Keysight N5990A-301 PCI Express Link Training Suite

Language Guide

KEYSIGHT TECHNOLOGIES

Notices

© Keysight Technologies 2017

No part of this manual may be reproduced in any form or by any means (including electronic storage and retrieval or translation into a foreign language) without prior agreement and written consent from Keysight Technologies as governed by United States and international copyright laws.

Trademarks

Manual Part Number

N5990-91210

Manual Edition

Edition 4.0, July 2017

Keysight Technologies Deutschland GmbH
Herrenberger Strasse 130,
71034 Böblingen, Germany

Technology Licenses

The hardware and/or software described in this document are furnished under a license and may be used or copied only in accordance with the terms of such license.

U.S. Government Rights

The Software is “commercial computer software,” as defined by Federal Acquisition Regulation (“FAR”) 2.101. Pursuant to FAR 12.212 and 27.405-3 and Department of Defense FAR Supplement

(“DFARS”) 227.7202, the U.S. government acquires commercial computer software under the same terms by which the software is customarily provided to the public. Accordingly, Keysight provides the Software to U.S. government customers under its standard commercial license, which is embodied in its End User License Agreement (EULA), a copy of which can be found at http://www.keysight.com/find/sweula. The license set forth in the EULA represents the exclusive authority by which the U.S. government may use, modify, distribute, or disclose the Software. The EULA and the license set forth therein, does not require or permit, among other things, that Keysight: (1) Furnish technical information related to commercial computer software or commercial computer software documentation that is not customarily provided to the public; or (2) Relinquish to, or otherwise provide, the government rights in excess of these rights customarily provided to the public to use, modify, reproduce, release, perform, display, or disclose commercial computer software or commercial computer software documentation. No additional government requirements beyond those set forth in the EULA shall apply, except to the extent that those terms, rights, or licenses are explicitly required from all providers of commercial computer software pursuant to the FAR and the DFARS and are set forth specifically in writing elsewhere in the EULA. Keysight shall be under no obligation to update, revise or otherwise modify the Software. With respect to any technical data as defined by FAR 2.101, pursuant to FAR 12.211 and 27.404.2 and DFARS 227.7102, the U.S. government acquires no greater than Limited Rights as defined in FAR 27.401 or DFAR 227.7103-5 (c), as applicable in any technical data.

Warranty

THE MATERIAL CONTAINED IN THIS DOCUMENT IS PROVIDED "AS IS," AND IS SUBJECT TO BEING CHANGED, WITHOUT NOTICE, IN FUTURE EDITIONS. FURTHER, TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, KEYSIGHT DISCLAIMS ALL WARRANTIES, EITHER EXPRESS OR IMPLIED WITH REGARD TO THIS MANUAL AND ANY INFORMATION CONTAINED HEREIN, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. KEYSIGHT SHALL NOT BE LIABLE FOR ERRORS OR FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH THE FURNISHING, USE, OR PERFORMANCE OF THIS DOCUMENT OR ANY INFORMATION CONTAINED HEREIN. SHOULD KEYSIGHT AND THE USER HAVE A SEPARATE WRITTEN AGREEMENT WITH WARRANTY TERMS COVERING THE MATERIAL IN THIS DOCUMENT THAT CONFLICT WITH THESE TERMS, THE WARRANTY TERMS IN THE SEPARATE AGREEMENT WILL CONTROL.

Safety Notices

CAUTION

A CAUTION notice denotes a hazard. It calls attention to an operating procedure, practice, or the like that, if not correctly performed or adhered to, could result in damage to the product or loss of important data. Do not proceed beyond a CAUTION notice until the indicated conditions are fully understood and met.

WARNING

A WARNING notice denotes a hazard. It calls attention to an operating procedure, practice, or the like that, if not correctly performed or adhered to, could result in personal injury or death. Do not proceed beyond a WARNING notice until the indicated conditions are fully understood and met.

Contents

1 Introduction

Overview of this guide 8

Script Structure 9

General Syntax 9

Script Processing 10

Data Rate Encoding 10

PWM Encoding 11

Data Types 13

Names 13

Numeric Data Types 13

Pattern Data 14

Other Data Types 15

Data Rate Definitions 17

Block Definitions 18

Block References 18

Repetitions 19

Multi-blocks 19

Macros 20

Pattern Distribution 21

8b/10b Encoding 22

Sequence definition 23

Complete Example 24

2 Common Macros

Filling, Padding and Synchronizing 28
- Fill, Pause0, Pause1 28
- Pad, Pad0, Pad1 28
- Sync, Sync0, Sync1 29

Pattern Distribution 30
- SetDistri 30

8b/10b Encoding 31
- ConvertTo8b10b, Disable8b10b 31
- DefineAlignSymbol 31
- DispReset 31

Data Rate and PWM Encoding 32
- Rate, CustomRate 32
- PWM 32

RBS Generation 33
- PRBS, PRBN 33
- HardwarePRBS 34

Error Insertion 35
- FlipNextBit 35
- FlipDisparity 35

3 Defining PCI Express Patterns

PCI Express Symbols 44

PCI Express Macros Overview 45

PCI Express Gen1/Gen2 Macros 46
- ScramblerOn, ScramblerOff 46
- EIOS, EIEOS 46
- SKP, FTS, EIE 46
- TS1, TS2 46
- DS, DB, DU 48

PCI Express Gen3/Gen4 Macros 49

EIOS, EIEOS 49

SKP, FTS, SDS 49

TS1, TS2 49

DS, DB, DU, OSS, OSB, OSU 51

ILT 52

CPH 52

Loopback Pattern Generation 53

4 External Pattern Files

5 Scripting Tips

Repetitions and loops 59

Fulfilling Granularity Restrictions 60

1 Introduction

Overview of this guide / 8
Script Structure / 9
Script Processing / 10
Data Types / 13
Data Rate Definitions / 17
Block Definitions / 18
Sequence definition / 23
Complete Example / 24

This document describes a script language that is intended to be used for defining patterns. Patterns can consist of 1's and 0's, and will eventually be generated by a pattern generator instrument.

Overview of this guide

The N5990A-301 PCI Express Link Training Suite provides a language to edit the generated pattern with PCI Express-common macros. This document describes the syntax, the macros and the possibilities of that language.

This document describes a script language that is intended to be used for defining `patterns`. Patterns can consist of 1's and 0's, and will eventually be generated by a pattern generator instrument.

The bit stream generated by the pattern generator instrument can consist of different patterns, which are organized in **blocks**. The blocks are ordered in a **sequence.** The sequence can generate complex bit streams by referencing blocks multiple times or looping them.

Every pattern generator instrument has restrictions on the pattern blocks. These restrictions are usually of a minimum pattern length and pattern **granularity**. The latter parameter describes the number of bits the pattern length must be an integer multiple of.

A pattern can be defined for multiple **channels,** if the pattern generator instrument supports multiple output channels. In this document, the first channel is referred to as channel 0.

Formatting:

In this document, all code examples are printed using the following formatting:

Inline code: `code example`

Multi-line code examples:

```
code example
code example
code example
```

Data types are highlighted: *data-type.*

Script Structure

The file format is organized in the following way:

```
Datarates:
<rate_1>, <rate_2>, …, <rate_n>;

Blocks:
<block_name_1>: <data_1>, <data_2>, …, <data_m>
@<data_rate>;
<block_name_2>: <data_1>, <data_2>, …, <data_m>
@<data_rate>;
…
<block_name_n>: <data_1>, <data_2>, …, <data_m>
@<data_rate>;

Sequence:
<number_1>: <block_name>, <loop_count>;
<number_2>: <block_name>, <loop_count>;
…
<number_n>: <block_name>, <loop_count>;
LoopTo <number_x>;
```

First, all the data rates that will be used later are defined. Then, the blocks are defined, where each block describes a pattern. Finally, the sequence in which the blocks shall be generated is defined.

