Keysight N5990A
USB 3.0/3.1
Link Training Suite

Language Guide

KEYSIGHT
TECHNOLOGIES

Notices

Manual Part Number

N5990-91120

Edition

Edition 1.0, May 2015

Keysight Technologies, Deutschland GmbH

Herrenberger Str. 130

71034 Böblingen, Germany

For Assistance and Support

http://www.keysight.com/find/assist

Limitation of Warranty

Warranty

Technology Licenses

Restricted Rights Legend

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.

NOTE

A **NOTE** provides important or special information.

Contents

1   Introduction

1.1   About This Document

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

1.1.1   Definitions

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 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 a minimum pattern length and a 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.

KEYSIGHT
TECHNOLOGIES

1.1.2 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: *datatype*

1.2   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.

1.2.1 General Syntax

The keywords `Datarates:`, `Blocks:` and `Sequence:` defines 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 # 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 */.

1.3   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.

1.3.1   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 1-1 illustrates how two patterns of different data rates can be generated by doubling every bit of the slower pattern.
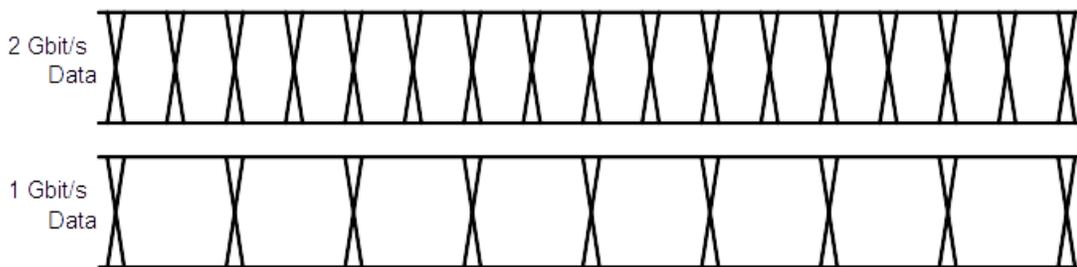
Figure 1-1: Data rate encoding (factor 2:1)

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 1-2). Note that this scheme introduces a small amount of jitter.
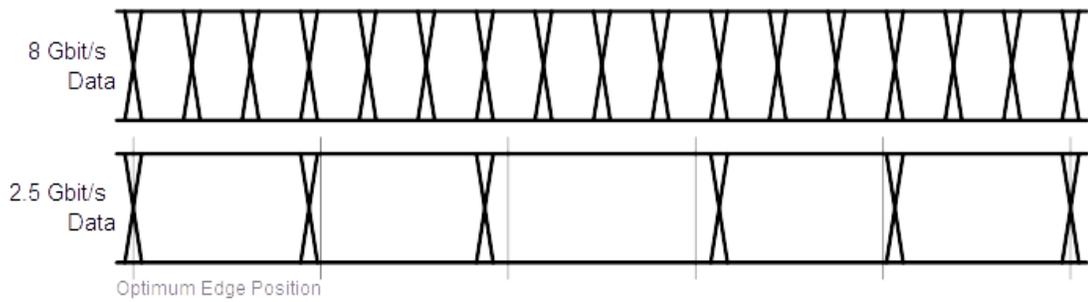
Figure 1-2: Data rate encoding (factor 16:5)

