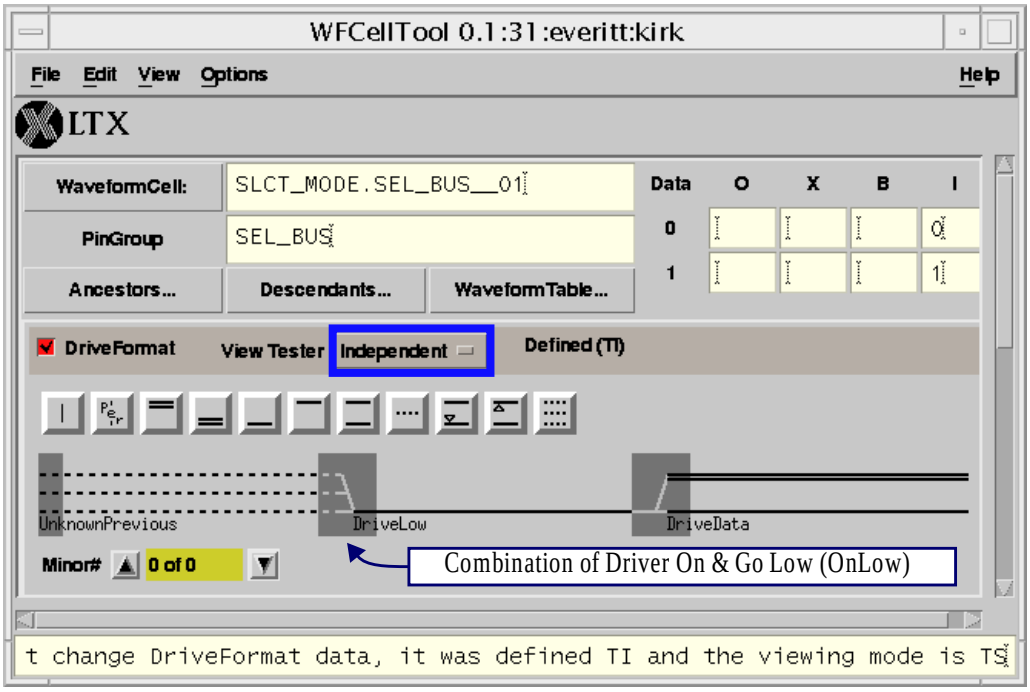
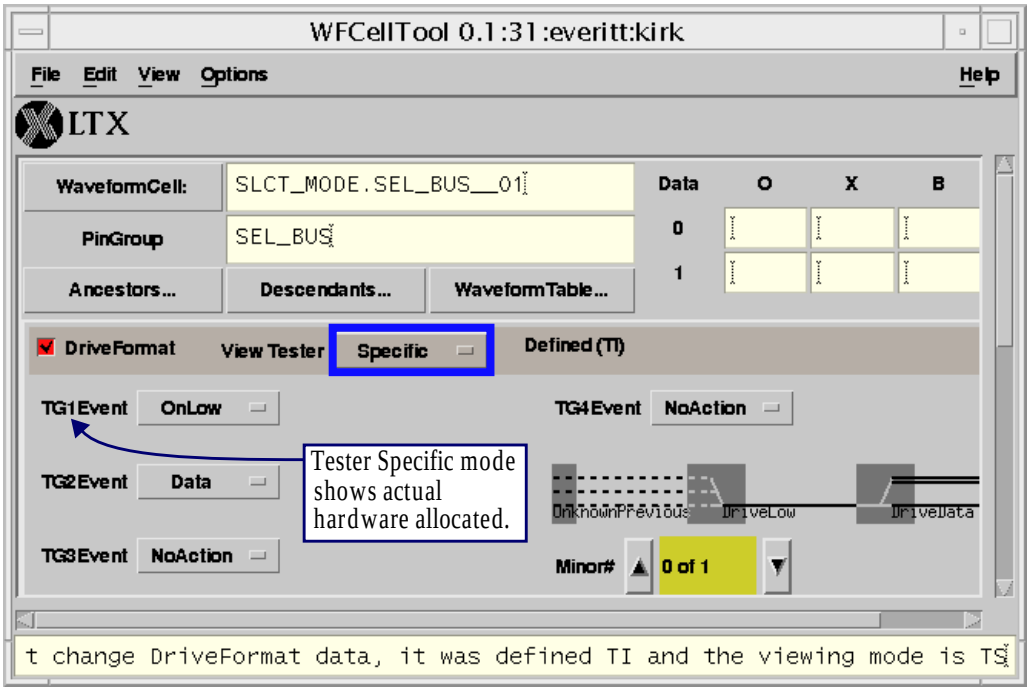


Figure 2.27: Creating Custom Waveform Using WFCellTool



Format created and viewed in Tester Independent mode.



Same format in Tester Specific mode.

Figure 2.28: Creating Custom Waveform Using WFCellTool, Tester Independent and Tester Specific Modes

Tutorial: Creating a WaveformTable Object

The following list is a summarizes how to create a `WaveformTable` object by using WaveTool:

1. Opening WaveformTool
2. Naming a New Waveformtable Object
3. Defining the Period
4. Creating WaveformTable Cells, AddingRows, and Assigning Row Names and Pins
5. Defining the OXBI Alias Characters
6. Assign formats to the cells:
   - Assigning Drive Waveform to WaveformTable Cells
   - Assigning Compare Waveform to WaveformTable Cells
7. Assigning Timing Relationships
8. Attaching WaveformTable Object to Zipper Table
9. Summary and examples; refer to page 2-133.

Opening WaveformTool

1. In OperatorTool, select Tools -> Development ->Waveform.
2. On WaveformTool, set Timing Display to Direct.
3. You can use the mouse buttons to quickly make certain WaveTool selections; refer to WaveformTool Mouse Conventions.

Naming a New Waveformtable Object

1. In the WaveformTable field, click mouse button M1.
2. Enter the name of the `WaveformTable` object; refer to Figure 2.29.
3. Press <**`Return`**>.

NOTE New waveform will be different color, meaning it is an exact copy of another cell. This cell will be ignored by the Waveform Compiler until a change is made, to differentiate it from the original cell.

Defining the Period

1. Click M1 on Period... button; refer to Figure 2.29.
2. In Define Period popup, click M1 on Period Expr...
3. In Find Period Spec popup, select Tper.
4. Click M1 on OK.

Creating WaveformTable Cells, AddingRows, and Assigning Row Names and Pins

1. In the blank view area below the Row heading, click mouse button 3 (M3) to open the Row Ops menu; Figure 2.30.
2. Click M3 on Add Row... of the Row Ops popup to add a row.
3. In the Add Row—WaveformCell popup, click mouse button (M1) on the Pins... button.
4. From the *enVision* Pin Finder window, use M1 to select desired pin.
5. In Pin Finder popup, click OK to put the selected pin into the Add Row—WaveformCell window.
6. Enter the Row name using the naming method in Cell Naming Convention.
7. Click M1 on the *OK* button of the Add Row—WaveformCell to add the row to the `WaveformTable`.

Defining the OXBI Alias Characters

1. Click M1 in the appropriate OXBI column; refer to Figure 2.31.
2. Type the alias symbols separated by a / in the text box.
3. Press <**`Return`**> to place the alias symbols into the appropriate triangles of the selected OXBI column.

Assigning Drive Waveform to WaveformTable Cells

1. Place the cursor in the black area to the right of each cell.
2. Click M3 in the View area to the right of the row just defined. The Edge popup appears.
3. In the Edge popup menu, select Drive Waveforms, go to the cascade menu and select appropriate I/O format, then go to the cascade menu and select the required Drive Data format; refer to Figure 2.32.

NOTE If a waveform does not appear, verify the Period has been defined. Remember the Period must be defined after the WaveForm object is created, otherwise the display may be inaccurate. If necessary, click on the AutoScale button and then click on Zoom Out button to adjust the display.

Assigning Compare Waveform to WaveformTable Cells

1. Place the cursor in the black area to the right of each cell.
2. Click M3 in the View area to the right of the row just defined. The Edge popup appears.
3. To add the Driver format: In the Edge popup menu, select Drive Waveforms, go to the cascade menu and select I/O format OFF, go to the cascade menu and select any for turning off the driver; refer to Figure 2.33.