General Syntax

The keywords `Datarates:`, `Blocks:` and `Sequence:` define the basic document structure and must appear in the correct order.

White-space is ignored, unless noted otherwise. White-space can be a regular space, a tabulator, or a line-break.

Comments are ignored and can be used to leave notes in the script. Comment text can be placed behind a `#` or `//`; this kind of comment extends until the end of the current line. If a comment text spans several lines, it can be placed between `/*` and `*/`.

Script Processing

When the pattern generator instrument is programmed, the following steps are conducted:

- The script is parsed; if there are any syntax errors, an error message is shown.
- Repetitions and block references are expanded.
- Macros are processed.
- Pattern data is distributed to all available channels.
- The pattern is encoded [to a specific bit rate or pulse-width modulation (PWM)].
- The pattern blocks and the sequence are converted into instrument-specific format and downloaded to the instrument.

Data Rate Encoding

In many cases, several different data rates must be generated; either the data rate is switched or different channels run at different data rates. Common pattern generator instruments cannot handle this. To compensate for this, the patterns are encoded to emulate a specific data rate.

To emulate a lower data rate than the generator data rate, pattern bits are just repeated. Figure 1 on page -10 illustrates how two patterns of different data rates can be generated by doubling every bit of the slower pattern.

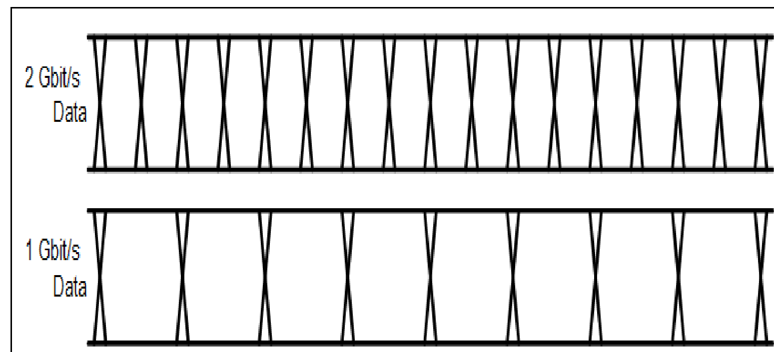

Figure 1 Data rate encoding (factor 2:1)

If the ratio of the data rates is not an integer number, the slower pattern is generated with different bit length. For example, if the generator runs at 8 Gbit/s and a 2.5 Gbit/s pattern should be generated, a 3-3-4-3-3 scheme is utilized (see Figure 2 on page -11). Note that this scheme introduces a small amount of jitter.

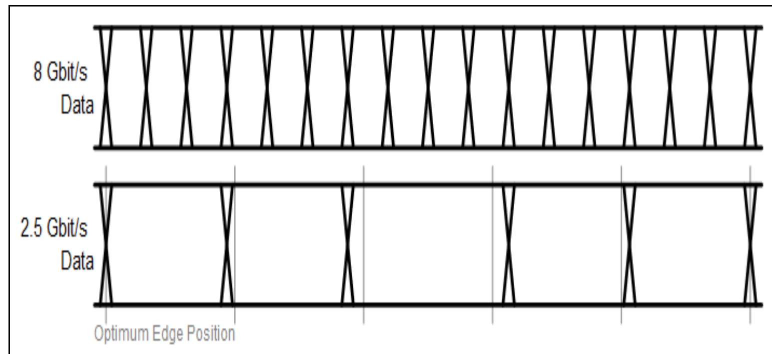

Figure 2 Data rate encoding (factor 16:5)

PWM Encoding

Instead of plain pattern encoding, a PWM encoding scheme can be utilized. A PWM waveform is defined by three parameters: the data rate, the inversion, and the duty cycle (DC) of the logical bits. Figure 3 on page -11 shows the impact of the inversion and the duty cycle (zero ratio, that is, the duty cycle of a logical zero).

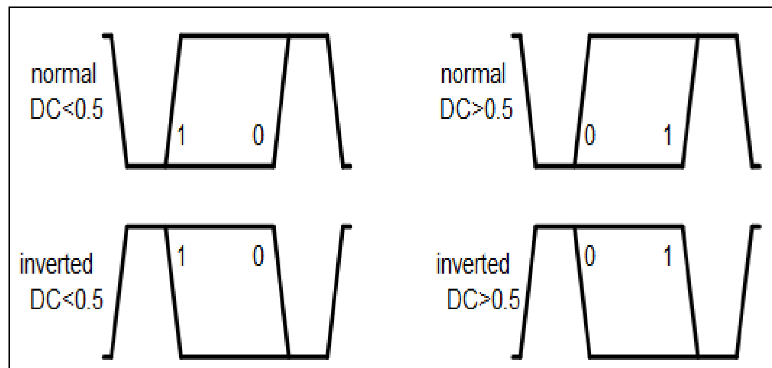

Figure 3 PWM Parameters

Figure 4 on page -12 shows an example of a PWM waveform compared to data rate encoded data.

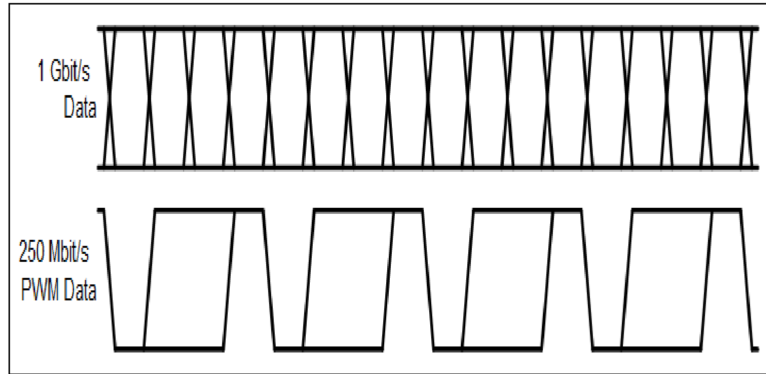

Figure 4 Example PWM waveform

Data Types

The script language knows several different data types. The latter part of this documentation will refer to these data types.

Names

A name can define a block, a macro, or different kinds of arguments. A name can consist of letters, digits and underscores (_), where the first symbol is not allowed to be a digit. Names are always case-sensitive.

Examples:

`DemoName`, `myMacro1`, `_123`

Numeric Data Types

The simplest numeric data type is *Integer*. The value can be given in decimal, binary or hexadecimal. In decimal representation, an optional sign is allowed. Binary numbers must be preceded by **`0b`**, hexadecimal numbers must be preceded by **`0x`**.

Examples:

`123`, `+100`, `-33`,`0b101`, `0xFF`

Rational numbers are referred to as floating-point or *float* numbers. A float number must be given in decimal representation. It can have an optional sign. An exponent can be given in scientific exponential notation, or as an SI-prefix.

The exponential notation uses the letter “e” or “E” as a synonym for “times ten to the power of”.

If an SI-prefix is used instead, there may be an additional single space between the number and the SI-prefix. Allowed SI-prefixes are **`a, f, p,`** `n`, `u`, `µ`, `m`, `k`, `M`, `G`, `T`, `P` and `E` (where `u` equals `µ`).

Examples:

`123`, `3.141`, `1e4`(=10,000),`-5e-3`(= -0.005), `3m`(= 0.003), `+0.1k` (=100)

Some macros might accept float numbers with units. These types are referred to as *duration*, *datarate, frequency*, *ui* (unit interval) and *si* (symbol interval). They follow the same rules as float numbers, except that they may be suffixed by a specified unit.

The unit for *duration* is `s`, the unit for *datarate* is `bps` (= bits per second, bit/s), the unit for *frequency* is `Hz`, the unit for *ui* is `UI`, the unit for *si* is `SI`. In the exponential notation, there may an additional single space between the exponent and the unit.