1.3.2 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 1-3 shows the impact of the inversion and the duty cycle (zero ratio, i.e. the duty cycle of a logical zero).
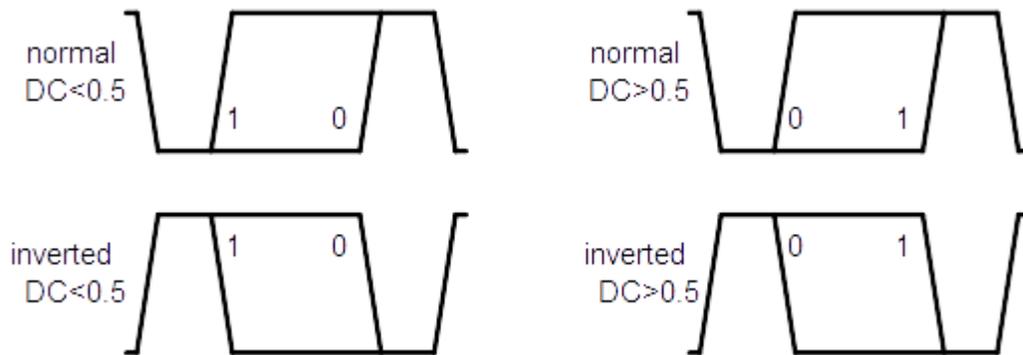
Figure 1-3: PWM Parameters

Figure 1-4 shows an example PWM waveform, compared to data rate encoded data.
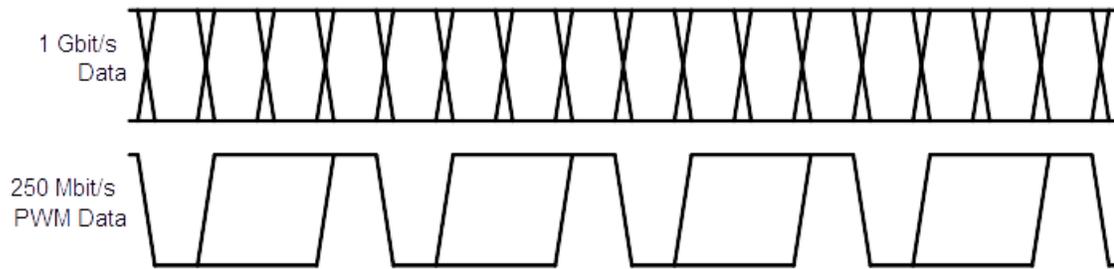
Figure 1-4: Example PWM waveform

1.4   Data Types

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

1.4.1   Names

A name can define a block, a macro, or different kinds of arguments, which are all explained later. 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`

1.4.2 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), `+0.1 k` (= 100)
For clarity, 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, 1 Gbps`

1.4.3 Pattern Data

Pattern data is one of the most important parts of this language. A pattern can be either 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, i.e. 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-granulariy 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.
Example: `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 suffix s (lower-case) and a number. Details of these mechanisms will be provided later in this document.
Example: `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, i.e. 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.

Example: `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.

Example: `"C:\demo.pat"`

Details about pattern files are handled later in this document.

1.4.4   Other Data Types

There are some other data types that are used by macros.

The data type bool 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. It's 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. It's meaning depends on the context. The data is allowed to not contain single-quotes.

Examples: `'any kind of data'`, `'123'`, `'$"/'`

1.5   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 is used.

Example:

```
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.

1.6   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 an at-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
```

1.6.1   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 the referenced block is defined.

Example:

```
my_pattern: 0xAA, 0xBB
block_1: my_pattern, 0xCC;        // equals 0xAABBCC
block_2: 0x00, my_pattern, 0x11; // equals 0x00AABB11
```

1.6.2   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 documentation of the `Sequence:` script section for details about looping.

