Keysight N5990A MIPI-MPHY Frame Generator

KEYSIGHT TECHNOLOGIES

Language Guide

Notices

Manual Part Number

N5990-91140

Edition

Edition 2.0, May 2016

Keysight Technologies, Deutschland GmbH

Herrenberger Str. 130

71034 Böblingen, Germany

For Assistance and Support

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

Limitation of Warranty

The foregoing warranty shall not apply to defects resulting from improper or inadequate maintenance by Buyer, Buyer-supplied software or interfacing, unauthorized modification or misuse, operation outside of the environmental specifications for the product, or improper site preparation or maintenance. No other warranty is expressed or implied. Keysight Technologies specifically disclaims the implied warranties of Merchantability and Fitness for a Particular Purpose.

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 of 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 shall control.

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.

Restricted Rights Legend

If software is for use in the performance of a U.S. Government prime contract or subcontract, Software is delivered and licensed as "Commercial computer software" as defined in DFAR 252.227-7014 (June 1995), or as a "commercial item" as defined in FAR 2.101(a) or as "Restricted computer software" as defined in FAR 52.227-19 (June 1987) or any equivalent agency regulation or contract clause. Use, duplication or disclosure of Software is subject to Keysight Technologies' standard commercial license terms, and non-DOD Departments and Agencies of the U.S. Government will receive no greater than Restricted Rights as defined in FAR 52.227-19(c)(1-2) (June 1987). U.S. Government users will receive no greater than Limited Rights as defined in FAR 52.227-14 (June 1987) or DFAR 252.227-7015 (b)(2) (November 1995), as applicable in any technical data.

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.

Technical Assistance: Contact Center

General Safety Precautions

If you need product assistance or if you have suggestions, contact the Keysight Technologies, Inc. Contact Center in your area listed on the Keysight homepage at:

http://www.keysight.com

Representatives of the Keysight Contact Centers are available during standard business hours.

Before you contact the Contact Center, please note the actions you took before you experienced the problem. Then describe those actions and the problem to the technical support engineer.

Find a Mistake?

We encourage comments about this publication. Please report any mistakes to the Contact Center. The Contact Center representative passes your comments to the Learning Products Department.

NOTE

Keysight continually strives to provide its customers with current, accurate, and complete information to assist them in the use of our products. The Keysight documents available online contain up-to-the minute changes that occurred after the preparation of this manual. They are located on the Keysight web at: http://www.keysight.com/find/support.

Contents

1 Introduction

1.1 About This Document

The MIPI M-PHY Frame Generator provides a language to edit the generated pattern with MIPI MPHY-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 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.

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.

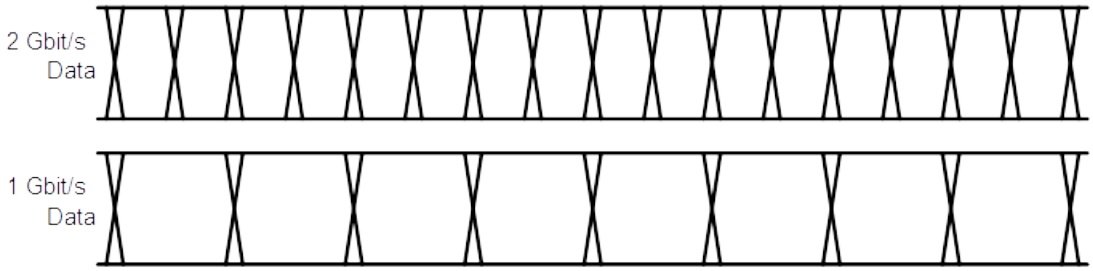

Figure 1-1: Data rate encoding (factor 2:1)

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.

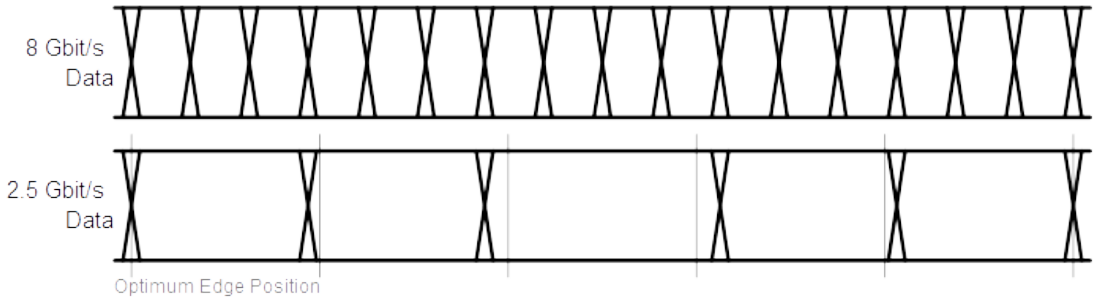

Figure 1-2: Data rate encoding (factor 16:5)

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).