Examples:

`123`, `3.141s`, `1Gbps`

Pattern Data

Pattern data is one of the most important parts of this language. A pattern can either be represented as bits, or as symbols, or the pattern can be loaded from a file.

The simplest kind of pattern data is called *rawdata*. This format is represented either as binary bits or as hexadecimal nibbles.

Binary data is preceded by `0b` and must consist solely of zeros and ones. Hexadecimal raw data is preceded by `0x` and must consist solely of the digits `0` through `9` and the letters `A` through `F`, all upper-case.

Hexadecimal rawdata is only accepted in byte granularity, that is, in multiples of two nibbles or eight bits. If an odd number of nibbles is provided, a zero will automatically be padded before the left-most digit. If non-byte-granularity rawdata is required, binary rawdata must be provided.

Examples:

`0b011`, `0xFF`, `0xABC`(= 0x0ABC due to padding), `0x1234`

Rawdata can be repeated with the suffix `n` (lower-case) and a number. This is a short way of repeating binary or hexadecimal symbols.

Examples:

`0b01n5`(= 0b01010101), `0xFFn2`(= 0xFFFF)

Rawdata can also be repeated such that it is applied to all available channels independently (if more than one channel is available) with the suffixe `s` (lower-case) and a number. Details of these mechanisms are provided later in this document.

Examples:

`0b01s2`(= 0b0101 per lane), `0xFFs2`(= 0xFFFF per lane)

Since hexadecimal data is used very commonly, there is a short-hand notation which omits the prefix `0x`. In this case, the hexadecimal digits must have byte-granularity, that is, there must be an even number of digits, and `n` or `s` suffixes are not allowed. Odd numbers of hexadecimal digits without preceding `0x` are never interpreted as hexadecimal data.

Note that this notation can easily be confused with numbers or names, so it should be avoided if there is ambiguity. In such cases, providing the prefix `0x` is recommended.

Examples:

`1234`, `ABCDEF`

For the 8b/10b encoding scheme, the data type *symbol* is provided. A symbol represents a D-character or a K-character.

Examples:

`K28.0`, `D10.2`

The disparity of symbols is automatically tracked and maintained. However, it can be overridden with a disparity symbol, either `+` or `-`. If the disparity symbol `+` is provided, the running disparity is set to +1 before the symbol is encoded. If the disparity symbol `-` is used, the running disparity is set to -1 before the symbol is encoded.

Just like *rawdata*, a *symbol* may have an additional `n` or `s` suffix.

If both, the disparity sign and the repetition suffix are used, the disparity symbol must come first. In that case, the disparity is applied to every symbol separately (which can lead to a disparity error).

Examples:

`K28.0+`,`D10.3-n5`,`K28.5s3`

Pattern data can also be placed in external files. This data type is called *filename*. File names of external pattern files are placed in double-quotes. File names are not allowed to contain double-quotes.

Examples:

`"C:\demo.pat"`

Details about pattern files are described later in this document.

Other Data Types

There are some other data types that are used by macros. The data type *boolean* represents a switch that can be either `true` (on, set) or `false` (off, unset).

Examples: `true`, `false`

The data type *option* represents an element from a set of names. Its meaning and the available names depend on the context.

Examples:

`flag`, `Value`, `_123`

The data type *text* represents any kind of data in single-quotes. Its meaning depends on the context. The data is allowed to not contain single-quotes.

Examples:

`'any kind of data'`,`'123'`,`'$'`,`'/'`

Data Rate Definitions

This section starts with the keyword `Datarates:`, followed by one or multiple comma-separated data rates, and it ends with a semicolon. The data rates are of the type *datarate*.

The data rate definition section can be omitted entirely. In that case, a set of protocol-specific default data rates are used.

Examples:

```
Datarates: 1.5e9bps, 3000000000, 6G;
```

When the data rates are specified, they are internally indexed starting from one. Therefore, the data rate index 1 refers to 1.5 GBit/s, index 2 refers to 3 GBit/s and index 3 refers to 6 GBit/s.These indexes are used when a data rate is assigned to a block later.

Data rates can be specified in any order. However, the numbering is always in the order they are specified.

Block Definitions

Blocks define the pattern data that will be sent to the generator. The order in which the blocks are transmitted is defined by the sequence, which is explained later in this document.

This section starts with the keyword `Blocks:`. Each block starts with a user-defined name, then a colon, and a series of pattern data, separated by commas. An optional data rate index, preceded by the '@' sign, can follow. A semicolon finishes the block.

The pattern that is represented by a block can be defined with one or more of the following items:

- pattern data: *rawdata*, *symbol*, *filename*.
- macros.
- references to other blocks.
- repetitions.
- multi-blocks.

All elements are comma-separated, with the exception of pattern data. Commas between *rawdata*, *symbol* and/or *filename* elements can be omitted.

Example of a simple block definition section:

```
Blocks:

block_1: 0xAA, 0xBB, 0xCC, 0xDD, 0xEn2; // equals
0xAABBCCDD0E0E

block_2: "C:\Pattern Files\Test.pat" @2; // data from a
file
```

Block References

Blocks can also be used to define commonly used patterns, which can be used in other blocks. The block reference must be in a block that is defined after definition of the referenced block.

Example:

my_pattern: 0xAA, 0xBB

```
block_1: my_pattern, 0xCC;       // equals 0xAABBCC

block_2: 0x00, my_pattern, 0x11; // equals 0x00AABB11
```

Repetitions

A simple way to repeat parts of a pattern is to use the repetition syntax. It consists of a positive *integer* value, representing the repetition count and the pattern to be repeated, in curly brackets. The pattern data inside a repetition can be everything a normal block can contain (thus, repetitions can be nested).

Note that this syntax only repeats the pattern bits, thus consuming pattern memory. Refer to the `Sequence:` script section defined in "Complete Example" on page 24 for details about looping.

Repetitions are generated before any further processing. This means, for example, that the running disparity of symbols is tracked properly.

Example:

```
block_1: 2{K28.5, D0.0}; // equals K28.5, D0.0, K28.5, D0.0
```

Multi-blocks

Multi-blocks allow patterns to be defined for each channel independently. Each multi-block is encapsulated in square brackets. Inside the brackets, there is a list of data assigned to one or more channels: `[<channels>: <data>; <channels>: <data>; …]`. The pattern data inside a multi-block can be everything a normal block can contain (thus, multi-blocks can be nested).

The channel specification can be a single channel, multiple comma-separated channels, a range of channels (two numbers with a dash in between), or the keyword `default`. Channel indices are zero-based.

When multiple channels are grouped (for instance, by using the index `0-1`), these channels are treated as compound. This means that the given data is distributed among those channels. When the `default` keyword is used, the given data is applied to all these channels separately.

Examples:

```
// static 1 in channel 0, clock pattern on channel 1

block_1: [0: 0xFn10; 1: 0b01n40];

// 0xABCD distributed to channels 1 and 2, 0x00 on all
other channels

block_2: [1-2: 0xABCD; default: 0x00];
```

It is recommended that the data streams of all channels are padded, so that they are all equal in length. The `Pad()` macro can be used for this purpose.

If data is specified for channels that don't exist, for example channel “3” in a two-channel-setup, the superfluous channel data is ignored.

Macros

Macros provide simplified access to complex patterns or functions. There are macros that can be used to define patterns, and there are macros that control the pattern processing flow.

To use a macro (to call it), the macro name is typed, followed by parentheses. Many macros have one or more parameters. If a macro has parameters, you can assign arguments to these parameters, which control the operation conducted by the macro. Arguments can be passed either by typing their value (Example: `42` as an argument for an *integer* parameter), or by typing the parameter name, followed by an equation sign and the argument value.