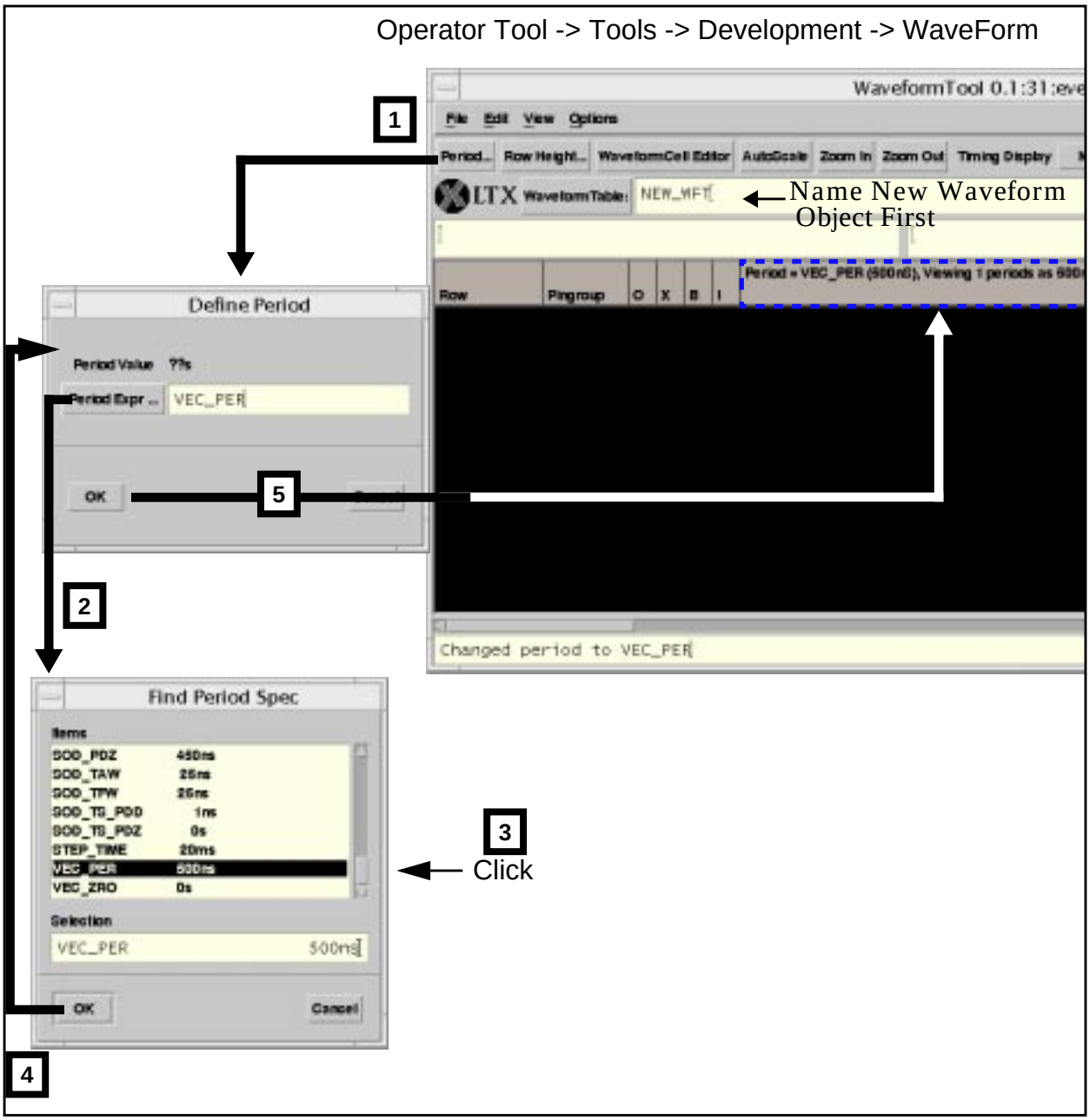


Figure 2.29: Creating a Waveform Table Object and Its Period

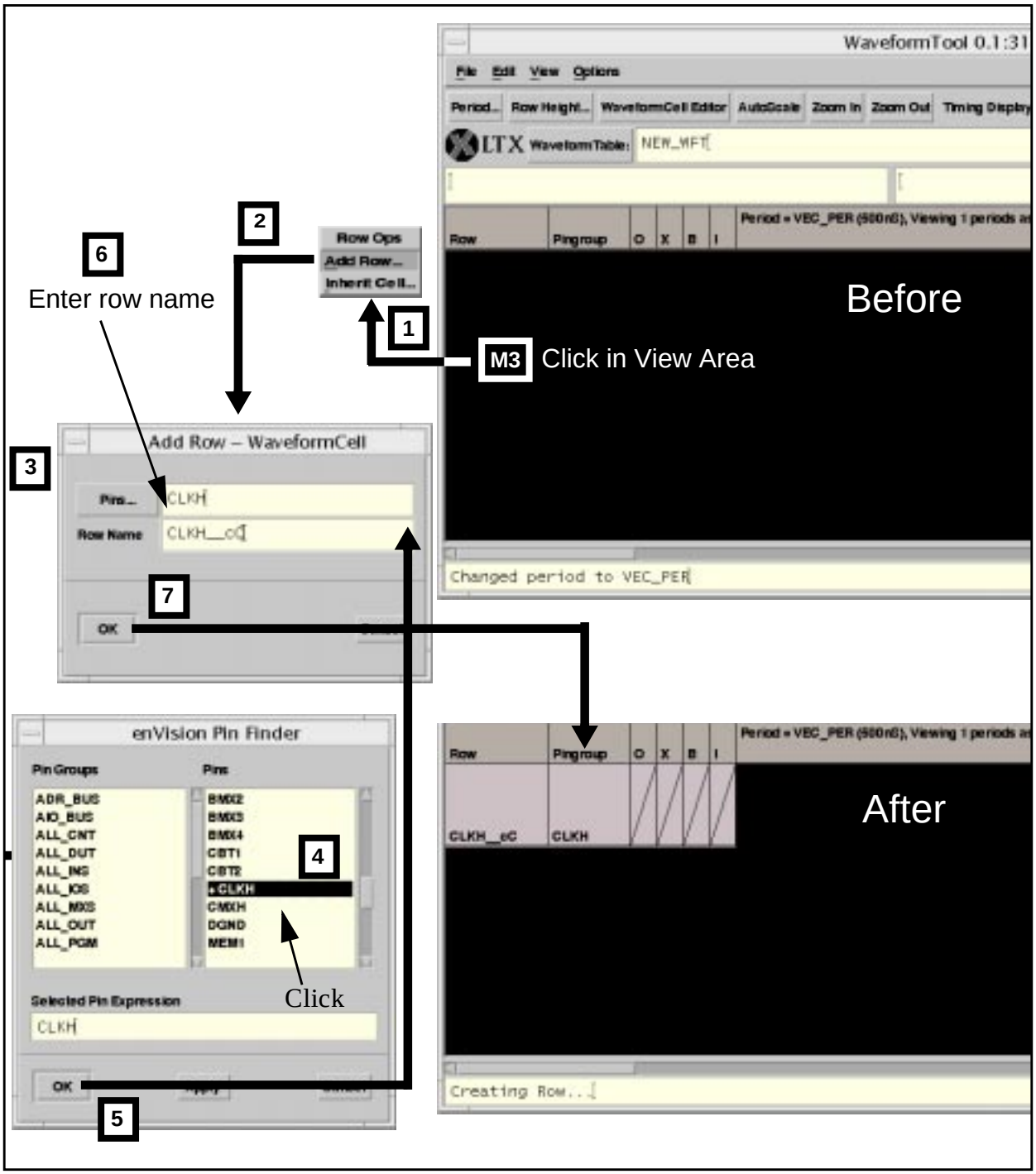


Figure 2.30: Creating a Partial Row or Cell

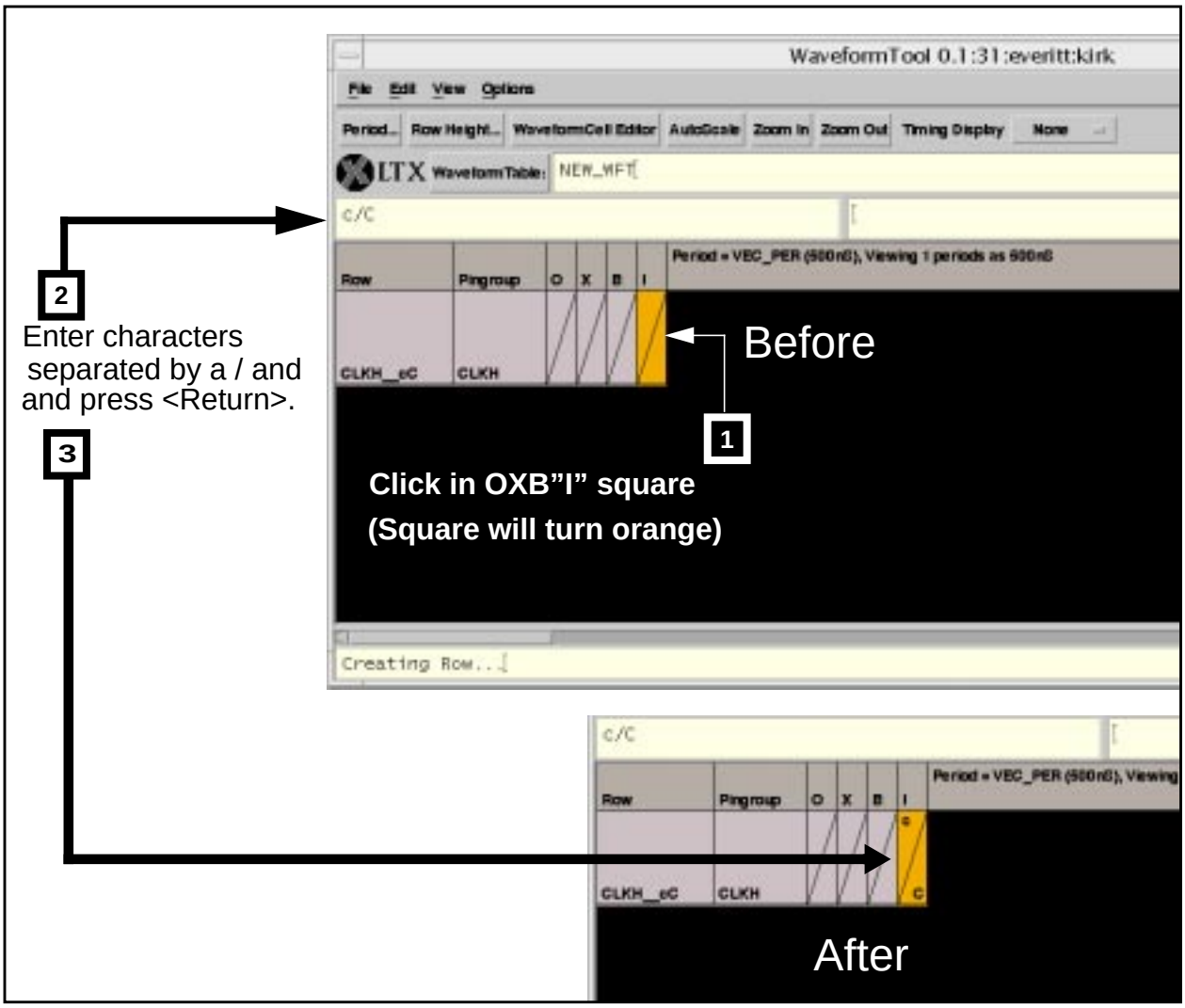


Figure 2.31: Adding Alias Characters to a Row

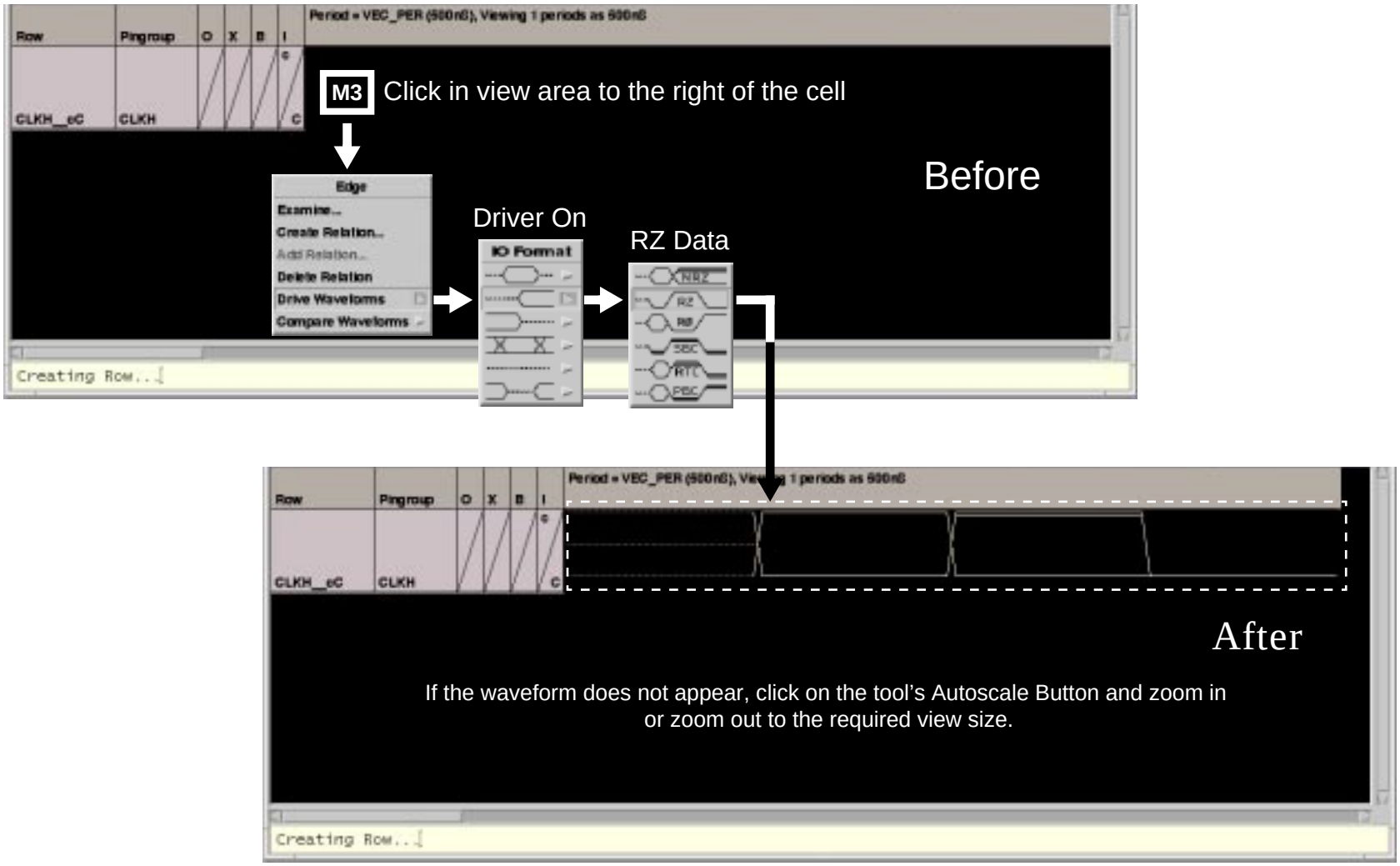


Figure 2.32: Adding Drive Information to a Cell

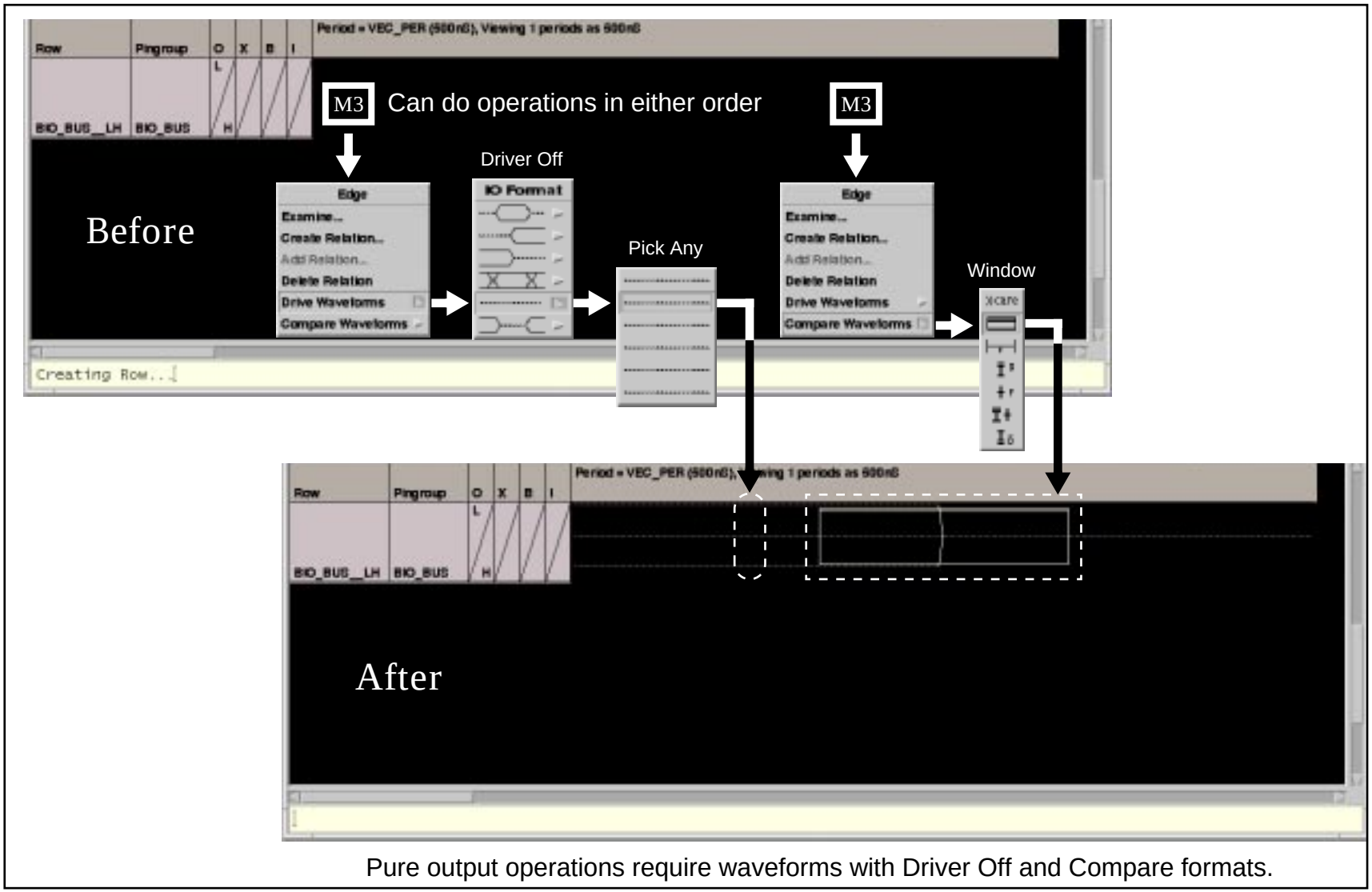


Figure 2.33: Adding Compare Information to a Cell

Assigning Timing Relationships

1. Locate the anchors for all Unknown Previous-to-Driver On/Off transitions; refer to Timing Relationships.

2. Click M2 on the transition point and release to open the Create Anchor popup. Anchor icon is an inverted triangle next to the timing expression or value.