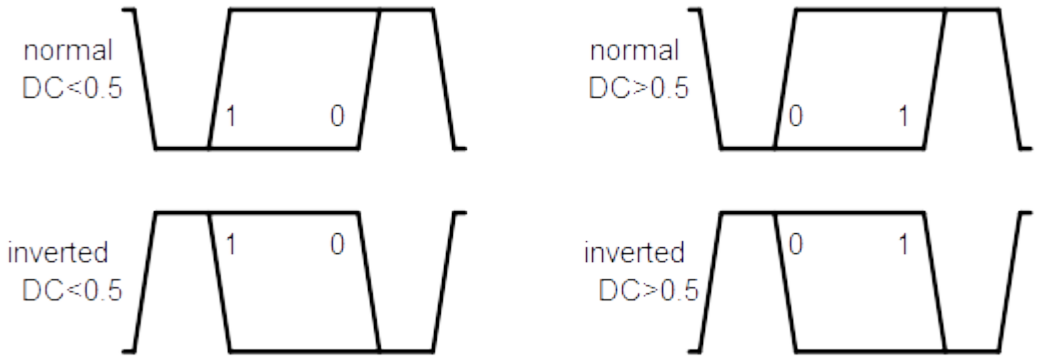

Figure 1-3: PWM Parameters

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

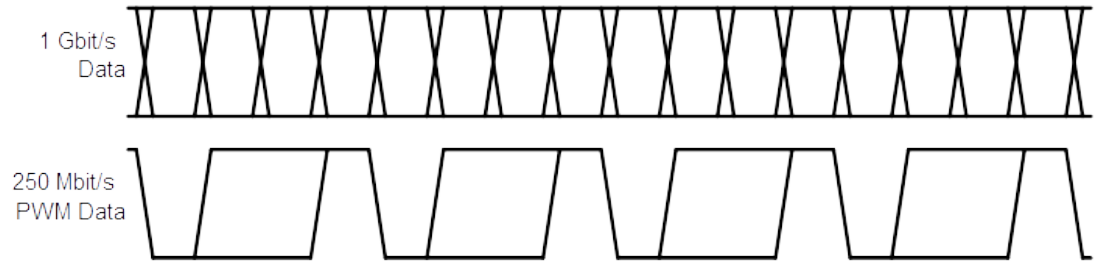

Figure 1-4: Example PWM waveform

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-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.

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

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. Its 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*, and *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 Using Data Rates

Section “Data Rate Definitions” on page 15 explained how data rates can be defined. The definition basically states which data rates are available. Now each block can use any of those data rates.

By default, each block uses the highest available data rate (i.e. the highest data rate that was defined in the Datarates: section). With the @-symbol at the end of a block definition the data rate for the entire block can be changed

Example:

```
_ Datarates: 1G, 2G;
Blocks:
   a) 0b01n10 @1;
   b) 0b01n10;
```

In this example, block “a” uses the data rate 1 Gbit/s, whereas block b uses the data rate 2 Gbit/s.

As section “Data Rate Encoding” on page 10 explains, the different data rates are achieved by repeating bits. Therefore, the above example would be identical to the following one:

```
Datarates: 2G;
Blocks:
   c) 0b0011n10;
   d) 0b01n10;
```

The “01” pattern of block “a” is simply replaced with a “0011” pattern, which effectively reduces the data rate from 2 Gbit/s to 1 Gbit/s.

Please note that the generator instrument can run at any data rate, which is independent of the data rates that are defined in the script! Therefore, if the generator instrument is set to e.g. 4 Gbit/s, the Pattern will actually run at 4 Gbit/s, even though the script says 2 Gbit/s. Put in other words, the data rates that are defined in the script are only used to derive the ratio between the ideal data rate and some slower data rates

1.6.2 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
```

1.6.3 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
```

1.6.4 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.

1.6.5 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 in 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

1.6.6 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`.

1.6.7 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.

1.7 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 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:

```
0xC2D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D8B62D
8B62D8B62D8B62D8B62D8B55555555555555555555555555555555555555
555555555555555555555555550000000000000000000000000000000000
000000000000000000000000000000555555555555555555555555555555
```

```
55555555555555555555555555555555000000000000000000000000000
00000000000000000000000000000000000000000555555555555555555
55555555555555555555555555555555555555555000000000000000000
00000000000000000000000000000000000000000000000055555555555
55555555555555555555555555555555555555555555555550000000000
00000000000000000000000000000000000000000000000000000555555
55555555555555555555555555555555555555555555555555555555500
00000000000000000000000000000000000000000000000000000000000
00555555555555555555555555555555555555555555555555555555555
55555000000000000000000000000000000000000000000000000000000
00000000005555555555555555555555555555555555555555555555555
55555555555550000000000000000000000000000000000000000000000
00000000000000000055555555555555555555555555555555555555555
55555555555555555550000000000000000000000000000000000000000
00000000000000000000000005555555555555555555555555555555555
55555555555555555555555555555555555555555000000000000000000
00000000000000000000000000000000000000555555555555555555555
55555555555555555555555555555555555555500000000000000000000
00000000000000000000000000000000000000000
```

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)`.