Example: The “Fill” macro generates a stream of a specified pattern to span a specified amount of time. It is documented as

```
Fill(t=<duration>, Pattern=<rawdata>)
```

This means that the macro name is `Fill`, and it has two parameters `t` and `Pattern`. An argument for `t` must be of the type *duration*, an argument for `Pattern` must be of the type *rawdata*.

If arguments are passed without a name, they must be in the order in which they are defined. Named arguments can be in any order. Note that the name of a parameter is case-sensitive, so the call `Fill(pattern=0b0)` fails because `Pattern` was written in lower-case.

Many macros have optional parameters, that is, specific arguments, which can be omitted. In that case, a specified *default* value is assumed. Optional parameters are documented in square brackets.

Example:

The macro `Pad([PaddingPattern=<rawdata>])` can be invoked with an argument for `Padding Pattern`, but it can also be invoked without parameters.

Example:

The `Fill` macro can be called in the following ways (which are all equivalent):

```
block_1: Fill(1m, 0b0);

block_2: Fill(t=1e-3, Pattern=0b0);

block_3: Fill(Pattern=0b0, 1m);
```

Most macros allow parameters of type `boolean` to be given as a flag. This means that omitting the argument implicitly means that the argument is `false,` whereas writing the parameter name instead of a value implicitly means that the argument is `true`.

Example:

If there were an artificial macro `DemoMacro(Param=bool)`, calling `DemoMacro()`, `DemoMacro(false)` and `DemoMacro(Param=false)` would be all equal, and mean that `Param` is `false`. Also, calling `DemoMacro(Param)`, `DemoMacro(true)` and `DemoMacro(Param=true)` would be all equal, and mean that `Param` is `true`.

Note that not all parameters are allowed as flags. Some arguments must be given explicitly, so that the macro can dynamically assign a default value if the argument is omitted.

For several macros, arguments of the type *rawdata*, *symbol* and *filename* can consist of multiple parts. These multiple parts can be put together in single-quotes.

Pattern Distribution

In many cases, only one generator channel will be used. However, if multiple channels are used, the pattern must be distributed among all available channels.

By default, all pattern data is distributed byte-wise. This means that the whole binary pattern is split into chunks of eight bits. The first block of eight bits goes to channel 0, the second to channel 1, and so forth, until all channels are handled. Then it starts on channel 0 all over again.

For example, if the pattern `0xAB`, `0x1234`, `0x001122` is generated on a three-channel system, channel 0 will generate the pattern `0xAB00`, channel 1 will generate `0x1211`, and channel 2 will generate the pattern `0x3422`.

Note that this distribution scheme might lead to channel patterns of unequal length; for example, if a ten-byte pattern is given for a three-channel system, channel 0 will be four bytes in length, whereas channel 1 and channel 2 will be only three bytes in length. The `Sync` macro can be used to bring all channels to equal length.

The default granularity of eight bits can be overridden with the `SetDistri` macro. It allows a different granularity to be defined. 8b/10b symbols are always distributed with ten-bit granularity.

Pattern data defined with the `s`-suffix is applied to every channel independently. For example, if the pattern `0xAB`, `0xFFs1`, `0xCD` is generated on a two-channel system, channel 0 will generate the pattern `0xABFF`, channel 1 will generate `0xFFCD`.

8b/10b Encoding

The running disparity for 8b/10b encoding is automatically tracked per channel. The disparity can be reset at any point with the `DispReset` macro. Alternatively, a symbol with an explicit disparity can be given.

The disparity is only tracked over valid 8b/10b symbols. This means that data explicitly given as *symbol* data, tracks disparity, and *rawdata* which can be interpreted as K- or D-characters, also tracks disparity. However, if invalid data is given, for instance, a stream of ten zeros, the disparity is lost. The pattern will then be searched for a valid sync word, which can be defined with the `DefineAlignSymbol` macro.

Rawdata can be converted to its 10b representation when enabled by the `ConvertTo8b10b` macro. This functionality splits *rawdata* into chunks of eight bits, then encodes it as D-characters. It can be disabled with the `Disable8b10b` macro.

Note that the running disparity is also tracked among different blocks. However, if a block is used more than once in the sequence, this mechanism fails.

Sequence definition

The sequence section of the script defines the order in which the earlier defined blocks are transmitted by the generator hardware. Blocks can be used more than once.

The sequence section starts with the keyword `Sequence:`, followed by several steps. Each step starts with a step label, then a block name, and an optional comma with a loop count or the "manual" keyword. Each step ends with a semicolon.

The step labels are numeric literals. The numbering scheme is arbitrary. However, the label numbers must be ascending and each label has to be unique.

If no loop count is specified, the block is only transmitted once. If the keyword "manual" is used instead, the block is looped until you break the loop manually. The method of breaking the loop depends on the generator hardware.

At the end, the optional keyword LoopTo, following a label, defines the start of the infinite main-loop. If not specified, the whole sequence is looped infinitely.

Example:

```
1.block_1, manual;

2.block_3;

5.block_2, 3;

LoopTo 2;
```

Using this sequence, the pattern generator hardware will generate the following pattern:

- First, the pattern defined in block_1 is transmitted until you trigger manually.
- Then, the block_3 pattern is transmitted once.
- Finally, the block_2 pattern is transmitted three times.
- As the pattern generator starts looping from step 2, block_3 and block_2 (three times) are repeated infinitely.

Complete Example

The following script shows the complete example of a script:

```
Datarates: 1G;

Blocks:

block1: K28.0+, 511{ D0.0 };

block2: 0b01n128, 0b0n256;

Sequence:

1. block1;

2. block2, 10;

LoopTo 1;
```

This script describes two blocks; the first block (`block1`) consists of 512 symbols, the second block (`block2`) consists of 512 bits of rawdata. The first block is transmitted once, then the second block is transmitted ten times. After that, the generator instrument starts with `block1` again.

`block1` consists of a K-character, which starts at positive disparity. The K- character is followed by 511 D-characters. Note that there are 512 symbols of ten bits each, so `block1` can be generated with any generator instrument with a pattern granularity of 1024 bits or less.

`block2` consists 256 bits of a clock pattern (010101...), followed by 256 zeros. Note that this pattern is generated ten times in a row.

The complete generated pattern will be:

```
0xC2D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B6
2D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8
B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62
D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B
62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B6
2D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8
B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62
D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B
62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B6
2D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8
B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62
D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B
62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
```

```
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B6
2D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8
B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62
D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B
62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B6
2D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8
B62D8B5555555555555555555555555555555555555555555555555555
5555555555550000000000000000000000000000000000000000000000
0000000000000000005555555555555555555555555555555555555555
5555555555555555555555550000000000000000000000000000000000
0000000000000000000000000000005555555555555555555555555555
5555555555555555555555555555555555550000000000000000000000
0000000000000000000000000000000000000000005555555555555555
5555555555555555555555555555555555555555555555550000000000
0000000000000000000000000000000000000000000000000000005555
5555555555555555555555555555555555555555555555555555555555
5500000000000000000000000000000000000000000000000000000000
0000000055555555555555555555555555555555555555555555555555
5555555555555500000000000000000000000000000000000000000000
0000000000000000000055555555555555555555555555555555555555
5555555555555555555555555500000000000000000000000000000000
0000000000000000000000000000000055555555555555555555555555
5555555555555555555555555555555555555500000000000000000000
0000000000000000000000000000000000000000000055555555555555
5555555555555555555555555555555555555555555555555500000000
0000000000000000000000000000000000000000000000000000000055
5555555555555555555555555555555555555555555555555555555555
5555000000000000000000000000000000000000000000000000000000
0000000000
```

2 Common Macros

Filling, Padding and Synchronizing / 28
Pattern Distribution / 30
8b/10b Encoding / 31
Data Rate and PWM Encoding / 32
PWM / 32
RBS Generation / 33
Error Insertion / 35

This section describes the common macros, which are not protocol-specific.