The 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
```

1.6.3 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 a 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 of 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.

1.6.4 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 (e.g. `42` as an argument for an integer parameter), or by typing the parameter name, followed by an equation sign and the argument value.

For 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 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 lower-case.

Many macros have optional parameters, that is, specific arguments may be omitted. In that case, a specified default value will be assumed. Optional parameters are documented in square brackets. For example, the macro `Pad([PaddingPattern=<rawdata>])` can be invoked with an argument for `PaddingPattern`, 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 `bool` 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. For 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.

1.6.5   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`.

1.6.6   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-charactes. 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.

1.7   Sequence Definitions

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 the user breaks 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 the user triggers manually
- Then, the block_3 pattern is transmitted once
- Finally, the block_2 pattern is transmitted three times
- Since the pattern generator loops starting from step 2, block_3 and block_2 (three times) are repeated infinitely

1.8   A Complete Example

The following script shows a 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 will start 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:
0xC2D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B555555555555555555555555555555555555555555555555555555555500000000000000000000000
00000000000000000000000000000000000000000555555555555555555555555555555555555555
55555555555555555555550000000000000000000000000000000000000000000000000000000000000000
55555555555555555555555555555555555555555555555555555555555555500000000000000000000000
00000000000000000000000000000000000000000555555555555555555555555555555555555555555555
55555555555555555550000000000000000000000000000000000000000000000000000000000000000000055
55555555555555555555555555555555555555555555555555555555555555500000000000000000000000
00000000000000000000000000000000000000000555555555555555555555555555555555555555555555
55555555555555550000000000000000000000000000000000000000000000000000000000000000000005555
55555555555555555555555555555555555555555555555555555555555000000000000000000000000000
00000000000000000000000000000000000000000555555555555555555555555555555555555555555555555
55555555555555000000000000000000000000000000000000000000000000000000000000000000000555555
55555555555555555555555555555555555555555555555555555555555000000000000000000000000000000
00000000000000000000000000000000000000000555555555555555555555555555555555555555555555555
55555555555500000000000000000000000000000000000000000000000000000000000000000000000

2 Common Macros

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

2.1 Filling, Padding and Synchronizing

2.1.1 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.
For 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)`.

KEYSIGHT
TECHNOLOGIES

2.1.2 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 can be 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 were 512 bits, and a pattern of 12 bits were already defined, the `Pad` macro would insert 500 additional bits.

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

Padding will only occur 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 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)`.

2.1.3 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.

2.2 Pattern Distribution

2.2.1 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.

2.3   8b/10b Encoding

2.3.1   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.

2.3.2   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.

2.3.3   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.

2.4   Data Rate and PWM Encoding

2.4.1   Rate, CustomRate

Every block has a data rate assigned, either with the at-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.

2.4.2   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 zero to one exclusively.

2.5   PRBS Generation

2.5.1   PRBS, PRBN

The macro `PRBS([Invert=<bool>], [Reverse=<bool>], [Order=<integer>], [Length=<integer>], [Polynomial=<integer>], [Distribute=<bool>])` generates a 2n-1 PRBS pattern using a LFSR implementation.

The parameter `Order` specifies the PRBS order and can be in the range 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 xn is omitted; the term x0 is defined with the most significant bit, the term xn-1 is defined with the least significant bit. For example, the polynomial x7+x6+1 can be given as `Polynomial=0b1000001` (the left-most "1" represents x0=1, the right-most "1" represents x6). 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 2n PRBS instead of a 2n-1 PRBS. This can be generate 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 2n PRBS.