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.

2.1.4 SyncData, SyncData0, SyncData1

These macros basically do the same as the corresponding Sync() macros. However, there are two major differences:

- The Sync() macros take care that after data rate and PWM encoding, all lanes are in sync. The SyncData() macros do not take data rate and PWM encoding into account, i.e. if the channels are encoded with different schemes (e.g. NRZ on one channel and PWM on another), the resulting data streams might not be in sync any more when using the SyncData() macros
- The Sync() macros insert as many bits as required, and does not support symbols. The SyncData() macros allow to insert symbols, and they make sure the specified pattern is inserted as a whole.

Therefore, the SyncData() macros are suitable if multiple channels use data of the same kind, and must be synchronized with a well-defined pattern or symbol. The Sync() macros, however, are also suitable if different channels use different data rate or PWM encoding schemes, but they don't guarantee that the specified pattern is inserted as a whole.

Example 1: `[0: 0b0n5; 1: 0b0n15], SyncData1(), Pad0();`

In this example, ten '1' bits will be inserted in the first channel.

Example 2`: [0: 0b0n5; 1: 0b0n15], SyncData(K28.5), Pad0();`

In this example, a K28.5 symbol will be inserted in the first channel. Note that this example would fail if the difference between the channels would not be a multiple of ten (because symbols are always ten bits).

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.

2.4.3 Remarks About Data Rates

All data rate and PWM macros accept the data rates that are provided “as is”; no cross-checks with the actually configured data rate of the generator instrument are done. When manually editing a script, the user must take care that the highest data rate that is defined in the `Datarates:` section of the script is actually the data rate that the generator instrument runs with.

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 `Polynomia`l 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 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.

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 MIPI M-PHY Patterns

To simplify the definition of MIPI M-PHY patterns, there are symbols and macros especially designed for MIPI M-PHY.

3.1 MIPI M-PHY Symbols

The following symbol names are recognized: `FILLER`, `MK0`, `MK1`, `MK2`. These symbol names are used for control functionality. They can also be combined with an `n`- or `s`-suffix.

3.2 MIPI M-PHY Macros Overview

There are several MIPI M-PHY specific macros that allow to generate the patterns that the DUT needs to enter or exit from a link state with the call of a single macro. All MIPI M-PHY macros are handled before the common macros are handled.

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

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

The DataRates section must not be included before the Block section. It is generated internally according to the Gear selected in the “Timing Setup” windows of the user interface.

3.3 MIPI M-PHY Protocol Independent Macros

The following macros are available with the base option, 366.

3.3.1 DIFP

The DIFP(uint_8 Period) macro generates 1´s for given Period in term of bits.

Example: DIFP(2) then output = 0xFF FF

3.3.2 DIFN

DIFN(uint_8 Period) generates 0´s for given `Period` in term of bits.

Example: DIFN(2) then output = 0x00 00

3.3.3 Activate

Activate() macro is used for exiting from HIBERN state. It drives DIFN for T_Activate. T_Activate duration can be configured with the timing parameter “Prepare Length” (refer to the section “Pattern Settings” of the “Keysight N5990A MIPI M-PHY Frame Generator Software User Guide”).

Example: Activate() then output = 0x00 00 00 if Prepare Length is 3.

3.3.4 HSStart

HSStart() macro is used to start the link in HS mode. It sends DIFP states, given by the Prepare Length, followed by sync symbols, given by the SYN Length (refer to the section “Pattern Settings” of the “Keysight N5990A MIPI M-PHY Frame Generator Software User Guide”).

Example: HSStart() then output = 0x FF FF, D10.5 (if Prepare Length is 2).

3.3.5 LSStart

LSStart() macro is used to start the link in LS mode. It sends DIFP states, given by the LS Prepare Length (refer to the section “Pattern Settings” of the “Keysight N5990A MIPI M-PHY Frame Generator Software User Guide”).

Example: LSStart() then output = 0x FF FF (if LS Prepare Length is 2).

3.3.6 HSExit

HSExit() (or Shall() for compatibility for old Frame Generator version) allows to exit from the HS mode and return to STALL state (STALL is the power saving state for HS mode). Stall Length can be configure in the Timing section with the parameter “Stall Length” (refer to the section “Pattern Settings” of the “Keysight N5990A MIPI M-PHY Frame Generator Software User Guide”).

Example: HSExit() then output = 0x 00 00 00

3.3.7 PAD

PAD() macro will add some DIFN states until the block meets the granularity. It has the same function that the common macro Pad0().