Filling, Padding and Synchronizing

Fill, Pause0, Pause1

The macro `Fill(t=<duration>, Pattern=<rawdata>)` repeats a pattern as often as is required to span a defined duration in time. The duration is specified by the parameter `t`, the pattern to be repeated is specified by the parameter Pattern.

Example:

If `Fill(1ms, 0xFF)` is used at a data rate of 1Gbit/s, the pattern `0xFF` is repeated 125,000 times.

Note that this macro can consume significant amounts of pattern memory, as it repeats the given pattern in memory. However, if the macro is used as the only element in a block definition, and the block is used only once in the sequence, the sequence step loop count will be adjusted to achieve the desired number of repetitions. In that case, the pattern is automatically repeated until the required pattern constraints are met.

The pattern is always repeated at least once, and it is always repeated as a whole. The repetition count is rounded up, so the actual duration can be slightly longer than given in the argument.

The macro `Pause0(t=<duration>)` is short for `Fill(t, 0b0)`.

The macro `Pause1(t=<duration>)` is short for `Fill(t, 0b1)`.

Pad, Pad0, Pad1

The most critical hardware limitations are the minimum pattern length and the pattern granularity. It is cumbersome to achieve these restrictions manually. To simplify this process, padding macros are used.

The macro `Pad([Pattern=<rawdata>])` inserts as many bits of a specified pattern as necessary to meet the pattern constraints; this process is known as "padding". For example, if the pattern granularity was 512 bits, and a pattern of 12 bits was already defined, the Pad macro would insert 500 additional bits.

Padding occurs on all available channels independently. Inside a multi-block, only the channels that are currently defined will be padded.

Padding occurs in places where a Pad macro is used. If multiple Pad macros are used on a single channel, the macro processor decides where padding is to be carried out.

The pattern that is used for padding is defined with the **`Pattern`** parameter. If this parameter is omitted, zeros are used for padding. Note that the given pattern is not guaranteed to be used as a whole; if a single bit of a given eight-bit pattern is sufficient to meet the constraints, only the first bit will be used.

The macro `Pad0()` is a shorthand for `Pad(0b0)`.

The macro `Pad1()` is a shorthand for `Pad(0b1)`.

Sync, Sync0, Sync1

When multiple channels are used, it might be desirable to synchronize the data streams on these channels. The macro `Sync(Pattern=<rawdata>)` serves this purpose. It fills all available channels with the pattern specified by the parameter `Pattern` of the type *rawdata*. It inserts as many bits as are required to bring all channels to the same length.

Example:

```
[0: 0xAB; 1: 0x1234], Sync(Pattern=0b0), 0xFs1
```

In this example, channel 0 is 8 bits in length, and channel 1 is 16 bits in length. It is desired that the zeros of `0xFs1` start at the same time in the bit stream. To achieve this, the `Sync` macro is used to insert zeros at the end of each channel to bring them to equal length. Channel 0 will be padded with eight zeros, channel 1 won't be padded.

Pattern Distribution

SetDistri

The macro `SetDistri(Granularity=<integer>)` changes the distribution granularity. Note that this cannot be done while automatic 8b/10b encoding is active, as that requires ten-bit-granularity.

Example:

`0x1234`, `SetDistri(4)`, `0xABCD`

This pattern on a two-channel system results in `0x12AC` on channel 0, and `0x34BD` on channel 1.

8b/10b Encoding

ConvertTo8b10b, Disable8b10b

The macro **`ConvertTo8b10b()`** enables the automatic 8b/10b encoding feature. While it is enabled, all further *rawdata* elements are converted to D-characters. Note that the distribution granularity cannot be changed while this feature is enabled. It can be disabled with the **`Disable8b10b()`** macro.

DefineAlignSymbol

When the running disparity is lost, the disparity tracking algorithm scans all data for a valid 8b/10b symbol. When this symbol is found, disparity is tracked again. The symbol can be defined with the macro **`DefineAlignSymbol(AlignSymbol=<symbol>)`**. The argument for **`AlignSymbol`** can be any *symbol*, but without disparity sign and without **`n`**- or **`s`**-suffix.

DispReset

The macro **`DispReset([Disparity=<integer>])`** resets the current running disparity to the value specified by the *disparity* parameter. The argument for *disparity* is an integer and can be either**`+1`** or**`-1`**.

Data Rate and PWM Encoding

Rate, CustomRate

Every block has a data rate assigned, either with the ‘@’ symbol and a data rate index, or it is the highest data rate by default. However, the data rate can be changed at any point during the block definition with the macro `Rate(Datarate=<integer|option>).`The argument `Datarate` can be a data rate index (as defined in the `Datarates:` section), or the keyword `max` for the highest data rate, or the keyword `default` for the current block's default data rate.

With the macro `CustomRate(Datarate=<datarate>)`, an arbitrary data rate can be defined. The argument `Datarate` is a *datarate* value.

After such a macro, all data is processed for the specified data rate. There can be multiple `Rate` or `CustomRate` macros per channel.

PWM

Instead of the simple bit-stretching mechanism, which just repeats bits, a pattern can also be transmitted as PWM. The macro `PWM([Datarate=<datarate>], [ZeroRatio=<float>], [Inverted=<bool>], [MinDeviation=<float>], [MaxDeviation=<float>])` defines the PWM characteristics. Every subsequent pattern in the block will be PWM encoded. PWM encoding can be deactivated with a `Rate` or `CustomRate` macro.

The parameter `Datarate` specifies the data rate of the PWM signal. The actual PWM data rate might be different, depending on how well the PWM data rate can be emulated with the generator data rate. The parameters `MinDeviation` and `MaxDeviation` are factors which define how much the actual data rate can deviate from the specified PWM data rate. `MinDeviation=-0.5` means that an actual data rate of 50% the specified data rate is allowed; `MaxDeviation=0.5` means that an actual data rate of 150% the specified data rate is allowed.

By default, the bit zero is represented by zeros followed by ones. If the parameter `Inverted` is set to `true`, a zero is represented by ones followed by zeros.

The parameter `ZeroRatio` specifies the duty cycle of a logical zero. The duty cycle of a logical one is one minus `ZeroRatio`. The argument can be in the range of zero to one, exclusively.

RBS Generation

PRBS, PRBN

The macro `PRBS([Invert=<bool>]`, `[Reverse=<bool>]`, `[Order=<integer>], [Length=<integer>]`, `[Polynomial=<integer>]`, `[Distribute=<bool>]`) generates a 2^n-1 PRBS (Pseudo Random Binary Sequence) pattern using a LFSR (Liner Feedback Shift Register) implementation.

The parameter **`Order`** specifies the PRBS order and can be in the range of 3 to 23. The parameter **`Polynomial`** specifies the PRBS polynomial. If both arguments are omitted, a PRBS-7 is generated. If only the argument for **`Polynomial`** is omitted, a standard polynomial for the specified order is chosen. If only the argument for **`Order`** is omitted, it is determined from the polynomial.

The polynomial is given as a number, interpreted as a bit field representing the exponents of the actual polynomial. The term x^n is omitted; the term x^0 is defined with the most significant bit, the term x^{n-1} is defined with the least significant bit. For example, the polynomial x^7+x^{6+1} can be given as **`Polynomial=0b1000001`** (the left-most "1" represents x^0=1, the right-most "1" represents x^6). Note that the PRBS implementation generates a data stream with one more one-bit than zero-bits.

The parameter **`Length`** specifies the length, in bits, of the generated bit stream. If the argument is omitted, the length is the default run length for the specified order. If an argument is given, the PRBS is cropped or repeated to meet the specified length.

If the argument for **`Inverted`** is **`true`**, the bits of the PRBS are inverted (a zero becomes a one and vice versa).