To generate a 2n 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.

2.5.2   HardwarePRBS

Most generator instruments can generate PRBS in hardware, using a built-in LFSR. The advantage of a hardware PRBS is that is doesn't 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 that this block cannot be referenced in other blocks. This is because a generator instrument cannot mix memory patterns and LFSR patterns.

2.6   Error Insertion

2.6.1   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 (i.e. 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.`

2.6.2   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 current running disparity in the block (i.e. +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 USB3 Patterns

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

3.1   USB3 Symbols

Additionally to the classic K-characters and D-characters, the following symbol names are recognized: `SKP`, `SDP`, `EDB`, `SUB`, `COM`, `SHP`, `EDN`, `SLC`, `EPF`. These symbol names are valid everywhere where K-characters are valid. They can also be combined with an `n`- or `s`-suffix.

3.2   USB Macros Overview

There are several USB specific macros that allow to generate training sequences and other specific patterns with the call of a single macro. All USB macros are handled before the common macros are handled.

Most macros differ between the USB3 speed classes. SuperSpeed class uses 8b/10b encoding, whereas SuperSpeedPlus uses 128b/130b encoding. The data rate must be define as 5Gbps for SuperSpeed and 10Gbps for SuperSpeedPlus. For example, for SuperSpeed the first line of the script must be:

DataRates: 5G;

KEYSIGHT
TECHNOLOGIES

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

Note that during macro processing, the macros `DispReset()` and `SetDistri()` are automatically placed at the beginning of the first block where UBS 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).

3.3   USB3 Common Macros

3.3.1   USBPadIdle, USBPauseIdle, USBIdle

The Electrical Idle is necessary to enter in the Rx.Detect LTSSM state and also to generate the LFPS signal.

Depending on the situation it can be used three different macros:

`USBPadIdle(double tPeriod = 50e-9, int dutyClycle = 50)` inserts as many bits of electrical Idle necessary to meet the granularity.

`USBPauseIdle(double t, double tPeriod = 50e-9, int dutyClycle =  50)` generates an Idle signal during t seconds.

For M8020A, due to timing issues, it is necessary to create a periodic signal with the same period as the LFPS Burst. Use `tPeriod` and `dutyCyle` input parameters to specify the period and duty cycle for M8020A case. The values should be the same as the ones used in the LFPS macros.

3.4   USB SuperSpeed Macros

3.4.1   USBIdle

USBIdle(int numIdleBits, double tPeriod = 50e-9, int dutyClycle = 50) generates an Idle signal with a length of numIdleBits.

3.4.2   SKP

The `SKP()` macro generates a Skip Ordered Set. It is used for clock compensation. The SKP Ordered Set shall consist of a SKP K-Symbol followed by a SKP K-Symbol.

3.4.3 LFPS, LFPSBURST

`LFPS(double tPeriod = 50e-9, double tBurst = 1e-6, double tRepeat = 10e-6, int dutyCycle = 50)` macro generates a Low frequency period signaling (see Figure 5). It is used among other things for the link training.
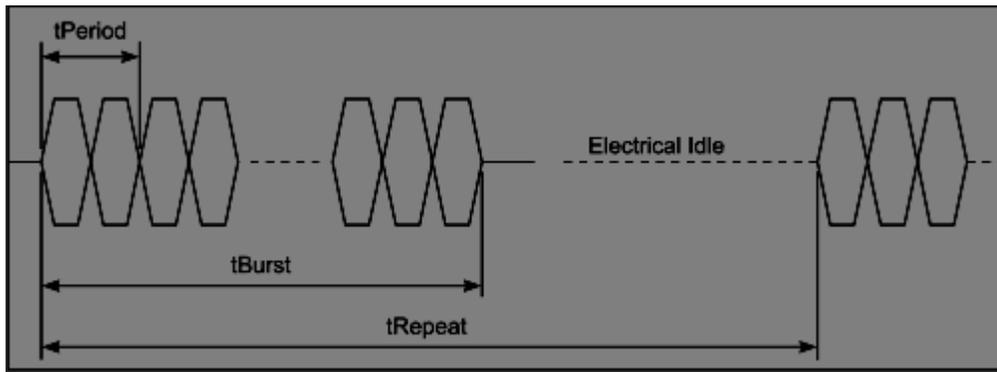

Figure 5: LFPS Signaling

`tPeriod` is the period of an LFPS cycle.

An LFPS burst is the transmission of continuous LFPS signal over a period of time defined by `tBurst`.

An LFPS sequence is defined by the transmission of a single LFPS burst of duration tBurst over a period of time defined by `tRepeat`. The link is in electrical idle between the two contiguous LFPS bursts.

The LFPS cycle can be defined with a `dutyCycle` (between 0 and 100).

`LFPSBurst(double tPeriod = 50e-9, double tBurst = 1e-6, int dutyCycle = 50)` macro generates a LFPS burst.

An LFPS burst is the transmission of continuous LFPS signal over a period of time defined by `tBurst`.

 `tPeriod` is the period of an LFPS cycle.

The LFPS cycle can be defined with a `dutyCycle` (between 0 and 100).

3.4.4 TSEQ

The `TSEQ()` macro generates the TSEQ training sequence. This Ordered Set should be repeated 65,536 times to allow for testing many coefficient settings.

3.4.5 TS1, TS2

The `TS1(bool reset = false, bool loopback = false, bool disableScrambling = false)` generates a TS1 Ordered Set.

The `TS2(bool reset = false, bool loopback = true, bool disableScrambling = false)` generates a TS2 Ordered Set.

The `reset` parameter is set by the Host only in order to reset the device.

When `loopback` is true, the receiving component enters digital loopback.

When `disableScrambling` is true, the receiving component disables scrambling.

During training the transmitter should insert SKP Ordered Sets with an average of one SKP for every 354 symbols. A TS1 or TS2 has 16 symbols, therefore a SKP Ordered Set shall be sent at least every 22 training sequences. Additionally, it is necessary to consider the granularity of the generator. The next example shows the block that the USB Link Training Suite will generate automatically for sending 1250 TS1