3. Select Param: Tzero; see Figure 2.34.

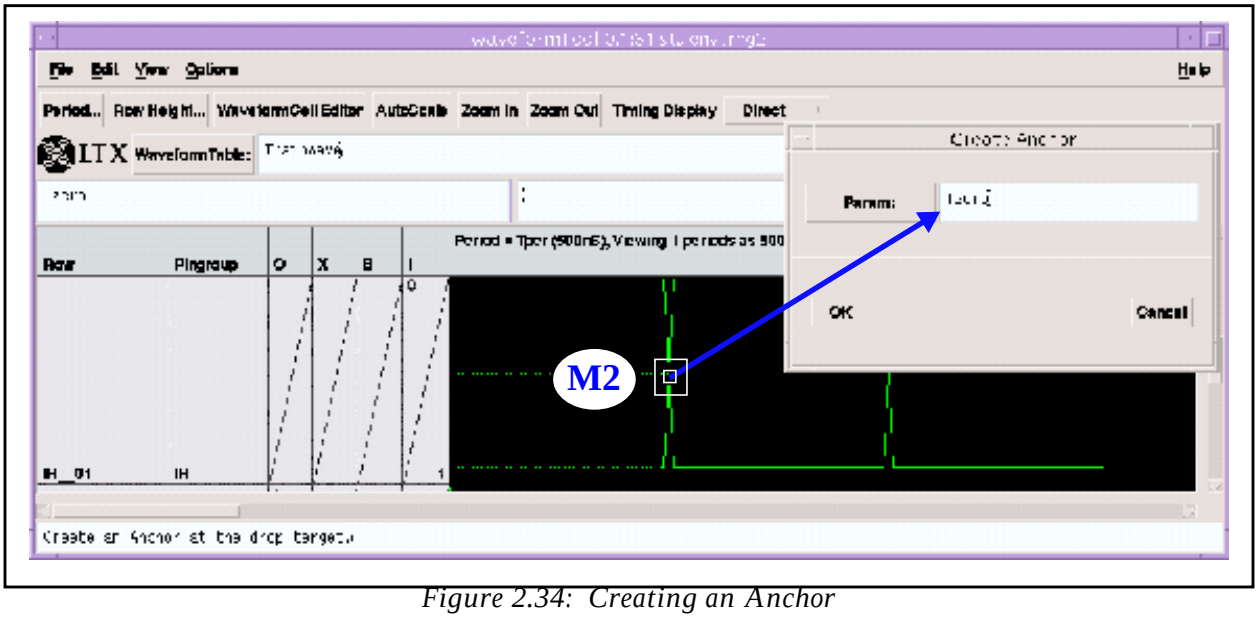


Figure 2.34: Creating an Anchor

4. Click M2 on the reference point to create a timing relation.

5. Drag the cursor to the required transition point and release M2.

6. The Create Timing Relation popup appears, where you select a parameter or type in an offset value; see Figure 2.35. The Timing Relation icon is a pointing left / right triangle next to the timing expression or value; refer to Timing Relationships.

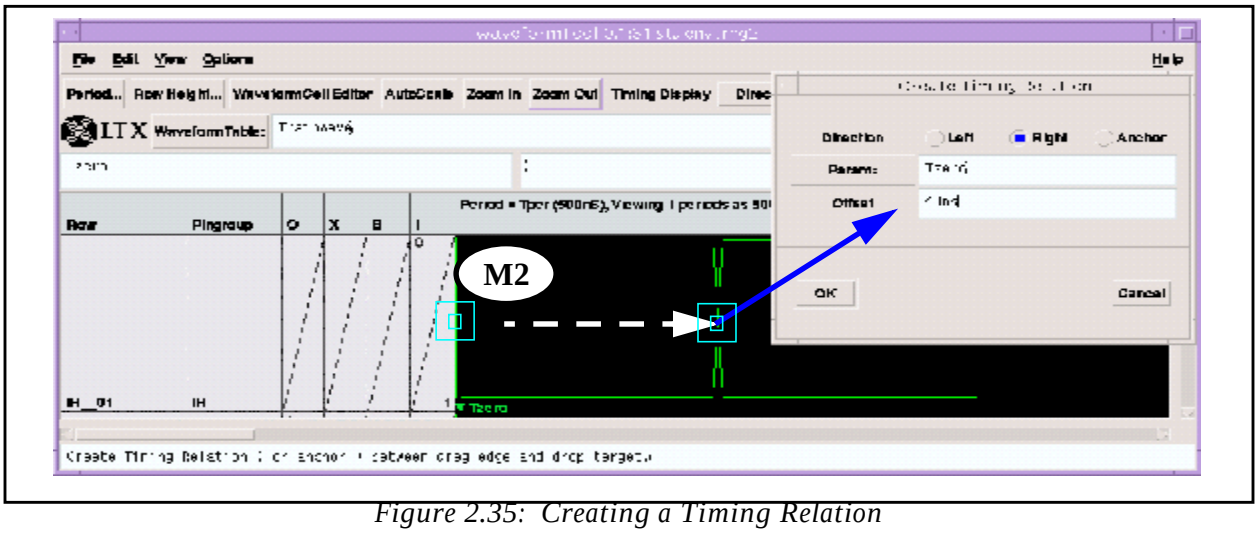


Figure 2.35: Creating a Timing Relation

7. To create relative timing with a timing spec parameter:

   a. Place cursor on reference timing edge.

   b. Press and hold M2. Cursor becomes a box.

   c. Drag cursor to timing edge to be defined.

   d. Release M2. Create Timing Relation window appears.

   e. Click M1 on Param: button. Time Parameters window appears.

   f. Use M1 to select required time spec parameter from list. Or, use keyboard to enter a timing value into text field next to Param: button.

   g. Click M1 on *OK* button of Timing Parameters window to place selected parameter into Create Timing Relation Param: field.

   h. Click M1 on *OK* button in Create Timing Relation window to complete this definition. Notice the edge just defined has arrow pointing to the right just to the left of the timing parameter name selected shown in Figure 2.36. Another similiar example is shown in Figure 2.37.

8. To create absolute timing with a timing spec parameter:

   a. Place cursor on the transition point of the edge.

   b. Click M2 and release. Create Anchor popup appears.

   c. Click M1 on Param: button. Time Parameters window appears.

   d. Use M1 to select the required time spec parameter from the list. Or, use the keyboard to enter an absolute timing value into the text field next to the Param: button; see Figure 2.38.

   e. Click M1 on the *OK* button to place the time parameter into the Param: field of the Create Anchor window.

   f. Click M1 on the *OK* button of the Create Anchor window to complete this timing definition on the timing edge as shown in Figure 2.39.

   g. Set Display mode to Expression. The time spec parameter name is displayed. A down pointing arrow (anchor) to the left of the time spec parameter confirms that it is an absolute timing definition.