If the argument for **`Reverse`** is **`true`**, the bits of the PRBS are reverted, that is, the first bit is transmitted last.

If the argument for **`Distribute`** is **`true`**, the PRBS data bits are distributed to all available channels, just like normal *rawdata*. Otherwise, the PRBS data bits are placed on all available channels synchronously (like *rawdata* with an **`s1`** suffix).

Example:

`PRBS(Order=7, Inverted)` generates a PRBS-7 with inverted bits. The length will be 127 bits.

Sometimes, it is desired to have a 2^n PRBS instead of a 2^n-1PRBS. This can be generated with the **PRBN([Invert=<bool>], [Reverse=<bool>], [Order=<integer>], [Length=<integer>], [Polynomial=<integer>], [Distribute=<bool>])** macro. The parameters are the same as for the PRBS macro, but it generates a 2^n PRBS.

To generate a 2^n PRBS, an extra zero-bit is inserted into the original data stream at the longest zero-run, thus generating a DC-balanced pattern.

Note that every PRBS generated with either of these macros consumes pattern memory.

HardwarePRBS

Most generator instruments can generate PRBS in hardware, using a built-in LFSR. The advantage of a hardware PRBS is that is does not consume pattern memory. To generate a PRBS in hardware, the **HardwarePRBS(Order=<integer>, Length=<integer>)**macro can be used. The argument for **Order** determines the PRBS order, the argument for **Length** determines the length of the bit stream in bits.

Note that the **HardwarePRBS** macro must be the only item in a block, and this block cannot be referenced in other blocks. This is because a generator instrument cannot mix memory patterns and LFSR patterns.

Error Insertion

FlipNextBit

The macro `FlipNextBit([Channel=<integer>])` can be used for single-bit error insertion. It can be placed anywhere in a block, and it will flip the next bit in the block (that is, a zero becomes a one or vice versa).

The parameter `Channel` determines the channel number where the bit is flipped. If the argument is omitted, the next bit on each channel will be flipped.

Note that the bit will be flipped before encoding, padding and syncing.

Example: `0x00`, `FlipNextBit()`, `0x00` results in `0x00800`.

FlipDisparity

The macro `FlipDisparity([Channel=<integer>])` can be used for symbol error insertion. It can be placed anywhere in a block, and it will flip the current running disparity in the block (that is, +1 becomes -1 or vice versa).

The parameter `Channel` determines the channel number where the bit is flipped. If the argument is omitted, the next bit on each channel will be flipped.

If a single-bit error instead of a disparity error is to be generated, the `FlipNextBit` macro can be used instead.

3 Defining PCI Express Patterns

PCI Express Symbols / 44
PCI Express Macros Overview / 45
PCI Express Gen1/Gen2 Macros / 46
PCI Express Gen3/Gen4 Macros / 49

To simplify the definition of PCI Express patterns, there are symbols and macros especially designed for PCI Express.

KEYSIGHT TECHNOLOGIES

PCI Express Symbols

In addition to the classic K-characters and D-characters, the following symbol names are recognized: `SKP,FTS,SDP,IDL,COM,EIE,PAD,STP,END,EDB,EDS`. These symbol names are valid wherever K-characters are valid. They can also be combined with an `n`- or `s`-suffix

PCI Express Macros Overview

There are several PCI Express specific macros that allow generating ordered sets and other specific patterns with the call of a single macro. All PCI Express macros are handled before the common macros are handled.

Most macros differ between PCI Express generations. For example, Gen1 and Gen2 use 8b/10b encoding, whereas Gen3 and Gen4 use 128b/130b encoding. The set of available macros depends on the current data rate. Recognized data rates are:

Block data rate	Effect
2.5 Gbit/s ± 300 ppm	PCI Express Gen1 macros available, 8b/10b encoding used
5.0 Gbit/s ± 300 ppm	PCI Express Gen2 macros available, 8b/10b encoding used
8.0 Gbit/s ± 300 ppm	PCI Express Gen3 macros available, 128b/130b encoding used
16 Gbit/s ± 300 ppm	PCI Express Gen4 macros available, 128b/130b encoding used

If PCI Express macros are used at different data rates, the macro `PCIeGen(Gen=<integer>)` can be used to declare the rest of the current block as a specific PCI Express generation. The argument `Gen` can be 1, 2, 3, or 4.

Note that it is always recommended to define a block for PCI Express, only by using macros. Raw data can be inserted, but that doesn't advance the PCI Express scrambler algorithm. Therefore, using raw data amongst PCI Express macros destroys the data integrity of a PCI Express bit stream.

Note that during macro processing, the macros `DispReset()` and `SetDistri()` are automatically placed at the beginning of the first block where PCI Express macros are used. This ensures that the disparity starts at negative disparity (according to spec), and that the following data is distributed onto the lanes (channels) in the correct word size (10-bit for Gen1/2, 130-bit for Gen3). These macros are also placed, for example, after the data rate is changed.

PCI Express Gen1/Gen2 Macros

The following sections describe all macros that are available in Gen1 and Gen2.

ScramblerOn, ScramblerOff

Scrambling can be fully deactivated with the macro `ScramblerOn().` Scrambling is enabled again with the `ScramblerOff()` macro. While scrambling is disabled, no D- characters are scrambled, and the scramblers are not advanced.

EIOS, EIEOS

The `EIOS()` macro generates an Electrical Idle Ordered Set. The `EIEOS()` macro generates an Electrical Idle Exit Ordered Set. The `EIEOS` macro is only available for Gen2.

SKP, FTS, EIE

The `SKP([n=<integer>])` macro generates a Skip Ordered Set. The parameter `n` determines the number of SKP symbols. The default value for this parameter is 3.

The `FTS()` macro generates a Fast Training Sequence.

The `EIE([n=<integer>])` macro generates multiple EIE symbols. The parameter `n` determines the number of EIE symbols. If the argument for `n` is omitted, four EIE symbols are generated. The `EIEOS` macro is only available for Gen2.

TS1, TS2

The `TS1([AutonomousChange=<bool>],[SpeedChange=<bool>],[HotReset=<bool>],[DisableLink=<bool>],[Loopback=<bool>],[DisableScrambling=<bool>],[ComplianceReceive=<bool>],[IsEQ=<bool>],[Link=<integer|option>],[Lane=<integer|option>],[FTS=<integer>],[Speed=<integer>],[ReceiverPresentHint=<integer>],[TransmitterPreset=<integer>])` generates a TS1 Ordered Set.

The `TS2([AutonomousChange=<bool>],[SpeedChange=<bool>],[HotReset=<bool>],[DisableLink=<bool>],[Loopback=<bool>],[DisableScrambling=<bool>],[ComplianceReceive=<bool>],[IsEQ=<bool>],[EqualizationCommand=<bool>],[Link=<integer|`

option>],[Lane=<integer|option>],[FTS=<integer>],[Speed=<integer>],[ReceiverPresentHint=<integer>],[TransmitterPreset=<integer>]) generates a TS2 Ordered Set.

Note that all parameters for these macros of type *boolean* can be given as a flag.

The **IsEQ** parameter can be set to true to indicate that this Ordered Set is an EQ Ordered Set.

The parameter **Speed** affects the data rate identifier of the Training Sequence. The default value for this parameter is **0b001** for a Gen1 block and **0b011** for a Gen2 block.

The parameters **AutonomousChange**, **SpeedChange**, **HotReset**, **DisableLink**, **Loopback**, **DisableScrambling**, **ComplianceReceive** and **EqualizationCommand** enable the corresponding bits in the Training Sequence when set. All these flags are off by default. The argument **EqualizationCommand** is only valid for EQ Ordered Sets.

Note that the **DisableScrambling** bit does not actually disable scrambling, it just sets the corresponding bit. To disable scrambling, the **ScramblerOff** macro can be used.