```
Blocks:
block1: 104 {12{TS1()},1{ SKP() }}, 2{TS1()}, 16{K28.1};
```

It is inserted one SKP Ordered Set every 12 TS1 training sequences 104 times. That makes 1248 TS1. Then two TS1 are sent to reach the 1250 and, at the end, 16 K28.1 (SKP symbols) to meet the granularity. These number are calculated to minimize the number of necessary pad symbols.

3.4.6 BRST

`BRST(string fillSymbol 0 "K28.1")` macro generates the BERT Ordered Set BRST. Any time a BRST is received, the error count register EC is set to 0 and the scrambling LFSR is set to 0FFFFh.

3.4.7 CP0, CPOM8000

`CP0(int numSKP = 2, int numSKPCycles = 188, int D0RotateNumber = 0)` macro for J-BERT and `M80000P0(int numSKP = 2, int numSKPCycles = 188, int D0RotateNumber = 0)` for M8020A can be used as test pattern. It is a pseudo-random data pattern generated by the scrambled D0.0 compliance sequence.

`numSKP` is the number of SKP symbols for each SKP Ordered Set inserted.

According the SKP rules, a SKP Ordered Set shall be inserted every 354 transmitted symbols (scrambled D0.0 in this case). numSKPCycles is the number of SKP Ordered Set sent. This value determines the length of the CP0 pattern that will be at least `numSKPCycles` x 354. Due granularity restrictions could be necessary to insert more symbols.

The transmitted CP0 pattern can be rotated for getting a faster synchronization in the Error Detector between the internal CP0 pattern and the CP0 pattern retransmitted by the DUT. `D0RotateNumber` is the number of D0.0 symbols rotated from the beginning of the sequence to the end.

3.5   USB SuperSpeedPlus Macros

3.5.1   USBIdle_GEN2

`USBIdle_GEN2(int numIdleBits, double tPeriod = 50e-9, int dutyCycle = 50)` generates an Idle signal with a length of numIdleBits.

3.5.2   LFPSBurst, LFPS_SCD1, LFPS_SCD2, LFPS_SCD_END

`LFPSBurst(double tPeriod = 50e-9, double tBurst = 1e-6, int dutyCycle = 50)` macro generates a LFPS burst.

An LFPS burst is the transmission of continuous LFPS signal over a period of time defined by `tBurst`.

tPeriod is the period of an LFPS cycle.

The LFPS cycle can be defined with a `dutyCycle` (between 0 and 100).