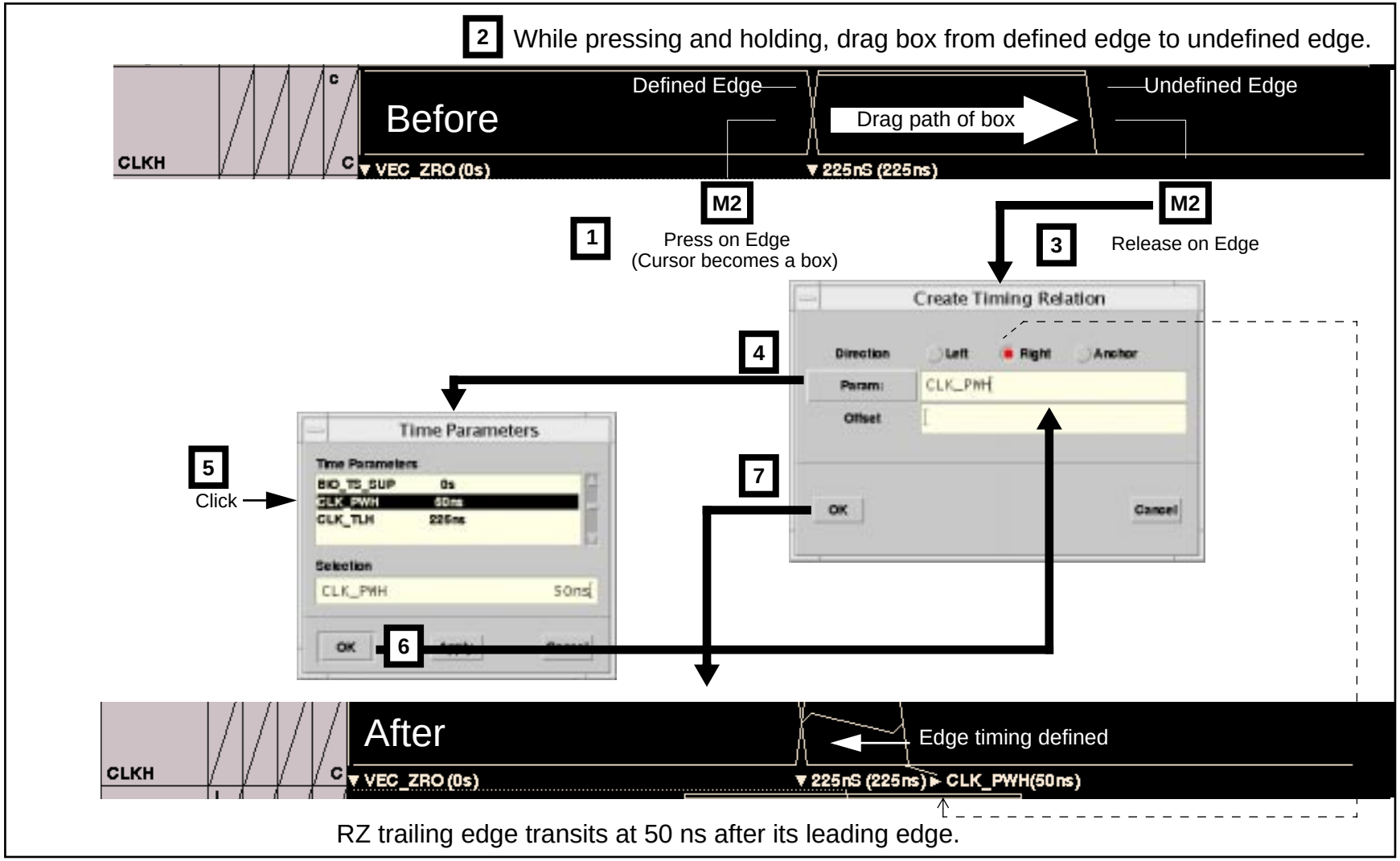


Figure 2.36: Attaching Relative Timing by Using a Spec Parameter

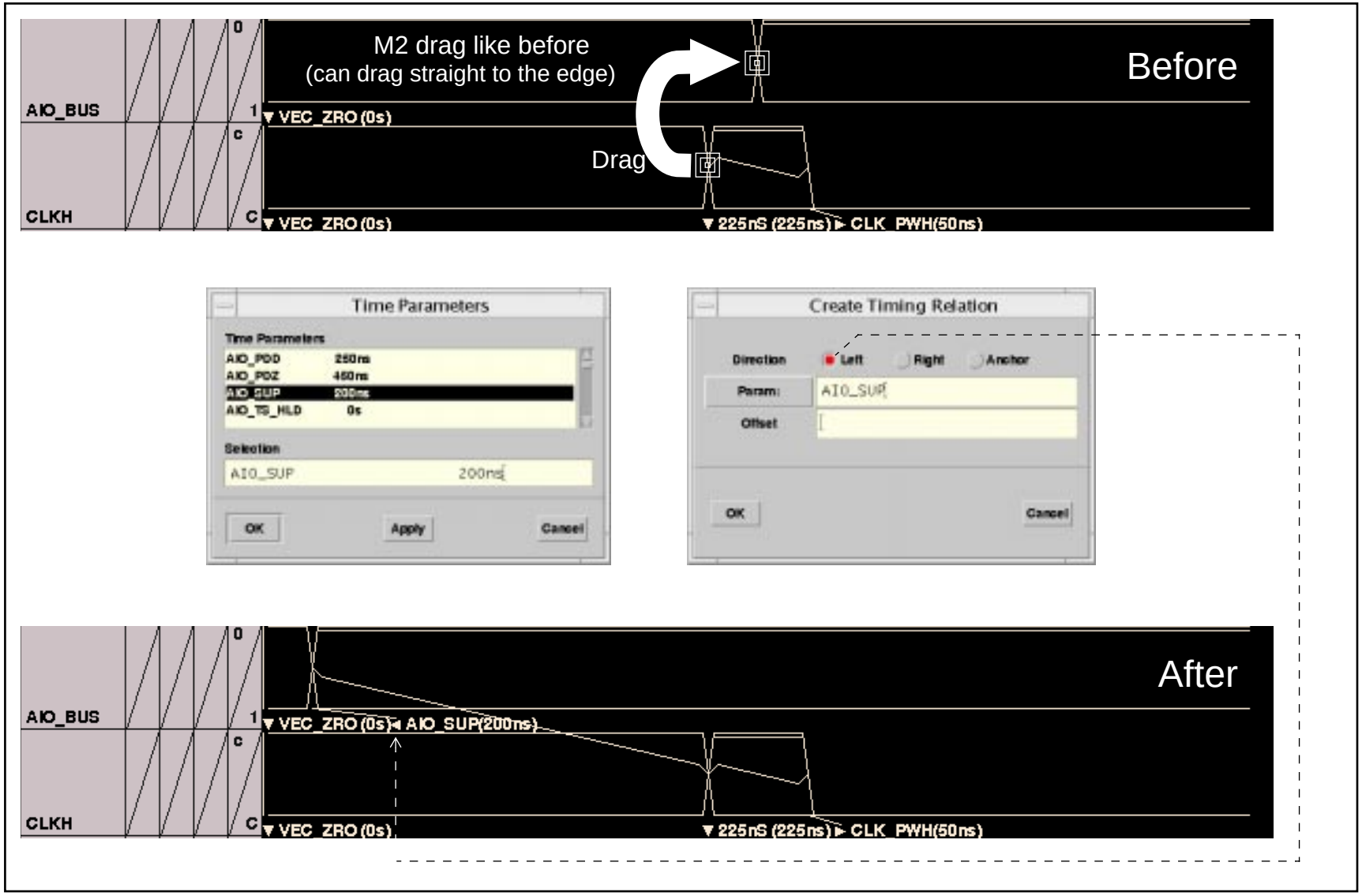


Figure 2.37: Second Example: Attaching Relative Timing by Using a Spec Parameter

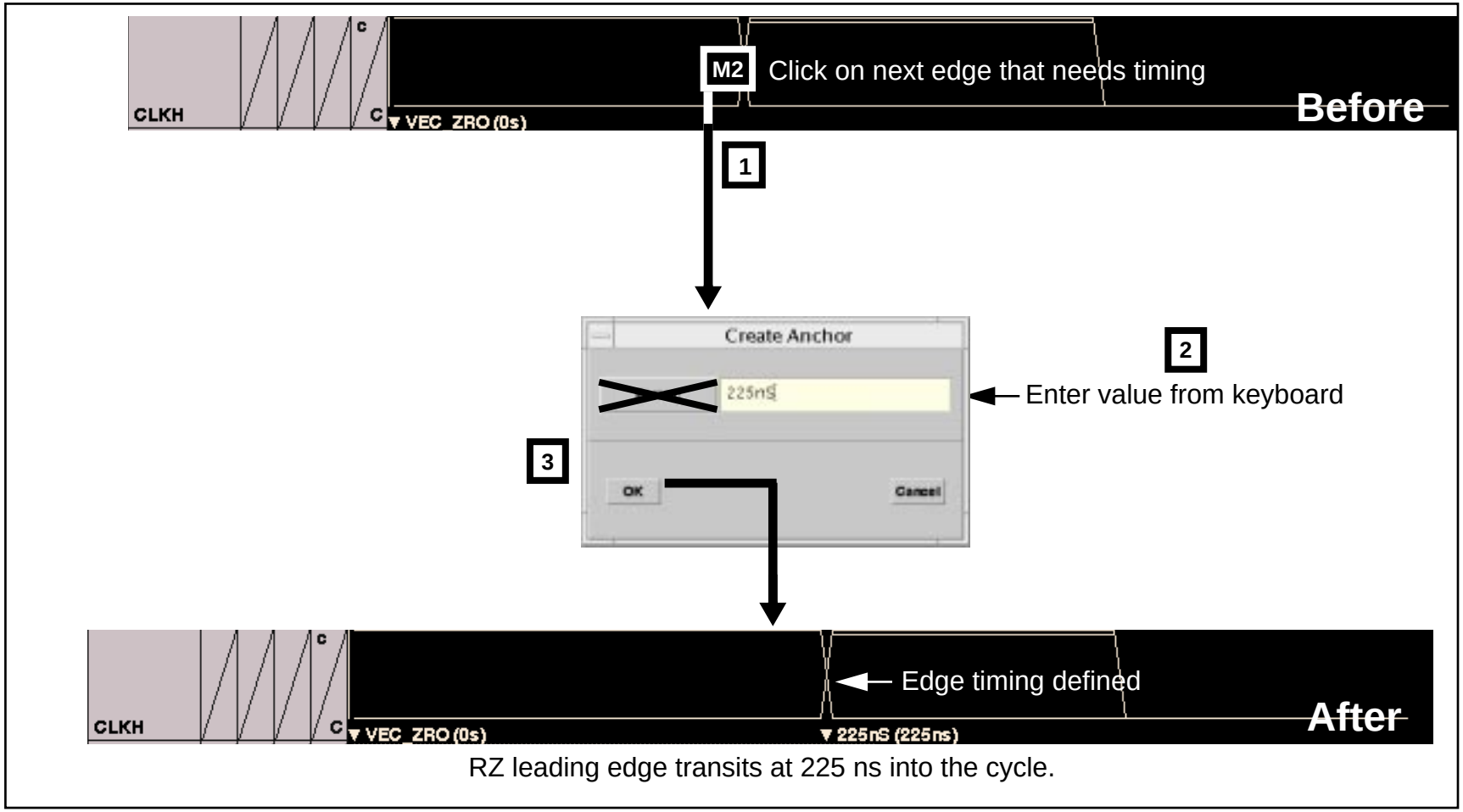


Figure 2.38: Attaching Absolute Timing by Using Keyboard

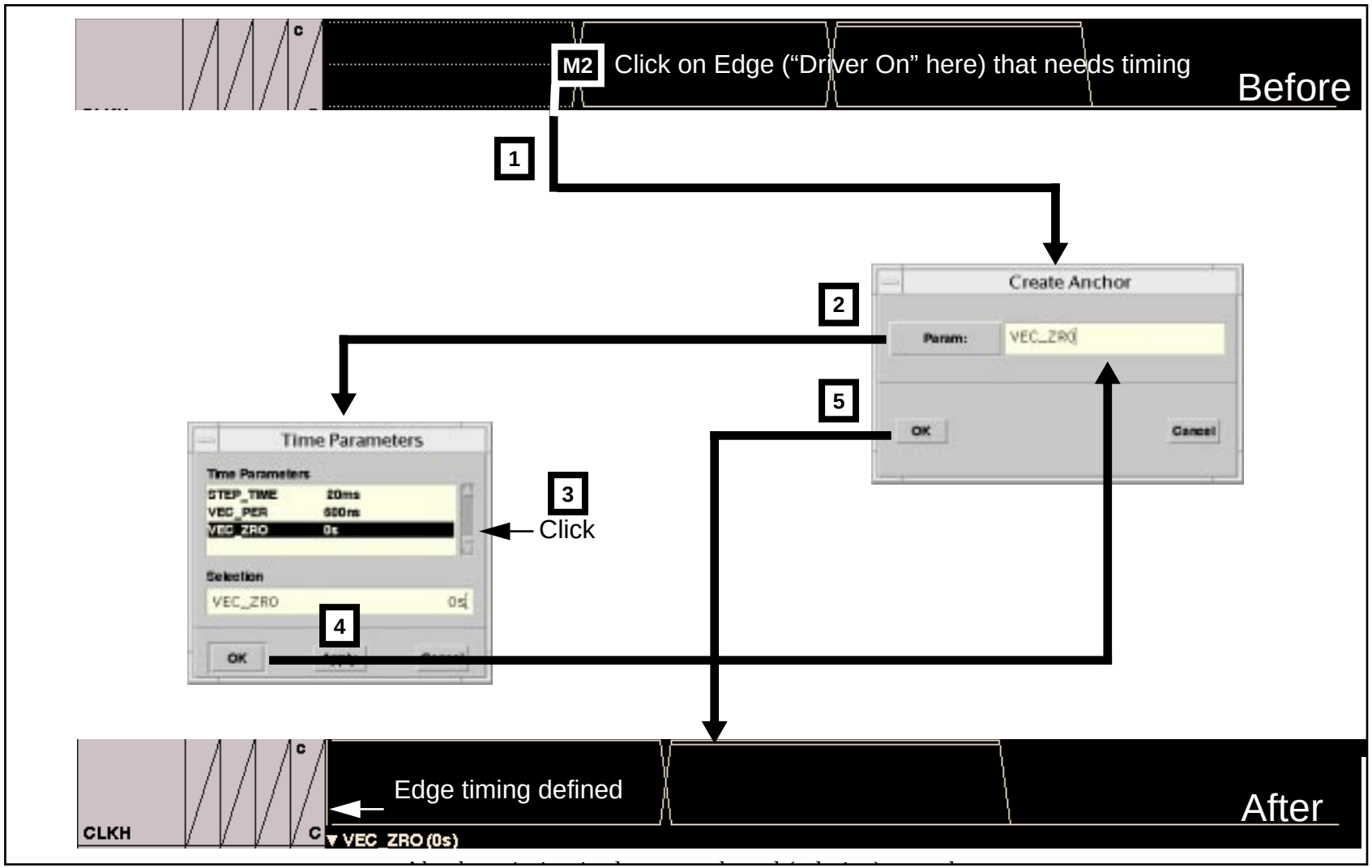


Figure 2.39: Attaching Absolute Timing by Using a Spec Parameter

Attaching WaveformTable Object to Zipper Table

1. Launch the `Pat_Seq PatternSequence` object fromthe `EZ_Functional Test` object:

   a. In FlowTool, double click on the EZ_Functional `Test` icon.

   b. In TestTool, double click on the `Pat_Seq` icon in the Entry Objects area.

2. Attach the `WaveformTable` object at the Zipper Table:

   a. Click only on the Waveform Tables button (ticked when selected) at the top of the PatternSequence Tool.

   b. In the first empty field under the WaveformTable column of the first row, press M3.

   c. At the Zipper Actions popup, choose Find...

   d. At the enVisionObject popup, select the desired wave from list.

   e. Click M1 on OK.

3. Figure 2.40 shows the *Zipper Table* in the `PatternSequence` object: `Pat_Seq`.

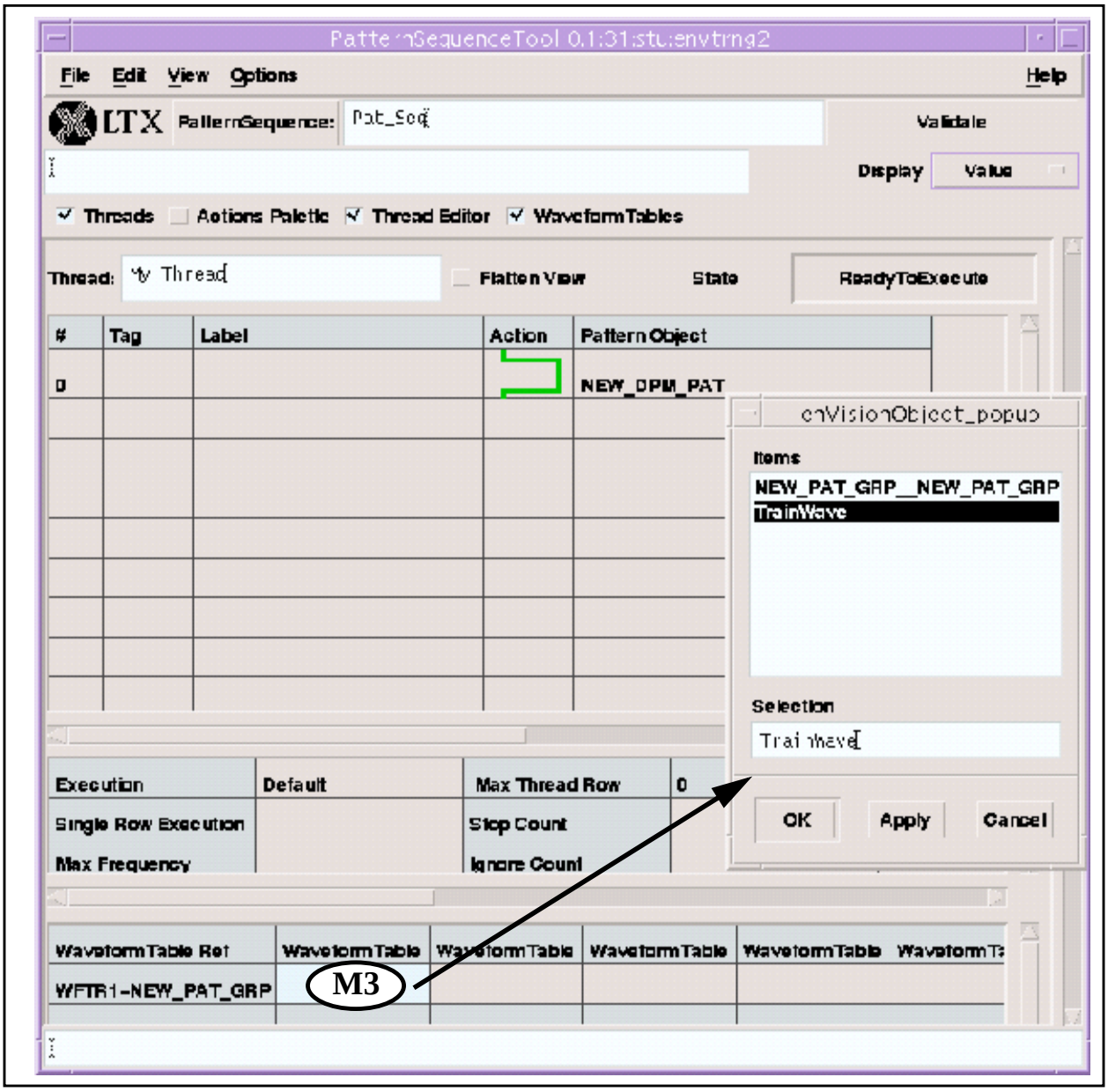


Figure 2.40: Waveform Object Attached to Zipper Table

Examples

Figure 2.41 shows three sample cells with formatted waveforms:

- First cell

  I/O format: OFF—ON.

  Driver Waveform: NRZ.

- Second cell

  I/O format: OFF—ON.

  Driver Waveform: RZ.

- Third cell

  I/O format: OFF.

  Compare Format: Active Window.