The parameters **Link** and **Lane** set the encoded link and lane numbers. The lane number for consecutive lanes is automatically incremented per lane. Both **Link** and **Lane** can be set to **PAD**, which is the default value for both parameters.

The parameters **ReceiverPresentHint** and **TransmitterPreset** affect the corresponding bit fields in EQ Ordered Sets. These parameters are zero by default.

The parameter **FTS** affects the N_FTS field in the Training Sequence. This parameter is 255 by default.

Example:

```
Datarates: 2.5Gbps;

Blocks:

block1: TS1(Link=0, Lane=0, AutonomousChange, Loopback,
Speed=0b111, IsEQ, TransmitterPreset=4);

Sequence:

1. block1;

LoopTo 1;
```

DS, DB, DU

There are three macros to generate data blocks:

`DS(Payload=<rawdata|symbol|filename>),`
`DB(Payload=<rawdata|symbol|filename>)` and
`DU(Payload=<rawdata|symbol|filename>)`

All three of them accept one or more pattern data elements of the types *rawdata*, *symbol* or *filename*. Note that the `s`-suffix for *rawdata* or *symbols* is not allowed, and that *rawdata* must be given in byte- granularity.

The `DS` macro (“Data Scrambled”) scrambles the payload. The `DB` (“Data Bypassed”) macro bypasses the scrambler, that is, the payload is not scrambled, but the scrambler is advanced for every byte. The `DU` macro (“Data Unscrambled”) neither scrambles the data nor advances the scrambler.

All three macros pad the payload with zeros until the total payload length in symbols is a multiple of the lane count. Thus, the data block will be placed properly on all lanes.

Example:

```
Datarates: 2.5Gbps;

Blocks:

block1: DS('COM, 0x00n15');

Sequence:

1. block1;

LoopTo 1;
```

In this example, block1 will generate a K-character, followed by fifteen D-characters representing scrambled zeros.

The following sections describe all macros that are available in Gen3.

PCI Express Gen3/Gen4 Macros

EIOS, EIEOS

The EIOS() generates an Electrical Idle Ordered Set.

The EIEOS([Lane=<integer>], UseGen3Eieos=false) macro generates an Electrical Idle Exit Ordered Set. The parameter Lane defines the lane number, which affects the scrambler reset value. The lane number is automatically incremented for consecutive lanes. The default value for this parameter is zero. If datarate is Gen4 (16GT/s) the EIEOS has an additional property UseGen3Eieos. By default, it is set to false which means Gen4 EIEOSs are used when datarate is 16GT/s. That is the implementation against PCIe4 Base rev. 0.7 and above. When it is set to true, Gen3 EIEOSs are used at gen4 speed. This is defined in PCIe4 Basespec rev. 0.5.

SKP, FTS, SDS

SKP([Compliance=<bool>],[n=<integer>],[Error=<integer>],[ShowParity=<bool>],[Parity=<integer>]) macro generates a Skip Ordered Set. The parameter n determines the number of SKP symbols. The default value for this parameter is 12.

The argument for Compliance. If Compliance is true, the last two bytes of the SKP Ordered Set encode the current error state, which is given by the parameter Error (default value zero).

If ShowParity (cannot be given as a flag) is true, the data parity is encoded in the Ordered Set; if it is false, the current LFSR state is encoded. If this parameter is omitted, the default value is true when the last block was a Data Block. The data parity can be overridden with the Parity parameter. If this parameter is omitted, the tracked disparity is used.

The FTS() macro generates a Fast Training Sequence.

The SDS() macro generates a Start of Data Stream Ordered Set.

TS1, TS2

The TS1([AutonomousChange=<bool>],[SpeedChange=<bool>],[HotReset=<bool>],[DisableLink=<bool>],[Loopback=<bool>],[ComplianceReceive=<bool>],[UsePreset=<bool>],[ResetEieosIntervalCount=<bool>],[RejectCoefficientValues=<bool>],[Link=<integer|option>],[Lane=<integer|option>],

[FTS=<integer>],[Speed=<integer>],[TransmitterPreset=<integer>],[EqualizationControl=<integer>],[FS=<integer>],[LF=<integer>],[PreCursor=<integer>],[Cursor=<integer>],[PostCursor=<integer>],[DcImbalance=<integer>]) generates a TS1 Ordered Set.

The TS2([AutonomousChange=<bool>],SpeedChange=<bool>],[HotReset=<bool>],[DisableLink=<bool>], [Loopback=<bool>],[ComplianceReceive=<bool>],[ResetEqualization=<bool>],[QuiesceGuarantee=<bool>],[Link=<integer|option>],[Lane=<integer|option>],[FTS=<integer>],[Speed=<integer>],[DcImbalance=<integer>]) generates a TS2 Ordered Set.

Note that all parameters for these macros of type boolean can be given as a flag.

The parameter Speed affects the data rate identifier of the Training Sequence. The default value for this parameter is 0b111.

The parameters AutonomousChange, SpeedChange, HotReset, DisableLink, Loopback, ComplianceReceive, UsePreset, ResetEieosIntervalCount, RejectCoefficientValues, ResetEqualization and QuiesceGuarantee enable the corresponding bits in the Training Sequence when set. All these flags are off by default.

The parameters Link and Lane set the encoded link and lane numbers. The lane number for consecutive lanes is automatically incremented per lane. Both Link and Lane can be set to PAD, which is the default value for both parameters.

The parameters TrasnsmitterPreset, EqualizationControl, FS, LF, PreCursor, Cursor and PostCursor affect the corresponding bit fields. These parameters are zero by default.

The parameter FTS affects the N_FTS field in the Training Sequence. This parameter is 255 by default.

The parameter DcImbalance can be used to distort the DC balance. A value of zero (default value) means that the DC balance compensation, which is done in the last two bytes of the Ordered Set, is done as usual. A non-zero value for DcImbalance means that this value is added to the tracked DC balance before the DC balance compensation is conducted.

DS, DB, DU, OSS, OSB, OSU

There are three macros to generate Data Blocks:

`DS(Payload=<rawdata|symbol|filename>),`
`DB(Payload=<rawdata|symbol|filename>)` and
`DU(Payload=<rawdata|symbol|filename>)`.

All three of them accept one or more pattern data elements of the types *rawdata*, *symbol* or *filename*. Note that a *symbol* in this context can only be one of the Framing Tokens `IDL`, `SDP`, `STP`, `EDB` or `EDS`. Framing Tokens must not have a disparity sign.

The `DS` macro ("Data Scrambled") scrambles the payload. The `DB` ("Data Bypassed") macro bypasses the scrambler, that is, the payload is not scrambled, but the scrambler is advanced for every byte. The `DU` macro ("Data Unscrambled") neither scrambles the data nor advances the scrambler.

All three macros pad the payload with zeros until the total payload length in 130b symbols is a multiple of the lane count. Thus, the Ordered Set will be placed properly on all lanes.

Example:

```
Datarates: 8Gbps;

Blocks:

block1: DS(EDB);

Sequence:

1. block1;

LoopTo 1;
```

In this example, block1 will generate an EDB ("end bad") Framing Token, followed by zeros.

The macros `OSS(Payload=<rawdata|symbol|filename>),`
`OSB(Payload=<rawdata|symbol|filename>)` and
`OSU(Payload=<rawdata|symbol|filename>)`

generate Ordered Set Blocks instead of Data Blocks. They accept the same arguments and process them the same way, but generate a different Sync Header.

The `OSS (“Ordered Set Scrambled”)` macro scrambles the payload. The `OSB (“Ordered Set Bypassed”)` macro bypasses the scrambler, that is, the payload is not scrambled, but the scrambler is advanced for every byte. The `OSU` macro (“Ordered Set Unscrambled”) neither scrambles the data nor advances the scrambler.