`LFPS_SCD1(double tPeriod = 50e-9, double tBurst = 1e-6, int dutyCycle = 50, double tRepeat0 = 7.5e-6, double tRepeat1 = 12.5e-6)` and `LFPS_SCD2(double tPeriod = 50e-9, double tBurst = 1e-6, int dutyCycle = 50, double tRepeat0 = 7.5e-6, double tRepeat1 = 12.5e-6)` macro generates a low frequency period signaling (see Figure 6). They are used for declaring the SuperSpeedPlus capability.
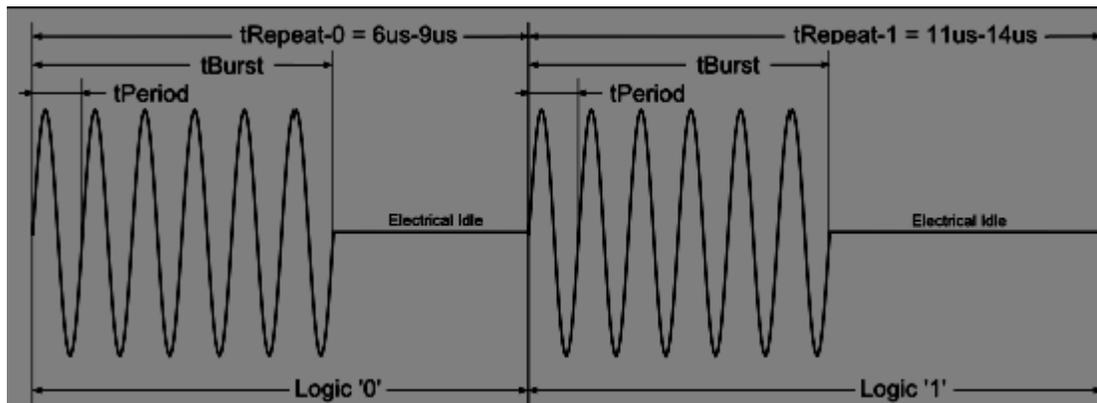

Figure 6: Signaling for SuperSpeedPlus

tPeriod is the period of an LFPS cycle.

An LFPS burst is the transmission of continuous LFPS signal over a period of time defined by tBurst.

An LFPS sequence is defined by the transmission of a single LFPS burst of duration tBurst over a period of time defined by tRepeat. The link is in electrical idle between the two contiguous LFPS bursts. In SuperSpeedPlus class the logic '1' and '0' can have different tRepeat defined by tRepeat0 and tRepeat1.

The LFPS cycle can be defined with a dutyCycle (between 0 and 100).

The LFPS_SCD1 macro is composed by 4 LPFS sequences: Logic '0', Logic '1', Logic '0' and Logic '0'.

The LFPS_SCD2 macro is composed by 4 LPFS sequences: Logic '1', Logic '0', Logic '1' and Logic '1'.

LFPS_SCD_END(double tPeriod = 50e-9, double tBurst = 1e-6, int dutyCycle = 50, double idleDuration = 40e-6) is used after a LFPS_SCD1 or a LFPS_SCD2 macro. It sends a LFPS sequence defined by tPeriod, tBurst, dutyCycle and a tRepeat of idleDuration.

3.5.3   LBPS_MSG, LBPS_END

LBPM_MSG(double tPeriod = 50e-9, int dutyCycle = 50, double tPWM = 2.2e-6, double tLBPS0 = 0.65e-6, double tLBPS1 = 1.565e-6, byte capability = 0) macro generates a SuperSpeedPlus LFPS Based PWM Message (LBPS) signal (refer to Figure 7). It is used for rate announcement and negotiation
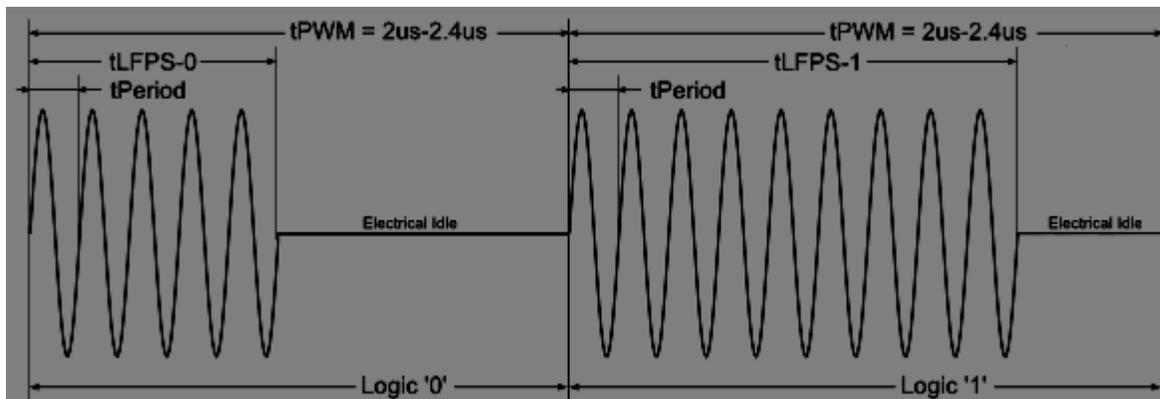

Figure 7: LBPS Signaling for SuperSpeedPlus

`tPeriod` is the period of an LFPS cycle.

The LFPS cycle can be defined with a `dutyCycle` (between 0 and 100).

For the LBPS signal, the LFPS burst duration can be different for logic '0' and logic '1'. These periods of time can be defined by `tLBPS0` and `tLBPS1`.

The duration of each LFPS sequence is defined by `tPWM`.

To send a capability LBPS the `capability` parameter must be 0x20. To send a ready LBPS the `capability` parameter must be 0x80.