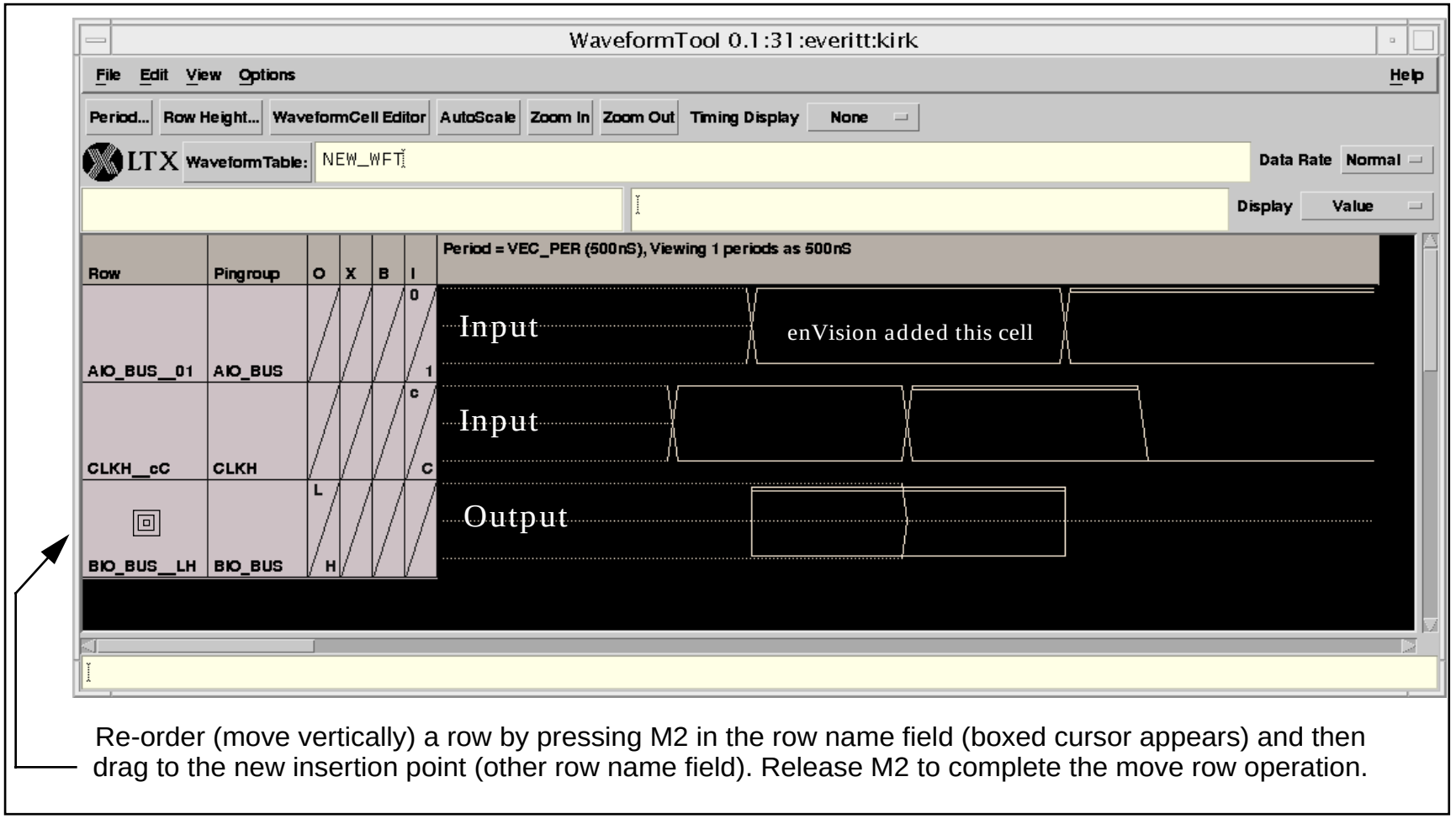


Figure 2.41: Sample Formatted Cells

Example: WaveformTable Object

Excerpt from .eva file shows example of a `WaveformTable` object:

```
WaveformTable Data_in {
    Period "Per";
    Cell "NRZ_ins_and_Ready" 2/3 Data_in_0 {
        Inherit Z8600_pats__Z8600_pats._2_3;
        Drive {
            Waveform { DriveOn @ "Ton"; DriveData @ D1 -> T0 + "0nS"; }
            Control DriveOn NonMuxed;
        }
    }
    Cell "Clk" 4/5 Data_in_1 {
        Inherit Z8600_pats__Z8600_pats._4_5;
        Drive {
            Waveform { DriveOn @ T0 + "0nS" <- Data_in_0.D1; DriveData
            @ D1 -> T1 + "0nS"; DriveLow @ D2 -> PW + "0nS"; }
            Control DriveOn NonMuxed;
        }
    }
    Cell "All_outs" L/H WFC3 {
        Inherit Z8600_pats__Z8600_pats._L_H;
        Drive {
            Control DriveOff NonMuxed;
        }
        Compare {
            Waveform { CompareOpenData @ Data_in_1.D3 -> Tpd + "0nS";
            CompareClose @ Data_in_1.D3 -> Strobe_end + "0nS"; }
            Control Care NonMuxed LoadTimed;
        }
    }
    Cell "RQ_gnt" L/H Data_in_3 {
        Inherit Z8600_pats__Z8600_pats._L_H;
        Drive {
            Waveform { DriveOff @ Data_in_0.D1 -> Ton + "0nS"; }
            Control DriveOn NonMuxed;
        }
        Compare {
            Waveform { CompareOpenData @ Data_in_1.D3 -> Tpd + "0nS";
            CompareClose @ Data_in_1.D3 -> Strobe_end + "0nS"; }
            Control Care NonMuxed LoadTimed;
        }
    }
    Cell "RQ_GT0" 2/3 Data_in_4 {
        Inherit Z8600_pats__Z8600_pats._2_3;
        Drive {
            Waveform { DriveOn @ Data_in_0.D1 -> Ton + "0nS";
            DriveData @ D1 -> T0 + "0nS"; }
            Control DriveOn NonMuxed;
        }
    }
```

```
    Cell "AD_bus" 0/1 Data_in_5 {
        Inherit Z8600_pats__Z8600_pats._0_1;
        Drive {
            Waveform { DriveDataNot @ Data_in_0.D1 -> Ton + "0nS";
            DriveData @ Ts + "0nS" <- Data_in_1.D3; DriveDataNot
            @Data_in_1.D3 -> Th+"0nS"; DriveOff @D1 -> Toff +"0nS"; }
            Control DriveOn NonMuxed;
        }
        Compare {
            Waveform { CompareData @ D2 -> Ton + "0nS"; }
            Control Care NonMuxed LoadTimed;
        }
    }
}
```

Example: WaveformTable in a Complete Program

In the pattern sequence portion of this example test program, a `WaveformTable` is specified that has not been yet defined. The following excerpt from the .eva file define this `WaveformTable`.

Creating a `WaveformTable` object by using the WaveformTool and WaveformCell EditorTool is easier than creating it with a text editor.

```
WaveformTable LS245_pats{
    Cell "LS245_pats.Pins" 0/1 Drive_data {
        Data 6/7;
    }
    Cell "LS245_pats.Pins" L/H Compare_data {
        Data 0/1;
    }
    Cell "LS245_pats.Pins" z/Z Tristate_data {
        Data 0/1;
    }
}
WaveformTable Timing {
    Period "100nS";
    Cell "All_ins+All_bids" 0/1 Ins {
        Data 6/7;
        Drive {
            Waveform { DriveOn @ "0nS"; DriveData @ D1 -> "0S"; }
        }
    }

    Cell "All_bids" L/H Outs {
        Data 0/1;
        Drive {
            EntryState DriveOn;
```

```
          Waveform { DriveData @ Ins.D1 -> "0S"; DriveOff @ D1 -> "0S"; }
        }
        Compare {
            Waveform { CompareData @ Ins.D1 -> TPHL + "20nS"; }
        }
    }
    Cell "All_bids" z/Z High_Z {
        Data 0/1;
        Inherit Outs;
        Compare {
            Waveform { CompareFloat @ Outs.D1 -> TPHZ + "20nS"; }
        }
    }
}
WaveformTable Tplh_tmg {
    Period "100nS";
    Cell "All_ins+All_bids" 0/1 Ins {
        Data 6/7;
        Drive {
            Waveform { DriveOn @ "0nS"; DriveData @ D1 -> "0S"; }
        }
    }
    Cell "All_bids" L/H Outs {
        Data 0/1;
        Drive {
            EntryState DriveOn;
          Waveform { DriveData @ Ins.D1 -> "0S"; DriveOff @ D1 -> "0S"; }
        }
        Compare {
            Waveform { CompareData @ Ins.D1 -> TPLH + "0nS"; }
        }
    }
}
WaveformTable Tpzh_tmg {
    Period "100nS";
    Cell "All_ins+All_bids" 0/1 Ins {
        Data 6/7;
        Drive {
            Waveform { DriveOn @ "0nS"; DriveData @ D1 -> "0S"; }
        }
    }

    Cell "All_bids" L/H Outs {
        Data 0/1;
        Drive {
            EntryState DriveOn;
          Waveform { DriveData @ Ins.D1 -> "0S"; DriveOff @ D1 -> "0S"; }
        }
        Compare {
```

```
            Waveform { CompareData @ Ins.D1 -> TPZH + "0nS"; }
        }
    }
}
WaveformTable Tphl_tmg {
    Period "100nS";
    Cell "All_ins+All_bids" 0/1 Ins {
        Data 6/7;
        Drive {
            Waveform { DriveOn @ "0nS"; DriveData @ D1 -> "0S"; }
        }
    }
    Cell "All_bids" L/H Outs {
        Data 0/1;
        Drive {
            EntryState DriveOn;
          Waveform { DriveData @ Ins.D1 -> "0S"; DriveOff @ D1 -> "0S"; }
        }
        Compare {
            Waveform { CompareData @ Ins.D1 -> TPHL + "0nS"; }
        }
    }
}
WaveformTable Tphz_tmg {
    Period "100nS";
    Cell "All_ins+All_bids" 0/1 Ins {
        Data 6/7;
        Drive {
            Waveform { DriveOn @ "0nS"; DriveData @ D1 -> "0S"; }
        }
    }
    Cell "All_bids" z/Z High_Z {
        Data 0/1;
        Drive {
            EntryState DriveOn;
          Waveform { DriveData @ Ins.D1 -> "0S"; DriveOff @ D1 -> "0S"; }
        }
        Compare {
            Waveform { CompareFloat @ Ins.D1 -> TPHZ + "0S"; }
        }
    }
}
WaveformTable Tpzl_tmg {
    Period "100nS";
    Cell "All_ins+All_bids" 0/1 Ins {
        Data 6/7;
        Drive {
            Waveform { DriveOn @ "0nS"; DriveData @ D1 -> "0S"; }
        }
```

```
    }
    Cell "All_bids" L/H Outs {
        Data 0/1;
        Drive {
            EntryState DriveOn;
          Waveform { DriveData @ Ins.D1 -> "0S"; DriveOff @ D1 -> "0S"; }
        }
        Compare {
            Waveform { CompareData @ Ins.D1 -> TPZL + "0nS"; }
        }
    }
}
WaveformTable Tplz_tmg {
    Period "100nS";
    Cell "All_ins+All_bids" 0/1 Ins {
        Data 6/7;
        Drive {
            Waveform { DriveOn @ "0nS"; DriveData @ D1 -> "0S"; }
        }
    }
    Cell "All_bids" z/Z High_Z {
        Data 0/1;
        Drive {
            EntryState DriveOn;
          Waveform { DriveData @ Ins.D1 -> "0S"; DriveOff @ D1 -> "0S"; }
        }
        Compare {
            Waveform { CompareFloat @ Ins.D1 -> TPLZ + "0S"; }
        }
    }
}
```

Timing

Timing in an enVision test program is not defined as a separate group of objects that are applied to another base object. Instead, timing is a set of expressions and values specified for each Waveform cell of a WaveformTable. You specify the timing in the WaveTool Timing Display. The timing expressions and values you specify depend on the enVision operating mode.

Timing Expressions

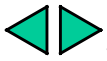
Timing is expressed as a relationship of events within the waveform cells; refer to Timing Relationships. For instance, each Driver, Drive Data, and Comparator waveform is an event. Drive events are named D1 and D2, while the Compare events are named C1 and C2. The numbering is assigned in the order of event creation, not event timing.

Also, each transition edge has a defined timing value, programmed as a timing offset, a timing spec parameter, or an expression consisting of a timing spec parameter and a timing offset; see Figure 2.42. These edges are also known as *markers*. Also, refer to Timing Expressions.

Timing Relationships

enVision supports both absolute and relative timing. WaveformTool uses the following timing symbols to signify these two types:

- Anchor  —signifies absolute timing for a timing spec parameter or a timing offset; refer to Assigning Timing Relationships.

- Left and right arrows  —signify relative timing, which requires a reference edge for a timing spec parameter, timing offset or combination of a timing spec parameter with a timing offset; see Figure 2.42.

  A reference edge may be from its own cell or from another waveform cell.

  A timing relation may be to the left or to the right of the reference timing edge.

WaveTool Timing Display

Timing information is displayed in one of three formats: value, expression, or edge value; see Figure 2.43, Figure 2.44, and Figure 2.45.

- Value—Timings are displayed with respect to *T0*. Clock signals are sent to the DUT pins at a defined rate, known as the *T-clock rate*. The pattern data sent to the DUT is updated at the clock rate, which is referred to as the *T0 period* or *bit-cell*.
- Expression—Only the variable name or constant value is displayed; refer to Example: Timing Expressions.
- Edge value—The values are shown as absolute numbers as programmed to the hardware.

Timing Syntax

To view the complete syntax for the timing expressions, use WFCell Tool. enVision creates these expressions in the WaveformTool as you draw the waveshapes and add the timing references; refer to Examples.

Examples

- Anchored marker is defined by a `spec_variable`. Timing marker is relative to the beginning (`T1`) of the period of the cycle in which it occurs:

```
@ spec_variable
@ T1
```

- Anchored marker similar to the previous example except it is defined by a timing offset expression consisting of constants, Spec variables, operators or functions:

```
@ “timing_offset_expression”
@ “12nS”
```

- Anchored expression with both a Spec variable and an offset timing expression:

```
@ spec_variable + “timing_offset_expression”
@ T1 + “7.5nS”
```

- Marker directly linked to another marker in another cell. In this format, you must specify all three parts of the reference, even if the reference is a cell in the same `WaveformTable`:

```
@ “waveform_table_name.cell_name.event”
@ DC.DC_6.D1
```

- Marker referenced to a prior marker in the current cell definition. The `->` symbol means to *go right* from the reference even by the value contained in `spec_variable`:

```
@ event -> spec_variable
@ D1 -> T0 + “0nS”
```

- Example is similar to the previous one except the reference is to a marker in another cell of the same `WaveformTable`:

```
@ cell_name.event -> spec_variable
@ DC_6.D1 -> T0
```

- Refers to a marker in another `WaveformTable`:

```
@ waveform_table_name.cell_name.event ->
spec_variable
@ Timing.DC_6.D1 -> T0 + “0nS”
```

- Event referenced to a later event. Note the order is reversed: referenced event is to the right, the `<-` symbol means *to go* left from the referenced marker by the value contained in `spec_variable`:

```
@ spec_variable <- event
@ Ts + “0nS” <- DC_6.D3
```

NOTE You can define a cell with only events or only timing, which is sometimes desirable if the event or timing information will be inherited.

Example: Timing Expressions

The following excerpt from the .eva file shows examples of `WaveformTable` cell definitions using timing expressions:

```
Drive { Waveform {
    DriveDataNot @ "3nS";
    DriveData @ D1 -> tspec + "tx/3";
    DriveDataNot @ tspec <- D4;
    DriveOff @ "w2.cell_g.D1";
```

```
} }
Drive { Waveform {
    DriveOn;
    DriveData;
    DriveLow;
    DriveOff;
} }
Drive { Waveform {
    @ "3nS";
    @ txyz;
    @ tspec + "tx/3";
    @ D1 -> tspec;
} }
Compare { Waveform {
    CompareOpenData @ D1 -> tzz + "5nS";
    CompareClose @ C1 -> tw;
} }
```