ILT

The `ILT(Direction, Source)` macro generates the blocks needed to perform the interactive link training using the internal M8041A feature.

The arguments of this macro are the `Direction` (Up or Down) and the block branch `Source` (DetectState or TargetState). The intended use of this block is as shown below.

Example:

```
Datarates: 8Gbps;

Blocks:

link_down: ILT(Down, DetectState);

link_up: ILT(Up, TargetState);

Sequence:

1. link_down, trigger;

2. link_up, trigger;
```

In this example, the link_down block is sent until the DUT is in the Detect State (waiting for the establishment of the link). The next block of the sequence, link_up, is sent until the DUT enters the Target State (loopback) or a link training error occurs. In case of success, the loopback pattern (which would be added to the sequence below the link_up block), is sent to the DUT. In case of link training error, the sequencer goes back to the link_down block.

CPH

The `CPH(EqPreset=<integer>,Lane=<integer>, UseGen3Eieos=<bool>)`macro generates the Compliance Pattern Header which is used as header of the 128b/130b coded compliance pattern.

The arguments of this macro allow to encode the DUT equalization preset `EqPreset` and the lane number `Lane`. It also contains an EIEOS. If speed is 16Gb/s, the EIEOS can be selected to gen3 or gen4 EIEOS with the `EUseGen3Eieos` parameter.

In order to get a full compliance pattern 32 data blocks with scrambled 0s has to be added.

Example:

```
Datarates: 8Gbps;

Blocks:

compliance_pattern:CPH(EqPreset=0,Lane=0),32{DS(0x00s16)};
```

Loopback Pattern Generation

The PCIe LTS software provides two loopback patterns:

- Compliance Pattern.
- Modified Compliance Pattern.

For Gen3, these patterns can be generated in a different way depending on the data generator used: N4903B J-BERT or M8020A J-BERT (Jitter Bit Error Ratio Tester).

In N4903B case, the loopback pattern is generated in one single block and uses a loop with a manual break. An example for the Modified Compliance Pattern is shown below:

```
Datarates: 2.5G, 5G, 8G;

Blocks:

...

loopback:EIEOS(Lane=0),256{DS(0x00s16)},255{SKP(),256{DS(0
x00s16)}} @3;

Sequence:

...

60. loopback, manual;
```

For M8020A, the loopback pattern is divided into several blocks and the sequence uses sub loops and loop repetitions to build the entire pattern:

```
Datarates: 2.5G, 5G, 8G;

Blocks:

...

loopback_eieos: EIEOS(Lane=0), DS(0x00s16) @3;
loopback_prbs: DS(0x00s16) @3;
```

```
loopback_skip:SKP(),DS(0x00s16),DS(0x00s16) @3;
Generatorsequence:

...

61. loopback_eieos;

62. loopback_prbs, 255;

63. loopback_skip;

64. loopback_prbs, 254;

LoopTo 63, 255;

LoopTo 61;
```

The main reason is that they have different pattern granularity restriction. The N4903B always has a granularity of 512-bits. The M8020A uses a granularity of 80-bits for Gen1 and Gen2 and a granularity of 130-bits for Gen3.

In the examples shown above, each macro: EIEOS(), DS() and SKP() generates 130-bits (since Gen3 uses 128b/130b encoding), that is exactly the bit granularity of the M8020A. That makes it possible to create a block with a single macro without repetitions. The necessary repetitions of the macros are done by using the loop count, which doesn't consume pattern memory.

The other advantage of the M8020A here is the use of sub-loops which are not possible for the N4903B pattern generator.

4 External Pattern Files

Pattern data can be placed in pattern files. The data type *filename* refers to such files. Pattern files can contain *rawdata*, symbols or the keywords `s` and `u`. All elements must be separated either by white-space or a single comma.

The keyword `s` is expanded to the macro `ConvertTo8b10b()`. The keyword `u` is expanded to the macro `Disable8b10b()`. Since these keywords are expanded to macros, using pattern files containing these keywords is only valid in a context where a macro is allowed.

Example:

```
Datarates: 1G;

Blocks:

block1: “C:\pattern.txt”;

Sequence:

1. block1;

Contents of example file “pattern.txt”:

0x1234ABCD K28.5, s 00
```

The resulting pattern will be the hexadecimal pattern data “0x1234ABCD”, then the symbol K28.5, then a D0.0 symbol (because 0x00 will be converted to D0.0 due to the `s` keyword).

5 Scripting Tips

Repetitions and loops / 59
Fulfilling Granularity Restrictions / 60

Repetitions and loops

There are two ways to define a repeated pattern:

- Using a repetition, for example `10{0x00}`
- Defining a loop count in the sequence, for example `10. pattern, 10`

The difference between these two definitions is that the first method (a repetition), generates everything inside the curly brackets multiple times, whereas the second method (a loop), instructs the pattern generator instrument to send the same data multiple times.

Using a repetition consumes more pattern memory than using a loop. However, using a repetition can also be utilized to meet the pattern granularity requirements.

Note that using a loop does not guarantee proper disparity tracking and other side effects. Consider the following script:

```
Datarates: 1G;

Blocks:

pattern1: K28.5;

pattern2: D0.0;

Sequence:

1. pattern1, 10;

2. pattern2;

LoopTo 1;
```

This script is intended to generate ten K28.5 symbols, then a D0.0 symbol. But the K28.5 symbols will violate the running disparity, since every symbol comes from the very same pattern, which has a fixed granularity. On top of that, the pattern won't meet the generator's granularity restrictions.

The following script fixes this:

Datarates: 1G;

```
Blocks:

pattern: 512{10{K28.5}, D0.0};

Sequence:

1.Pattern;
```

Note that the repetition of 512 was added to meet the granularity restrictions of the instrument. The factor of 512 was arbitrarily chosen and can be different for different instruments.

Fulfilling Granularity Restrictions

Most generator instruments have a pattern granularity restriction, and most patterns won't fulfill these restrictions.

There are two recommended methods to fulfill granularity restrictions:

- Padding the pattern.
- Repeating the pattern.

Consider the following script, which is intended to be sent to a generator instrument with a granularity of 512-bits:

```
Datarates: 1G;

Blocks:

pattern_1: 0x00n16;

pattern_2: 0xFFn16;

pause: Pause0(1m);

pattern_3: PRBN(7);

Sequence:

1.pattern_1, 1024;

2.pattern_2;

3.pause;
```

```
4.pattern_3;

LoopTo 4;
```

Downloading this script to the generator instrument will fail, because `pattern_1`, `pattern_2` and `pattern_3` don't have 512-bits granularity. Note that the block `pause` will have a granularity of 512-bit, since the `Pause0` macro was used exclusively on a block, so it will automatically be aligned.

`pattern_1` is repeated 1024 times in the sequence. By reducing the loop count from 1024 to 256, and repeating the pattern itself 4 times, the pattern length becomes 512 bit.

`pattern_2` cannot be repeated, as this would alter the pattern (however, depending on the context where the pattern is used, repeating could be possible anyway). But, the block is followed by an arbitrary number of zeros, so padding the block with zeros at the end would not alter the actual pattern.

`pattern_3` does not have 512 bit granularity either. The PRBN is of seventh order, so the length will be 128 bits. However, the only solution is to repeat the PRBN four times to fit it into the pattern memory.

The fixed script could look like this:

```
Datarates: 1G;

Blocks:

pattern_1: 4{0x00n16};   // repeated 4x

pattern_2: 0xFFn16, Pad0(); // padded at the end

pause: Pause0(1m);

pattern_3: 4{PRBN(7)};   // repeated 4x

Sequence:

1.pattern_1, 256;// decrease loop count by 4x

2.pattern_2;

3.pause;

4.pattern_3;

LoopTo 4;
```

This information is subject to
change without notice.
© Keysight Technologies 2017
Edition 4.0, July 2017

www.keysight.com