Example: Pad() then output = b0000000...

NOTE

All data blocks need to meet the requirements of the ParBERT/J-BERT pattern granularity and if the reference clock is needed the pattern length needs to be also a multiple of the ratio between the high speed data rate and the reference clock frequency!

Example: J-BERT has a pattern granularity of 512 bits. For Gear 1A (1248 Mb/s) and a reference clock frequency of 26 MHz a factor of 48 needs also to be applied, which leads to a pattern length of 512*3=1536 bits or a multiple of that.

Via the user interface the “Auto Length” feature (refer to Figure 6-8 ”Data Dialog Windows” of the “Keysight N5990A MIPI M-PHY Frame Generator Software User Guide”) can be enabled. In this case the pattern will be padded/rolled out until the pattern granularity is used. The “PAD” will be applied first, so that a block which contains a “PAD” entry will already be extended to meet the pattern granularity feature and the auto padding will not effect this block.

3.3.8 LB

LB(fileName) generates the LS data given in the file fileName. The file should be in the same folder as the sequence file. The type of LS transmission (SYS burst or PWM) will be set via the user interface. The filename needs to end with ".dat" or ".txt"; if not, the following string will be interpreted as data of the same format as a pattern file. If the data contains valid 10 bit words, a recoding is done to keep the disparity.

3.3.9 B

B(fileName) macro generates the HS data given in the file fileName. The file should be in the same folder as the sequence file. The filename needs to end with ".dat" or ".txt", if not, the following string will be interpreted as data of the same format as a pattern file. If the data contains valid 10 bit words then a recoding is done to keep the disparity.

3.3.10 LUI

The data configured in the Data Dialog (refer to Figure 6-8 of the "Keysight N5990A MIPI M-PHY Frame Generator Software User Guide") will be set at the location of LUI() macro and sent out as low speed data (either PWM or SYS burst, as selected by the user interface).

3.3.11 UI

The data configured in the Data Dialog (refer to Figure 6-8 of the "Keysight N5990A MIPI M-PHY Frame Generator Software User Guide") will be set at the location of `UI()` macro and sent out as HS data.

3.3.12 LSExit

The macro LSExits() (LSStall() or Sleep() for compatibility with old Frame Generator()) is used to enter in the power saving state in LS-mode. It will add as many DIFN (0) states as given by the Sleep length in the user interface. Refer to the section "Pattern Settings" of the "Keysight N5990A MIPI M-PHY Frame Generator Software User Guide" for more details.

3.3.13 Guard Band

GuardBand() macro is used to set the MK2 Extended Guard Band. It should be used alone in one block and placed in between HS/LSStart and the payload (refer to the section “Pattern Settings” of the “Keysight N5990A MIPI M-PHY Frame Generator Software User Guide”).

3.3.14 PWMExit

PWMExit() macro is used to exit from PWM transmission state and return to Save state. Will generate 9+10N PWM b0 states followed by a b1 state.

Example. PWNExit() then output = b00000000000000000001

3.3.15 SYSExit

SYSExit() macro is used to exit form SYS-BURST and return to Save state. It will generate DINF for a time greater or equal to 10UI.

Example: SYSExit() then output = DIFN10

3.3.16 Sequence Example

Blocks:
Init1: LSStart, LB"Esc0ms.txt", DIFN10;
HS: HSStart, B"CJPat.txt", DIFN20;
Sequence:
1. Init1,1;
2. HS, 1;

LoopTo 2;
//It is the general structure of a sequence file
With this sequence, the first block is used for initialization. The second block will generate a burst with the payload given by the pattern file “CJPat.txt” and followed by 20 DIFN states. This second block will be looped infinitely by the “LoopTo 2” entry in the sequence. The analyzer will get the final block to compare with the incoming data. Therefore in the case of the line loop back mode only a loop over the last block makes sense. Using a “//” symbol, comments can be added to a sequence file.

3.4 Unipro Marcros

For the Unipro specification, the M-PHY Frame Generator provides a set of macros that will create frames for the Unipro PHY Adapter layer (PA). They are part of Option 367, UniPro Error Counter and Test Script Wizard.