Figure 2.42: Examples of Spec Parameters with Offsets

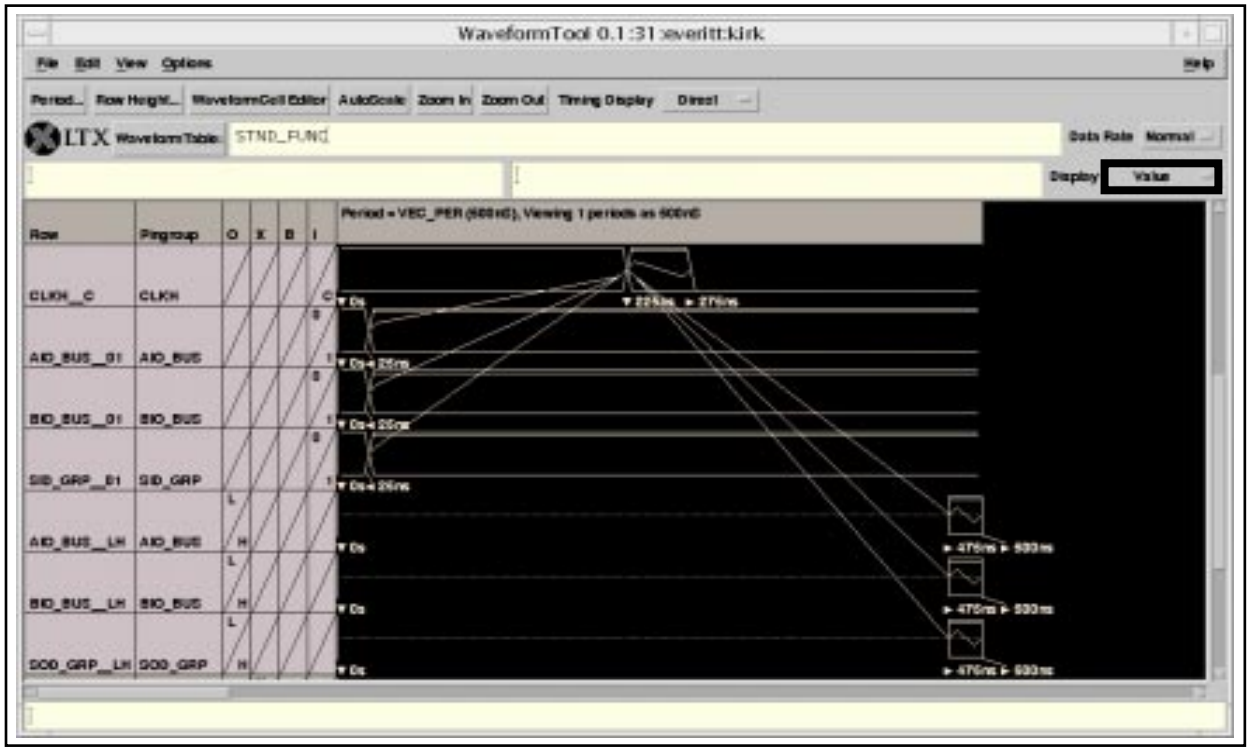


Figure 2.43: Waveform Timing Relationship, Value Format

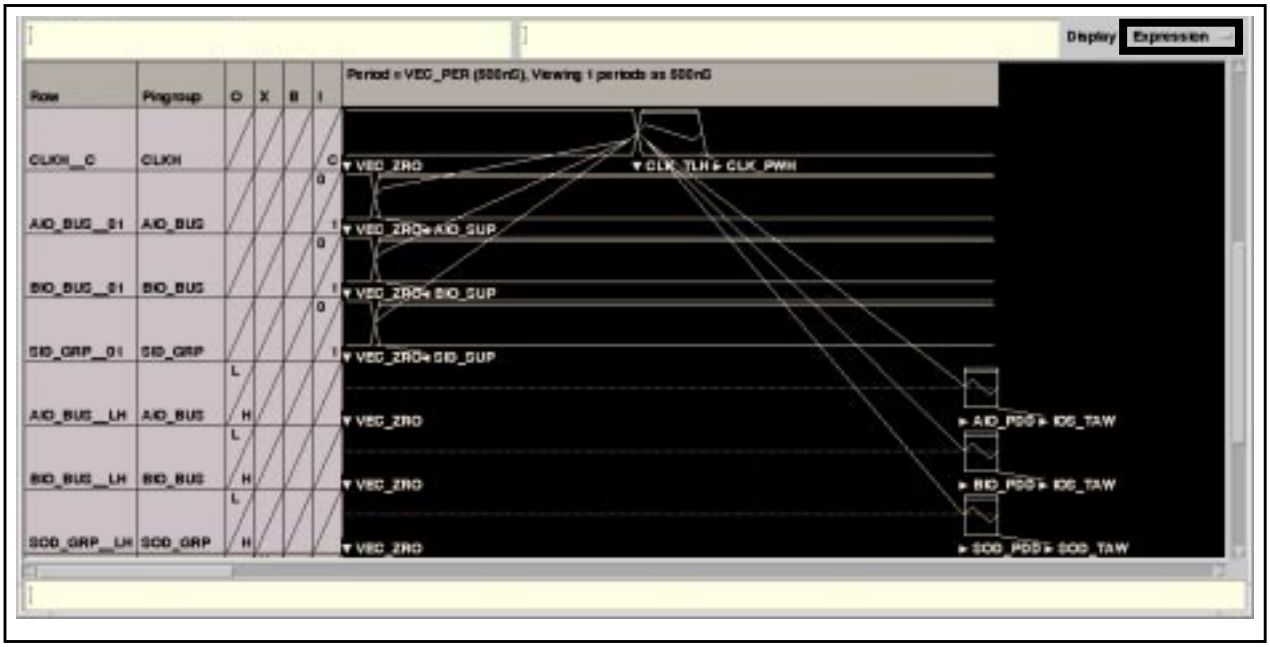


Figure 2.44: Waveform Timing Relationship, Expression Format

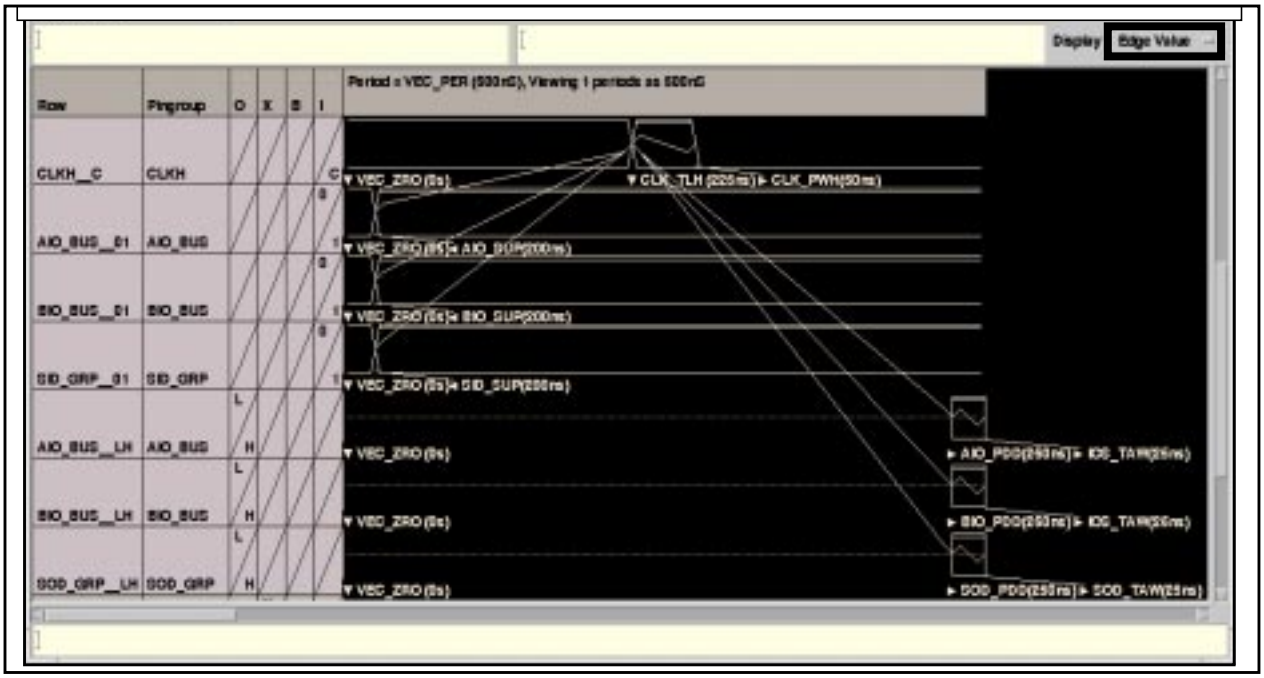


Figure 2.45: Waveform Timing Relationship, Edge Format

Operating Modes

NOTE The alpha release of the enVision software for the Fusion HFi (VX IV) tester supports test rates up to 1.4 GHz.

The VX IV tester has two normal operating frequency modes and three enhanced frequency modes; refer to Table 2.7. Each mode is operated in the normal, `MuxMode`, or `MultiState` `PatternMode`:

- Normal operating modes:

a. SVM (Single Vector Mode), also known as *VX125* mode.

b. DVM (Dual Vector Mode), also known as *VX250* mode.

- Enhanced frequency modes:

a. DMM (Dual Marker Mode)

b. TMM (Triple Marker Mode)

c. Quad Marker Mode (QMM)

The default frequency range of a Fusion HFi tester is 1 to 125 MHz, which is also known as the *SVM* or *VX125* mode. To run a thread at speeds higher than 125 MHz, select one of the Mux modes or set the thread to run in the VX250 mode, which ranges from 250 to 500 MHz:

- To select the VX250 mode, enter a frequency number greater than 125 MHz but less than 500 MHz in the Max Frequency field of the Thread Editor in the PatternSequenceTool.

- For frequencies greater than 500 MHz, you must select one of the other operating modes; refer to Table 2.7.

The entered number does not determine the actual operating frequency of the thread. It is a switch that changes the operating mode from the default VX125 mode to another mode. The operating frequency is specified in the Period field of the WaveformTables in WaveformTool. Be aware that the period range and available per-pin time sets depend on the mode.

Operating Mode Characteristics

Each mode has advantages and different restrictions compared with the other modes; thus, one or more operating mode may support your digital application; refer to Table 2.7.

Each operating mode has the following characteristics:

- Makers per cycle—up to 4 edges per DUT cycle. As the tester rate increases, the number of markers (edges) per cycle decreases.
- DUT Rate—Channel throughput to the DUT is from 156.25 to 1400 Mb/s. Increasing this rate decreases the user period and the number of markers per cycle, and vice versa.
- User period—Time period of a cycle ranges from 0.741 to 6.4 ns.
- I/O Type—all modes use formatted I/O data except for one type of TMM that operates as an input or output channel.
- DUT/Tester Ratio—Test rate of DUT is up to 8 times the base tester rate of 156.25 MHz.
- RPT Micro-instructions—ranges from one RPT every 8 DUT cycles (1:8) to one every DUT cycle (1:1). Increasing the DUT rate decreases this ratio.
- Per-Pin Micro-instruction Ratio—ranges from one micro-instuction every 8 cycles (1:8) to one micro-instruction every cycle (1:1). Increasing the DUT rate decreases this ratio.
- CPM Micro-instructions Ratio—ranges from one CPM micro-instruction every 16 DUT cycles (1:16) to one every cycle (1:1) for CPM branching and period switching. Increasing the DUT rate decreases this ratio.
- Maximum DPM Size—DPM sizes from 64 to 512M are supported.

Table 2.7: Summary of Fusion Operating Frequency Modes and Characteristics

Operating Mode	Markers per DUT Cycle	DUT Rate (Mb/s)	User Period (ns)	I/O Type	DUT/ Tester Ratio	RPT Micro-Instruction	Per Pin Micro-Instruction Ratio	CPM Micro-Instruction Ratio	Max DPM Size (M)
SVM	4	156.25	6.400	I/O	1	1:1	1:1	1:1	64
DVM	4	312.50	3.200	I/O	1	1:1	1:1	1:2	64
DMM	2	625	1.600	I/O	2	1:2	1:2	1:4	128
TMM	1	[a]	[a]	I/O	3	1:3	1:3	1:6	192
TMM	1	937.5	1.067	I or O	3	1:3	1:3	1:6	192
QMM	1	1250	0.800	I/O	4	1:4	1:4	1:8	256
Mux SVM	4	312.5	3.200	I/O	2	1:2	1:1	1:2	128
Mux DVM	4	625	1.600	I/O	2	1:2	1:1	1:4	128
Mux DMM	2	1250[b]	0.800	I/O	4	1:4	1:2	1:8	256
Mux TMM	1	1400	0.714	I/O	6	1:6	1:3	1:12	384
Mux QMM	1	1400	0.714	I or O	8	1:8	1:4	1:16	512

a. TMM I/O Has three use models:
For these rule sets: X = Drive cycle
1. I/O changes every 3 cycles, phase aligned; maximum frequency = 937.5 Mb/s
2. I/O changes every 2 cycles, phase aligned, with no I/O in 3rd phase; maximum rate = 833 Mb/s.
3. User I/O mode. You must precede H or L in a pattern with X or 0. Maximum rate = 937.5 Mb/s.

b. QMM Muxed in output mode supports Don't Care cycles with the following rules:
A) Don't Care (X) cycles must be phase-aligned DUT cycle pairs.
B) The following three combinations are supported: XL, XH, and XX.

SVM (Single Vector Mode)

NOTE SVM is also known as the *VX125* mode.

Of all operating modes, the SVM supports all micro-instructions every cycle and offers more features every cycle than the other modes in the normal mode and MuxMode:

- SVM Normal mode properties:

  64 wavesets for each pin across the 4 timing markers (6^2 unique addresses = 64 bits).

  Maximum DUT rate is 156 Mb/s, 4 events per DUT cycle.

  Provides RZ, RO, and SBC with full-formatted I/O data.

  DUT rate equals the base tester rate repeat.

  `RPT` micro-instructions supported every DUT cycle.

  Per-pin micro-instructions supported every DUT cycle.

  Micro-instructions for CPM branching and period switching are supported every DUT cycle.

- SVM `MuxMode` properties:

  64 wavesets for each pin across the 4 timing markers.

  Maximum DUT rate equals 312 Mb/s, 4 events per DUT cycle.

  DUT rate is twice the base tester rate.

  `RPT` micro-instructions supported every second DUT cycle.

  Per-pin micro-instructions supported on all DUT cycles.

  Micro-instructions for CPM branching and period switching are supported on every second DUT cycle.

The example listed in Table 2.8 uses:

- 6 characters—H, L, 1, 0, X, and F—provide 6 waveforms (A0/B0 to A5/B5) per pin for each Timing Generator T1A/B to T4A/B.

- 10 global time sets (6-bits per pin x 10 waveforms/time sets = 60 bits, limit is 64 bits).
- 32 global period values.

Table 2.8: Example: Single Vector Mode

Timing Generator	T1A	T2A	T3A	T4A	T1B	T2B	T3B	T4B
Per Pin Waveset Pointers	A0	A0	A0	A0	B0	B0	B0	B0
	A1	A1	A1	A1	B1	B1	B1	B1
	A2	A2	A2	A2	B2	B2	B2	B2
	A3	A3	A3	A3	B3	B3	B3	B3
	A4	A4	A4	A4	B4	B4	B4	B4
	A5	B5	A5	A5	B5	B5	B5	B5

DVM (Dual Vector Mode)

NOTE DVM is also known as the *VX250* mode.

Both DVM Normal mode and MuxMode provide I/0 data in the RZ, RO, and Window strobe formats in the normal mode and MuxMode:

- DVM Normal mode properties:

  64 wavesets for each pin across the 4 timing markers.

  Maximum DUT rate is 312.50 Mb/s, 4 events per DUT cycle.

  Provides RZ, RO, and SBC with full-formatted I/O data.