`LBPM_END(double tPeriod = 50e-9, int dutyCycle = 50, double tPWM = 2.2e-6)` macro generates a LFPS signal representing the end of a LBPM. It is define by `tPeriod`, `dutyCycle` and `tPWMle` and a `tRepeat` of `idleDuration`.

3.5.4 TSEQ

The `TSEQ()` macro generates the TSEQ training sequence. This Ordered Set should be repeated 65,536 times to allow for testing many coefficient settings.

3.5.5 TS1, TS2

The `TS1(byte linkConfigValue = 0x04)` generates a TS1 Ordered Set.

The `TS2(byte linkConfigValue = 0x04)` generates a TS2 Ordered Set.

The `linkConfigValue` parameter is used to define the Link Functionality symbol within the TS1 and TS2 ordered sets. To configure the DUT into loopback set this values as 0x04.

3.5.6 SYNC

The `SYNC()` generates a SYNC Ordered Set. It is used to determine the location of the Block Headers in the received bit stream.

A `SYNC` Ordered Set shall be transmitted every 16384 TSEQ sets during training session.

A SYNC Ordered Set shall be transmitted every 32 ordered sets, when sending TS1 or TS2 ordered sets.

3.5.7 SKP

The `SKP(uint n = 12)` macro generates a Skip Ordered Set. It is used to compensate for frequency differences between the two ends of the link.

In SuperSpeedPlus, the SKP Ordered Set has variable length. The n parameter can be used to define that length. It must be a number between 4 and 36 and multiple of 4.

3.5.8   CP9

The `CP9()` macro generates a SuperSpeedPlus compliance pattern. It comprises a pseudo-random data pattern that is used to test transmitter and receiver compliance. The pattern repeat every 65536 symbols and starts with a SYNC Orderet Set.

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).

KEYSIGHT
TECHNOLOGIES

5   Scripting Tips

5.1   Repetitions and Loops

There are two ways to define a repeated pattern:

- Using a repetition, e. g. 10{0x00}
- Defining a loop count in the sequence, e. g. 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) just tells 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.

5.2   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 the 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;
```

KEYSIGHT
TECHNOLOGIES

N5990-91120

www.keysight.com