One type of frame that can be generated is the PACP (PA Control Primitive). It has very specialized usage such as to change the Power Mode, exchange of capability information, remote Get/Set and M-PHY testing. The FrameGenerator will create the PACP structure automatically as shown in the folliowing table:

Table 1:

16	15	14	12	11	10	9	8	7	6	5	4	3	2	1	0
1			ESC_PA					Esc_Param_PA =PACP_BEGIN							
0	PACP_FunctionId														
0	Parameters														
0	....														
0	CRC														

3.4.1 PwrReq

PwrReq(uint_8 DevID, uint_8 Flags, enum TxMode, uint_8 TxLane, enum TxGear, enum RxMode, uint_8 RxLane, enum RxGear, char[] UserDataArray) will create a PACP) frame and internally will call the PACKET Macro of Protocol to generate the Unipro packet for Power Change request. The parameters are:

- DevID: Local Device ID
- Flags
    - Flags[5]: Scrambling request: set to ‘1’ if scrambling is requested (PA_Scrambling).
    - Flags[4]: UserDataValid, set to ‘1’ if the frame contains valid PAPowerModeUserData
    - Flags[3]: HS Series, set to ‘0’ for Series A and to’1’ for Series B (PA_HSSeries)
    - Flags[2]: LINE-RESET request
    - Flags[1]: TX-direction termination enable (PA_TxTermination)
    - Flags[0]: RX-direction termination enable (PA_RxTermination)
- TxMode and RxMode can be chosen between the following values: HS_MODE, HS_MODE_auto, LS_MODE, LS_MODE_auto, HIBERN8, UNCHANGED.
- TxLane and RxLane: Active Lane count for TX/RX-direction
- TxGear and RxGear parameter can take different values such as: PWM_G1, PWM_G2, PWM_G3, PWM_G4, PWM_G5, PWM_G6, PWM_G7, HS_G1, HS_G2, HS_G3
- UserDataArray: user-data for peer DME (Device Management Entity).

Example:

PwrReq(DevID=0x00, Flags=0x10, TxMode=HS_MODE, TxLane=0x01, TxGear=PWM_G1, RxMode=HS_MODE, RxLane=0x01, RxGear=PWM_G1, 0x01, 0x11) then output = MK1 (ESC_PA) + 0x01 (PACP_BEGIN) + 0x010E (PACP_FunctionID for Power Mode change request) + 0x00 (DevID) + 0xD0 (Flags) + 0x2929 (Tx and Rx configuration) + 0x01 11 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 (UserDataArray = 12 words of 16 bits) + CRC

3.4.2 SetReq

SetReq(bool IsCnf, bool IsStatic, uint_8 MIBAttributeID, uint_8 MIBAttributeValue, uint_8 GenSelectIndex) will create a PACP Set request to set the value of an attribute. The parameters are:

- IsCnf: defines the behavior of the receiving PA Layer
  - false: no PACP_SET_cnf frame shall be sent
  - true: a PACP_SET_cnf frame shall be sent to acknowledge the reception of the PACP_SET_req frame and to return the result of the operation
- IsStatic: defines the target Attribute type (AttrSetTypeType)
  - false: NORMAL
  - true: STATIC
- MIBAttributeID: defines the Attribute to be accessed in the peer Device
- MIBAttributeValue: holds the value which the Attribute shall take
- GenSelectIndex: the index required for certain Attributes

Example:

SetReq(IsCnf=false, IsStatic=false, MIBAttributeID=0x 00 21, GenSelectIndex=0x 00 00, MIBAttributeValue=0x 00 02) then output MK1 (ESC_PA) + 0x01 (PACP_BEGIN) + 0x08 05 (PACP FunctionID to set the value of an attribute) + 0x3F FF (IsCnf and IsStatic) + 0x00 21 (MIBAttributeID) + 0x00 00 (GenSelectIndex) + 0x00 00 00 02 (MIBAttributeValue) + CRC.

3.4.3 GetReq

GetReq(uint_16 MIBAttributeID, uint_16 GenSelectIndex) macro will create a PACP Get request to get the value of an attribute.

The parameters are:

MIBAttributeID: defines the Attribute to be accessed in the peer Device

GenSelectIndex: the index required for certain Attributes

Example: Get the Tx Mode value.

GetReq(MIBAttributeID=0x0021, GenSelectIndex=0x0000) then output MK1 (ESC_PA) + 0x01 (PACP_BEGIN) + 0x06 02 (PACP FunctionID for request the value of an attribute) + 0x00 21 (MIBAttributeID) + 0x00 00 (GenSelectIndex)+ CRC.

3.4.4 TestModeReq

TestModeReq() will create a PACP test mode request.

Example:
TestModeReq() then output = 0x MK1 (ESC_PA) + 0x01 (PACP_BEGIN) + 0x05 00 (PACP FunctionID to set PA Layer in PHY testing mode) + CRC

3.4.5 TrgUpro0

TrgUpr0(uint_8 TxLaneNumber) macro sets the UniPro trigger TRG_UPR0. This trigger has a 2-bit parameter representing the physical lane number that the trigger was transmitted on.

Example:
TrgUpr0(TxLaneNumber=1) then output= MK1, 0x41

3.4.6 TrgUpro1

TrgUpr1(uint_8 TxConnectedLaneMask)TrgUpr1(uint_8 TxConnectedLaneMask) macros sets the UniPro trigger TRG_UPR1. This trigger has a 4-bit parameter containing information regarding connected lanes in a bitmap format.