  DUT rate equals the base tester rate.

  RPT micro-instructions supported every DUT cycle.

  Per-pin micro-instructions supported every DUT cycle.

  Micro-instructions for CPM branching and period switching are supported on every second DUT cycle.

- DVM MuxMode properties:

  64 wavesets for each pin across the 4 timing markers.

  Maximum DUT rate is 625 Mb/s,4 events per DUT cycle.

  DUT rate is twice the base tester rate.

  RPT micro-instructions supported every second DUT cycle.

  Per-pin micro-instructions supported on all DUT cycles.

  Micro-instructions for CPM branching and period switching are supported on every fourth DUT cycle.

The example listed in Table 2.9 uses:

- 6 characters—H, L, 1, 0, X, and F—provide 6 waveforms (A0/B0 to A5/B5) per pin for each Timing Generator T1A/B to T4A/B.
- 10 global time sets (6-bits per pin x 10 waveforms/time sets = 60 bits, limit is 64 bits).
- 5 global period values.

Table 2.9: Example: Dual Vector Mode

Timing Generator	T1A	T2A	T3A	T4A	T1B	T2B	T3B	T4B
Per Pin Waveset Pointers	A0	A0	A0	A0	B0	B0	B0	B0
	A1	A1	A1	A1	B1	B1	B1	B1
	A2	A2	A2	A2	B2	B2	B2	B2
	A3	A3	A3	A3	B3	B3	B3	B3
	A4	A4	A4	A4	B4	B4	B4	B4
	A5	B5	A5	A5	B5	B5	B5	B5

DMM (Dual Marker Mode)

Both DMM Normal mode and MuxMode provide I/O data in the NRZ, RZ, RO, and Edge strobe formats:

- DMM Normal mode properties:

  Maximum DUT rate is 625 Mb/s, 2 events per DUT cycle.

NRZ, RZ, RO, and Edge strobe, full-formatted I/O data.

DUT rate is twice the base tester rate.

`RPT` micro-instructions supported every second DUT cycle.

Per-pin micro-instructions supported every second DUT cycle.

Micro-instructions for CPM branching and period switching are supported on every fourth DUT cycle.

- DMM `MuxMode` properties:

  Maximum DUT rate is 1.25 Mb/s, 4 events per DUT cycle.

  DUT rate is four times the base tester rate.

  `RPT` micro-instructions supported every fourth DUT cycle.

  Per-pin micro-instructions supported every second DUT cycle.

  CPM branching and period switching supported every fourth DUT cycle.

The example listed in Table 2.10 and Table 2.11 uses:

- 5 characters: H, L, 1, 0, and X. Note that Don't Care (X) cycles are translated into Driver Low by the Waveform Compiler.
- Bits are dedicated to pins, 2 bits per pin, one data and one I/O per Timing Generator T1A/B to T4A/B.
- 16 global time sets.
- 4 global period values: G1 to G4. Each is shared by all Timing Generators.

Table 2.10: Example: Dual Marker Mode

Timing Generator	T1A	T2A	T3A	T4A	T1B	T2B	T3B	T4B
Data Bit Per Marker Pair	A0	A0	A2	A2	B0	B0	B2	B2
I/O Bit Per Marker Pair	A3	A3	A5	A5	B3	B3	B5	B5
Waveset Address: Global Bits Across Both Pins	G1	G1	G1	G1	G1	G1	G1	G1
	G2	G2	G2	G2	G2	G2	G2	G2
	G3	G3	G3	G3	G3	G3	G3	G3
	G4	G4	G4	G4	G4	G4	G4	G4

Table 2.11: Example: DMM I/O Pins

I/O Bit Per Marker Pair	Data Bit Per Marker Pair	M1	M2
D	0	Drive On	Drive Low
D	1	Drive On	Drive High
S	0	Drive Off	Compare Low
S	1	Drive Off	Compare High

TMM (Triple Marker Mode)

Both TMM Normal mode and TMM MuxMode provide independent I/O data on three markers for each pin:

- TMM Normal mode properties:

  Maximum DUT rate is 833 Mb/s.

  DUT rate is three times the base tester rate.

  I/O rate—not applicable in this mode.

  RPT micro-instructions supported every third DUT cycle.

  Per-pin micro-instructions supported every third DUT cycle.

  Micro-instructions for CPM branching and period switching supported on every sixth DUT cycle.

  One TMM normal mode example listed in Table 2.12 uses:

5 characters: H, L, 1, 0, and X. Note that Don't Care (X) cycles are translated into Driver Low by the Waveform Compiler. Individual I or O must be separated by 2 cycles, minimum.

Bits are dedicated to pins, 2 bits per pin: one data and one I/O per Timing Generator T1A/B to T3A/B. On TG4A and TG4B, note that T4A0/B0 and T4A1/B1 monitor the activity of other I/O marker bits; refer to Table 2.13.

16 global time sets.

4 global period values: G1 to G4. Each is shared by all Timing Generators.

Two examples of the TMM Normal mode are shown in Figure 2.46 and Figure 2.47.

- TMM `MuxMode` properties:

  Maximum DUT rate is 1.4 Mb/s.

  DUT rate is six times the base tester rate.

  I/O rate is one-half the DUT rate.

  `RPT` micro-instructions supported every sixth DUT cycle.

  Per-pin micro-instructions supported every third DUT cycle.

  Micro-instructions for CPM branching and period switching supported every twelfth DUT cycle.

- In `MuxMode`, the A3 and B3 data bits are swapped in the Timing Generator to aid in interleaving the TGs. On TG4A and TG4B, note that T4A0/B0 and T4A1/B1 monitor the activity of other I/O marker bits; refer to Table 2.14.

  In the `MuxMode` example shown in Figure 2.48, each of the three Timing Generators, TG1, TG2, and TG3, has two independent bits of data. TG4 uses the I/O bit of these 3 markers to know when to fire.

  In another TMM `MuxMode` example shown in Figure 2.49, uses phase-shifted data, which is the data coming into the cycle in the output state.

.

Table 2.12: Example, Triple Marker Mode

Timing Generator	T1A	T2A	T3A	T4A	T1B	T2B	T3B	T4B
Per-Pin Data Bits	A0	B0	A0	T4A0	B0	B0	B0	T4B0
Per-Pin I/O Bits	A3	B3	A3	T4A1	B3	A5	B5	T4B1
Waveset Address: Global Bits Across Both Pins	G1	G1	G1	G1	G1	G1	G1	G1
	G2	G2	G2	G2	G2	G2	G2	G2
	G3	G3	G3	G3	G3	G3	G3	G3
	G4	G4	G4	G4	G4	G4	G4	G4

Table 2.13: TG4 Lookup Table, Normal Mode

3 I/O Bits from 3 Markers			State Number	4 I/O Actions for TG4		
A3	A4	A5		T4A0	T4A1	Phase
D	D	D	0	0	0	n/a
D	D	S	1	1	1	C
D	S	D	2	1	0	B (not valid)
D	S	S	3	1	0	B
S	D	D	4	0	1	A
S	D	S	5	0	1	A (not valid)
S	S	S	6	0	1	A
S	S	S	7	0	1	A

Table 2.14: TG4 Lookup Table, MuxMode

3 I/O Bits from Each Pin						State Num	4 I/O Actions for TG4				
A4	B3	A5	B4	A3	B5		T4A0	T4B0	T4A1	T4B1	Phase
TG2A	TG1B	TG3A	TG2B	TG1A	TG3B						
D	D	D	D	D	D	0	0	0	0	0	n/a
D	D	S	D	D	S	1	1	1	0	1	C
D	S	D	D	S	D	2	1	0	1	0	B (not valid)
D	S	S	D	S	S	3	1	0	1	0	B
S	D	D	S	D	D	4	0	1	1	1	A
S	D	S	S	D	S	5	0	1	1	1	A (not valid)
S	S	D	S	S	D	6	0	1	1	1	A
S	S	S	S	S	S	7	0	1	1	1	A