Example:
TrgUpr1(TxConnectedLaneMask=3) then output = MK1, 0x83

3.4.7 TrgUpro2

TrgUpr2() macro sets the UniPro trigger TRG_UPR2. This is mapped to an escaped PA_PDU with EscParam set to 0xC0. This translates to <MK1, 0xC0>. The trigger is transmitted on all available Lanes.
Example:
TrgUpr2() then output = MK1, 0xC0

3.4.8 DataFrame

DataFrame(Enum TC, uint_8 DeviceID, uint_8 PortID, bool IsFCT, Char[] Payload) will create the Data Frame for Unipro. Each data frame can contain maximum of 256 byte application payload. If payload is more than 256 byte this macro creates multiple bytes and increments the frame sequence number for each frame. The parameters are:

- TC: sets the Unipro traffic class. It can be chosen between TC0, TC1, TC2 or TC3.
- DeviceID: It can be selected from 0 to127 (7bits)
- PortID: It can be selected from 0 to 31 (5bits)
- IsFCT: If false, no Flow Control Token is sent/received. If is true, Flow Control Token is carried by the PDU.
-

Example:

DataFrame(TC=TC0, DeviceID=0x02, PortID=0x01, IsFCT=false, Payload={...})

then output = 0x MK0 (ESC_DL) + 0x07 (StartOfFrame=000 + TC + reserved bits=111) + 0x82 (L3s=1 + DeviceID) + 0x85 (L4s=1 + PortID + IsFCT + EndOfMessage) + Payload + MK0 (ESC_DL) +CRC.....

3.4.9 TestDataFrame

TestDataFrame(uint_16 Id, bool IsCJTPAT) will generate the test pattern based on function Id.

Function Id can be as follows: 0x0A3F - PACP_TEST_DATA_0,
0x0A7D - PACP_TEST_DATA_1,
0x0ABD - PACP_TEST_DATA_2,
0x0AFD - PACP_TEST_DATA_3

The PA Layer supports two Test Patterns, CJTPAT and CRPAT with four variations for different lane counts.

Example: Unipro CJTPAT test data example.

TestDataFrame(Id=0x 0A 3F, IsCJTPAT=true) then output = 0xFE (ESC_PA) + 0x01 (PACP_BEGIN) + 0x0A 3F (Function Id) + CJTPAT_1[63] + CRC[0xAE43]

3.5 LLI Macros

The following macros can be used for devices that support LLI (Low Latency Interface) point to point interconnection to communicate. They are included as part of Option 368, Protocol-specific macros for LLI, SSIC and DigRF v4.

3.5.1 SVCFrame

SVCFrame(Uint_8 Crd, bool IsResponse, bool IsRead, uint_16 Addr, uint_32 Data) will create the SVC transaction. This is used for reading and writing control and configuration attributes. There are defined four Packet Formats for Service Transaction:

- Service Read Address Request Packed: Is used to read data from the remote Device. To send this packed type IsResponse parameter must be false and IsRead true. The addr parameter is required to specify the address.
- Service Write Address Request Packed: Is used to write data to the remote Device. To send this packed IsResponse and IsRead must be false. The addr parameter is required to specify the address. The data filed must contain the Write Data.
- Read Data Response Packed: Is used to read data from the remote Device. To send this packed IsResponse and IsRead parameters must be true. The data filed must contain the Read Data.
- Write Status Response Packed: Is used to get the status of a previous write request from the remote Device. To send this packed IsResponse must be true and IsRead false.

The Crd parameter indicates the number of flow control Credits sent to the Credit counter indicated by the CCI (Credit Counter Index). The Data and the Addr fields are not used in all Packed types but they cannot be empty.

Seq and CRC will be calculated programatically.

Example1: Service Read Address Request Packed

SVCFrame(Crd=0x02, IsResponse=false, IsRead=true, Addr=0x4520, Data=0x02)

then output = 0x22 (ChID=010 + CCI + Crd + IsResponse) + 0x00 (IsRead=true) + 0x00 00 00 00 00 00 + 0xA204 (Address) +Seq + CRC

Example: 2 Service Write Address Request Packed

SVCFrame(Crd=0x02, IsResponse=false, IsRead=false, Addr=0x4520, Data=0x02)

then output = 0x22 (ChID=010 + Crd + IsResponse) + 0x01 (IsRead=false) + 0x00 00 00 40 (Data) + 0x00 00 + 0xA204 (Address) +Seq + CRC

Example 3: Read Data Response Packed

SVCFrame(Crd=0x02, IsResponse=true, IsRead=true, Addr=0x4520, Data=0x02)

then output = 0x45 (ChID=010 + Crd + IsResponse) + 0x00 (IsRead=true) + 0x00 00 00 40 (Data) + 0x00 00 00 00 +Seq + CRC

Example 4: Write Status Response Packed

SVCFrame(Crd=0x02, IsResponse=true, IsRead=false, Addr=0x4520, Data=0x02)

then output = 0x45 (ChID=010 + Crd + IsResponse) + 0x01 (IsRead=false) + 0x00 00 00 00 00 00 00 00 +Seq + CRC

3.5.2 TLFrameCmdReq

The macro TLFrameCmdReq(uint_8 ChID, uint_8 Crd, bool IsLast, uint_8 OrdID, enum Opc, enum BT, uint_8 Len1, bool IsT, bool IsS, bool IsP, bool IsD, uint_16 User, double Addr) sends the Interconnect Command Request Packet that is used to initiate a write or read request to or from the remote Device. The configurable parameters are:

- ChID: Must be 4 or 6 for this Frame type
- Crd: Indicates the number of flow control Credits sent to the Credit counter indicated by the CCI field
- IsLast: The IsLast field shall indicate the last Packet of a Transaction Unit when set to '1'
- OrdID: The OrdID field shall contain the Transaction ordering identifier. Response Units are tagged with the same Ordering Identifier.as their corresponding Request Units
- Opc:The Opc field shall contain the opcode of a Transaction. It can be choosen between:
    - RD (Read Transaction)
    - BAR (Reserved Barrier)
    - WRNP (Write, Non-Posted Transaction)
    - WRP (Write Posted Transaction)
    - RSV0 (Reserved User 0 Transaction)
    - RSV1 (Reserved User 1 Transaction)
    - RSV2 (Reserved User 2 Transaction)
    - ESC (Reserved LLI Escape Transaction)
- BT: The BT field shall contain the burst type. It can be choosen between:
    - INCR (Incrementing Burst Transaction)
    - WRAP (Wrapping Burst Transaction)
    - Reserved
- Len1 Shall contain the transaction size (length) minus one, expressed in bytes.
- IsT: This field shall indicate the transaction type using the values: Data Access = '0' or Instruction Access = '1' It is named ReqAttrType
- IsS: This field shall indicate the transaction security level using the values: Secure Access = '0' or Insecure Accsess = '1' . It is named ReqAttrSecureN
- IsP: This field shall indicate the transaction privilege using the values: User Access = '0' or Kernel Access ='1'. It is named ReqAttrPrivilege
- IsD: This field shall indicate a debug agent transaction using the values:.Normal System Access = '0' or Debug Agent Access = '1'. It is named ReqAttrDebug.
- User: The User field shall be used for in-band user defined supplementary information
- Addr: Byte address of the first byte targered by the transaction

Seq and CRC will be calculated programatically. This frame has to be followed by a sequence of TxFrameWReq if Opc field value is either WRP or WRNP.

Example:Command request for writing data at starting address=0x01.

TLFrameCmdReq(ChID=0x04, Crd=0x02, IsLast=false, OrdID=0x01, Opc=WRP, BT=INCR, Len1=0x0F, IsT=false, IsS=true, IsP=false, IsD=false, User=0x1FF, Addr=0x01)

then output = 0x21 A0 81 07 F9 0F 00 00 00 80 (ChID + CCI+ Crd + IsLast +OrdID + Opc + BT + Len1 + RSV ="0000" + IsT + IsS + IsP + IsD + User + Addr) + Seq +CRC

3.5.3 TLFrameWDataReq

TLFrameWDataReq(uint_8 ChID, uint_8 Crd, bool IsLast, uint_8 Be, char [] WData)sends the Interconnect Write Data Request Packet that is used to write data to the remote Device, after sending the write command request. Write data is sent through this macro with the parameter WData. Seq and CRC will be calculated programatically. The configurable parameters are:

- ChID: Must be 4 or 6 for this Frame type
- Crd: Indicates the number of flow control Credits sent to the Credit counter indicated by the CCI field
- IsLast: The IsLast field shall indicate the last Packet of a Transaction Unit when set to '1'
- Be: The ByteEn field shall contain write byte enables, indicating which bytes within the WData are valid
- WData field shall contain the write data.
-

Example: Write data at address=0x01.

TLFrameWData (ChID=0x04, Crd=0x02, IsLast=true, Be=0xFF, Data={....})

then output = 0xA1 (ChID + CCI + Crd + IsLast) + 0xFF (Be) + Data(Payload max 64 bits) + Seq + CRC

3.5.4 TLFrameReadRsp

TLFrameReadRsp(uint_8 ChID, uint_8 Crd, int OrdID, bool Error, bool IsLast, char [] RData) sends the Interconnect Read Response Packet that is used to read data from the remote Device, after sending the write command request. Write data is sent through this macro with the parameter WData. Seq and CRC will be calculated programatically. The configurable parameters are:

- ChID: Must be 5 or 7 for this Frame type
- Crd: Indicates the number of flow control Credits sent to the Credit counter indicated by the CCI field
- OrdID: Transaction Ordering Identifier
- Error: Indicates if there was an error
- IsLast: The IsLast field shall indicate the last Packet of a Transaction Unit when set to '1'
- WData field shall contain the read data.