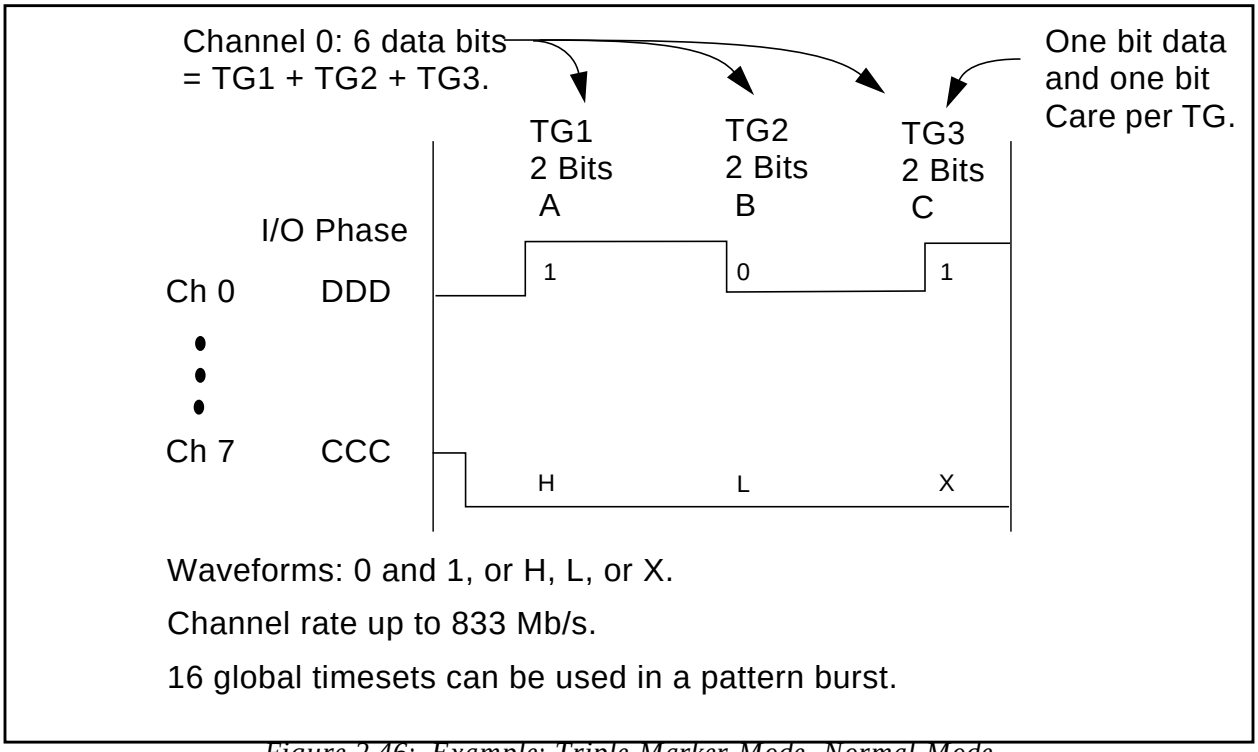


Figure 2.46: Example: Triple Marker Mode, Normal Mode

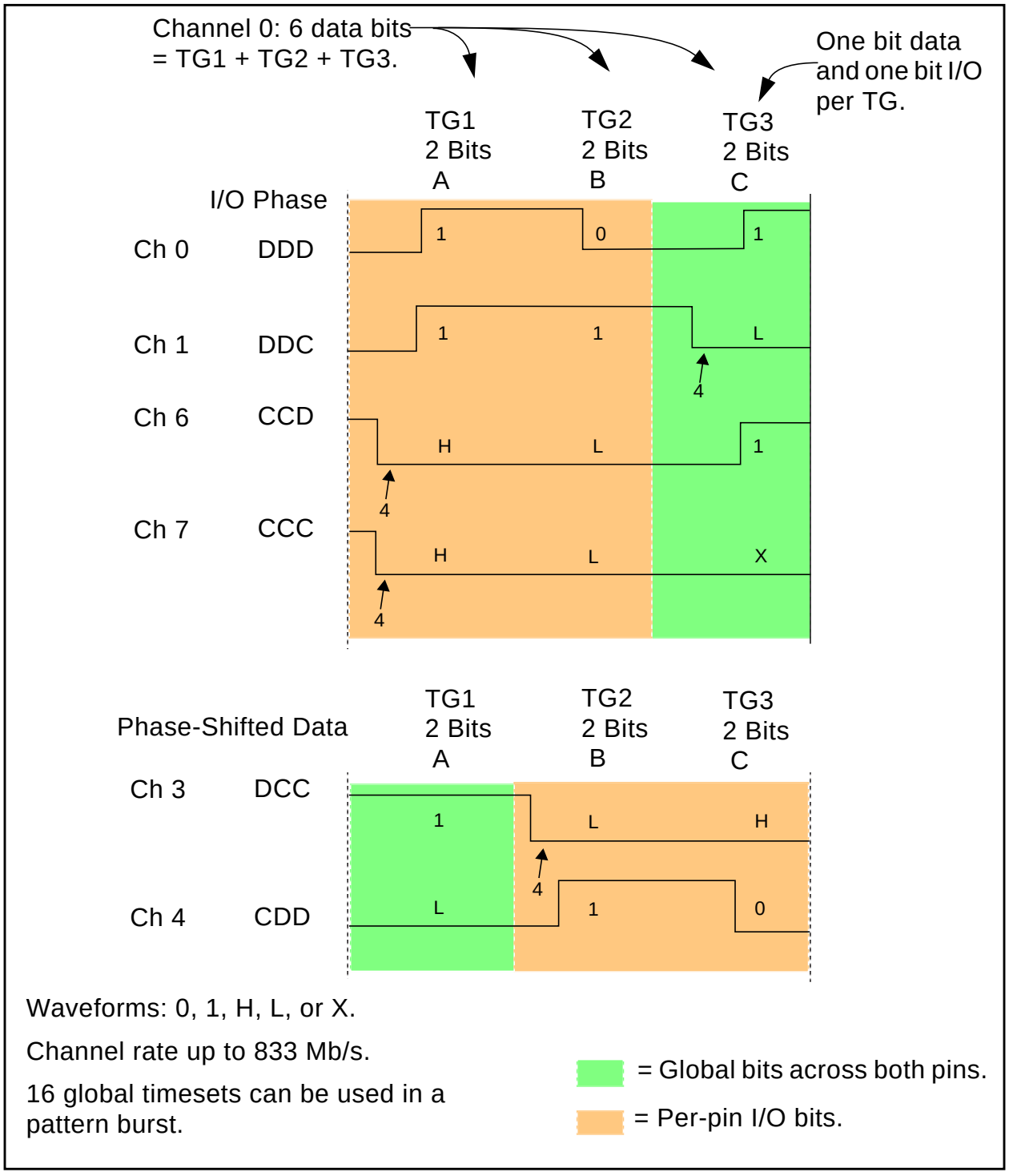
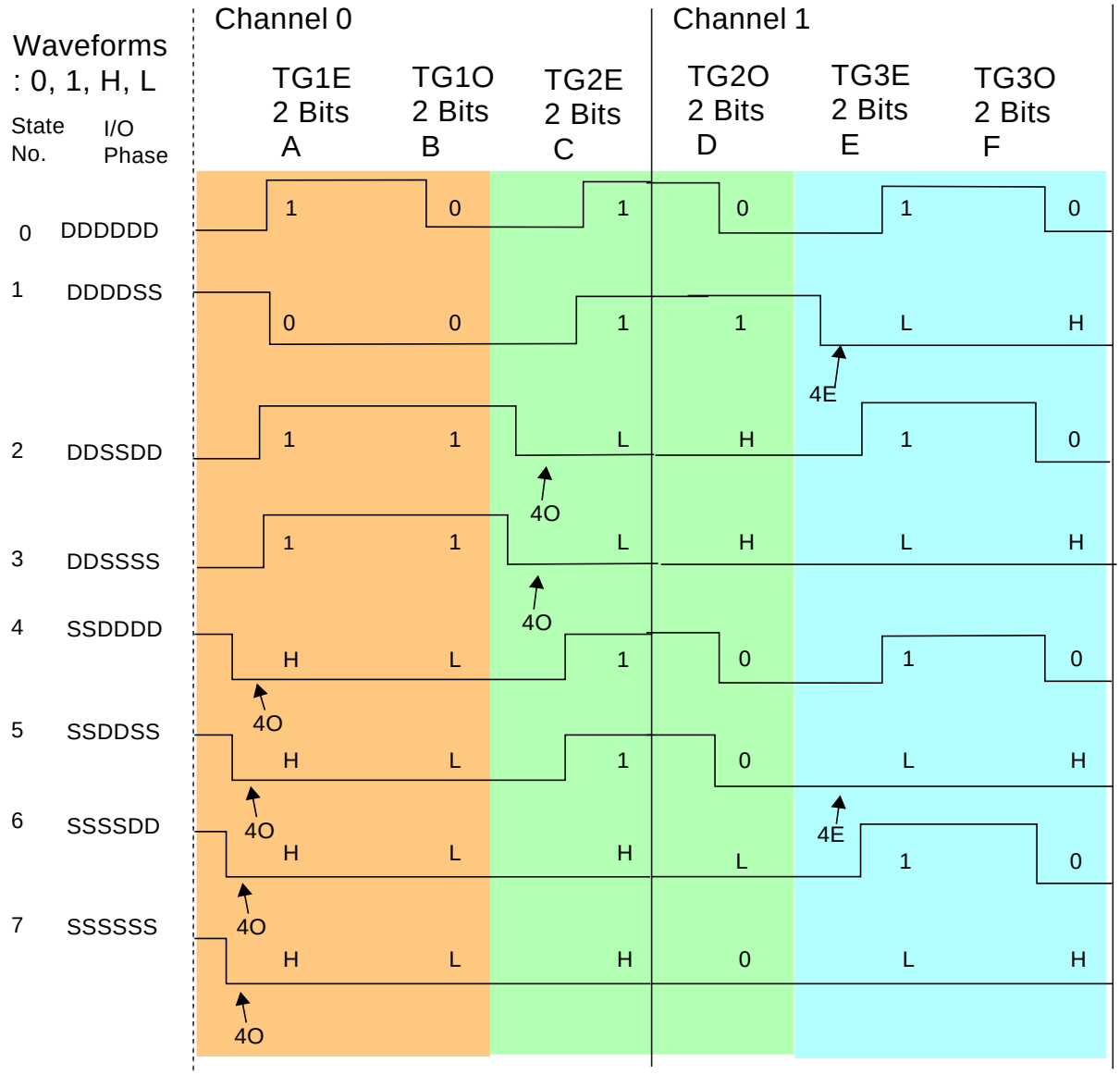


Figure 2.47: Triple Marker Mode Using Driver Low and Strobe, Normal Mode, I/O Phase and Phase Shifted Data

Channels 0 and 1: one bit data and one bit I/O per TG; 6 bits per channel.

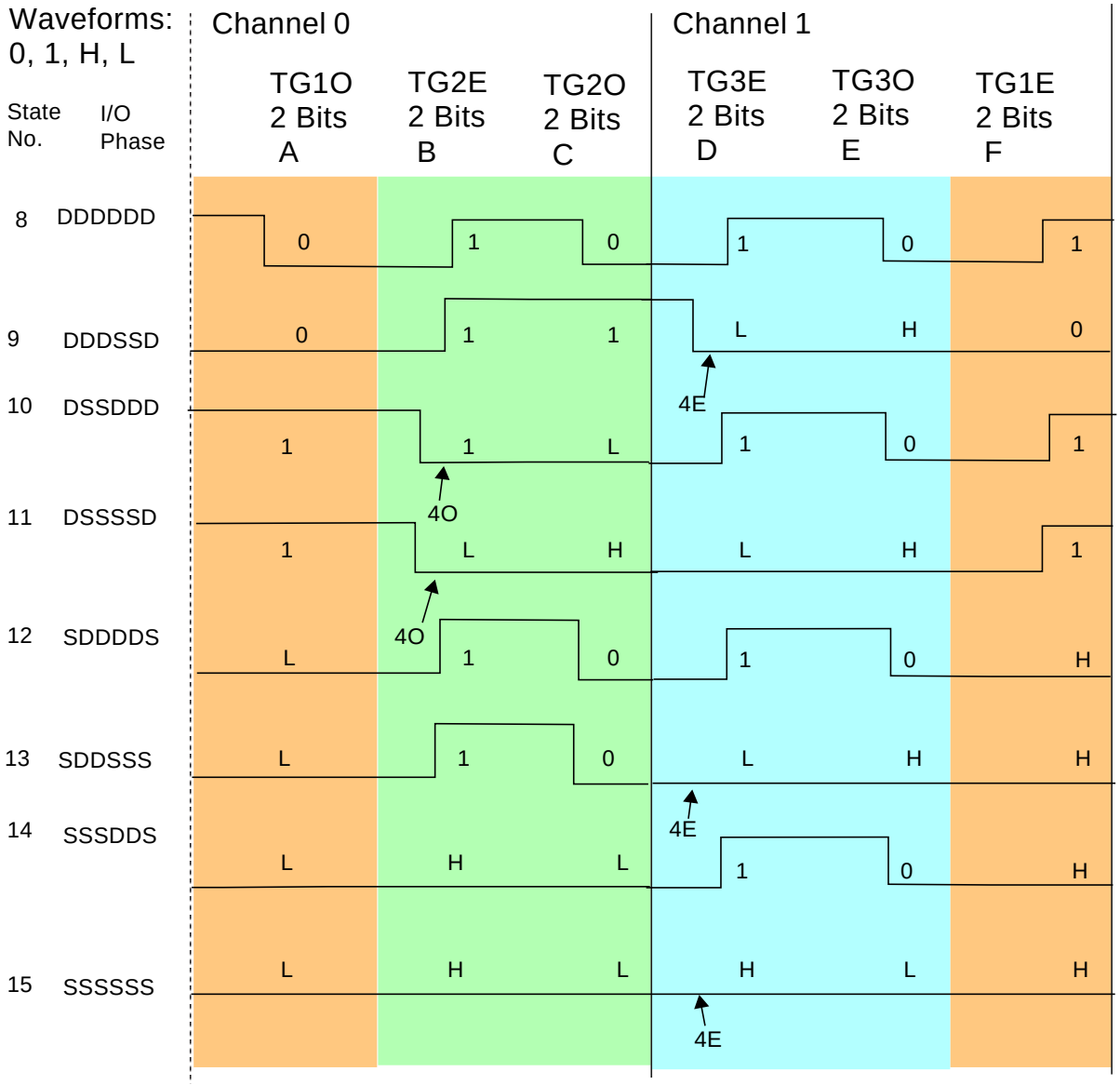


Each channel runs up to 833 Gb/s. Current VX4 hardware is limited to 1.4 Gb/s.

Cell 2 followed by Cell 3 above shows the retrigger case.

Figure 2.48: Triple Marker Mode Using Driver Low and Strobe, MuxMode

Channels 0 and 1: one bit data and one bit I/O per TG; 6 bits per channel.



Each channel runs up to 833 Gb/s. Current VX4 hardware is limited to 1.4 Gb/s.

Maximum frequency is limited by the retrigger of the marker producing the I/O edge. Retrigger equals 2.4 ns, which equals 4 DUT bit cells; thus, the maximum bit cell is 600 ps or 1.666 Gb/s.

Figure 2.49: Triple Marker Mode, Phase Shifted Data, MuxMode

Quad Marker Mode (QMM)

Both QMM Normal mode and MuxMode provide independent input or output data in a NRZ or Edge strobe format:

- QMM Normal mode:

  Maximum DUT rate is 1.25 Gb/s.

  DUT rate equals four times the base tester rate.

  RPT micro-instructions supported every fourth DUT cycle.

  Per-pin microinstructions supported every fourth DUT cycle.

  CPM branching and period switching supported every eighth DUT cycle.

- QMM MuxMode:

  Maximum input or output rate LVM is 2 Gb/s (PEIC3 on VX IV alpha), 2.4Gb/s (PEIC4 pm VX IV beta).

  DUT rate equals eight times the base tester rate.

  RPT micro-instructions supported every eighth DUT cycle.

  Per-pin micro-instructions supported every fourth DUT cycle.

  Micro-instructions for CPM branching and period switching supported every sixteenth DUT cycle.

The example listed in Table 2.15 uses:

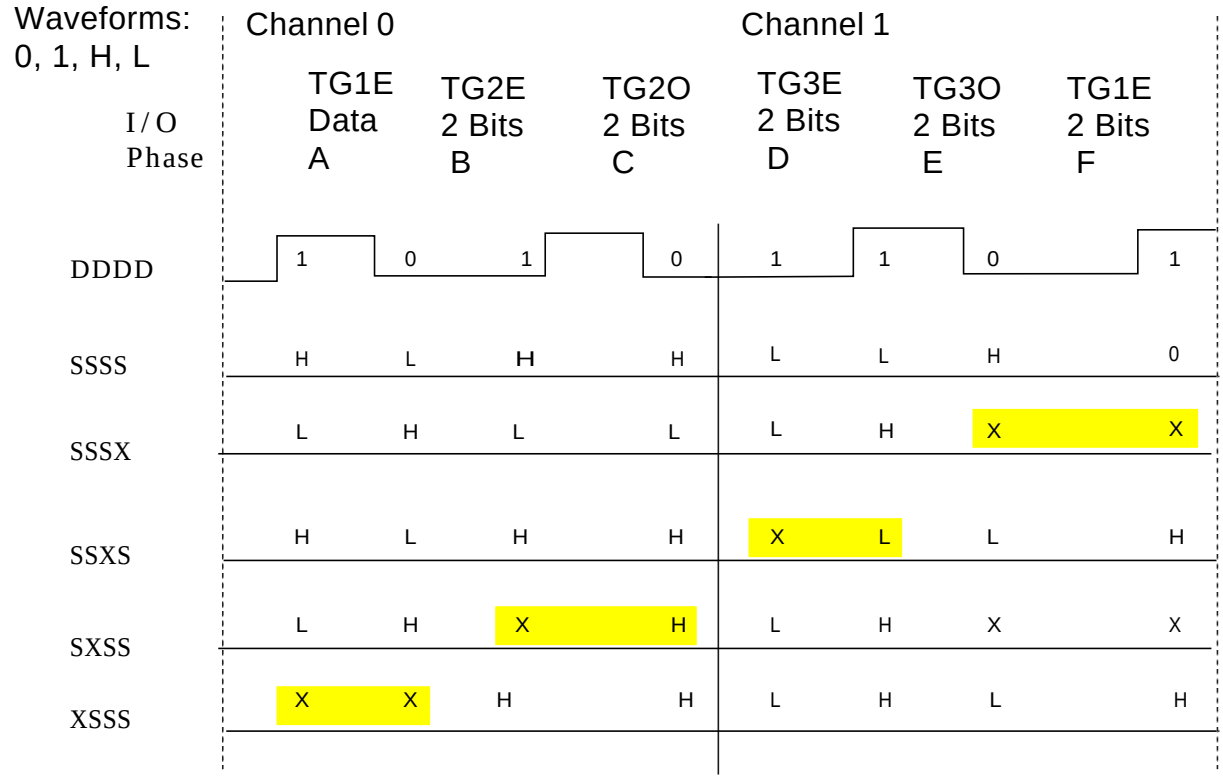
- 2 characters: either H and L or 1 and 0.
- Bits are dedicated to pins, 1 data bit per pin, per Timing Generator T1A/B to T4A/B.
- 16 global time sets.
- 4 global period values: G1 to G4. Each is shared by all Timing Generators.

Another QMM example is shown in Figure 2.50. This figure shows that while all signal combinations for inputs are supported; not all signal combinations for outputs are supported because the masking bit is shared by both input and output modes.

Table 2.15: QMM Input or Output Data, NRZ or Edge Strobe

Timing Generator	T1A	T1B	T2A	T2B	T3A	T3B	T4A	T4B
Per-Pin Data Bits	A0	B0	A1	B1	A2	B2	A3	B3
Waveset Address: Global Bits Across Both Pins	G1	G1	G1	G1	G1	G1	G1	G1
	G2	G2	G2	G2	G2	G2	G2	G2
	G3	G3	G3	G3	G3	G3	G3	G3
	G4	G4	G4	G4	G4	G4	G4	G4

Channels 0 and 1: one bit data and one bit I/O per TG; 6 bits per channel, 312.5 Mb/s.



Input or output pin = low-speed definition.
X in yellow box must in phase-aligned cycle pairs and follow the rules listed below:

Output Mode

Mask		PA	PB	PA	PB	Note
0	1	0	0	L	L	All High and Low Combinations Supported
1	1	0	1	L	H	
2	1	1	0	H	L	
3	1	1	1	H	H	
4	0	0	0	H	H	
5	0	0	1	X	H	Don't Care Combination Supported
6	0	1	0	X	X	
7	0	1	1	X	X	
				H	X	Not Supported: Use XX
				L	X	

Input Mode: All 4 data states supported.

PA	PB	PA	PB
0	0	0	0
0	1	0	1
1	0	1	0
1	1	1	1

Figure 2.50: Quad Marker Mode, Mux I/O