Example: Read data from address=0x01.

```
TLFrameReadRsp(ChID=0b111, Crd=0x01, OrdID=0b001000,
Error=false, IsLast=true, Data={…});
then output = 0x4F (ChID + CCI + Crd + 0) + 0x44 (OrdID + La
+ E) + Data(Payload max 64 bits) + 0x00 (Seq) + 0x11 (CRC)
```

3.5.5 3.5.5 TLExtendedFrame

TLExtendedFrame(int ChID, int Crd, bool IsLast, int OrdID, RawdataDescription[] WData) sends an Extended Frame that can either be carry Write Data Request or Read Data Response packets, depending on the specified ChID

- ChID
- Crd: Indicates the number of flow control Credits sent to the Credit counter indicated by the CCI field
- IsLast: The IsLast field shall indicate the last Packet of a Transaction Unit when set to '1'
- OrdID: Transaction Ordering Identifier, if carrying Read Data Response packets, otherwise these bits are Reserved
- WData field shall contain the data corresponding to four IC_WDATA_REQ_P or IC_RDATA_RSP_P Transaction Packets.

Example: `TLExtendedFrame(ChID=4, Crd=1, IsLast=false, OrdID=0b111111, Data);`

3.5.6 DLMessageFrame

DLMessageFrame(Uint_8 TCrd3, Uint_8 TCrd2, Uint_8 TCrd1, Uint_8 TCrd0) will create a MessageFrame. This is used for exchanging the flow control information. Seq and CRC will be calculated programatically.

Each TCrdX parameter indicates the number of flow control Credits send to Credit Counter X.

Example:

DLMessageFrame(TCrd3=0x00, TCrd2=0x00, TCrd1=0x00, TCrd0=0x07)

then output = 0x06E0 (ChID = "011" + C=0 + TCdr3 + TCdr2 + TCdr1 + TCdr0) + 0x00 00 00 00 00 00 00 00 + seq + CRC

3.5.7 PAM

The PAM(int MsgTyp) macro will Create a PAM message. Request for Physical_Link_Update (PLU) sequences is generated with MsgTyp = 0 and PA_MEDIATEST_START with MsgTyp = 1. Seq and CRC will be calculated programatically.

Example: PAM (PHY- Adapter Layer) message example.

PAM() then output = CRC + 0x0A 00 00 00 00 00 00 00 00 00 00

3.5.8 TestPattern

TestPattern(bool TestPatternSelect) will generate the test pattern. If the TestPatternSelect is set to true the CRPAT is sent, if the TestPatternSelect is set to false the CJTPAT is sent.

Sending in burst mode or continuous mode is outside of the scope of the macro. The mode needs to be defined at the sequence level.

Example: TestPattern(TestPatternSelect=true).

3.6 DigRF Macros

For DUTs that support the DigRF interface the following macros will generate DigRF messages. These macros are included as part of Option 368, Protocol-specific macros for LLI, SSIC and DigRF v4.

3.6.1 ICLC

ICLC(bool IsRTI, uint_8 Command, uint_8 Argument) will be used to create an ICLC (Interface Control Logical Channel) message frame.

The IsRTI field will enable/disable the Retrasmission Indicator. CLC_ID (Control Logical Channel) will be 000b for this frame (Tx Interface Control Logical Channel / Rx Interface Control Logical Channel) . For Command and Argument parameters refer to Table 25 “ICLC Messages” of the “Mipi Specification for DigRF” (See Identifier and Arguments columns). CRC will be taken care in macro Implementation

Example: Create a ICLC frame to Enable Hibernate Mode:

ICLC(IsRTI=false, Command=0x11, Argument=0x01)

then output = 0x00 (IsRTI + CRI (Cyclic Running Index)+ CLC_ID=“000” + LC Indicator=0) + 0x11 (Command) + 0x01 (Argument)+ CRC.

3.6.2 Dummy

Dummy() macro is used to generate the Dummy ICLC message frame. CRI will be taken care in macro Implementation.

Example:

Dummy() then output = 0x00 (RTI=0 + CRI + CLC_ID=“000” + LC Indicator=0) + 0x96 00 + CRC..

3.6.3 TrgT

TrgT() will create a configuration trigger (TRG-T). This is one kind of ICLC message. This macro has to be called after some of ICLC messages in TxData sublink. CRC will be taken care in macro Implementation.

Example:

TrgT() then output = 0x00 (RTI=0 + CRI + CLC_ID=“000” + LC Indicator=0) + 0x08 3D + CRC.

3.6.4 TrgR

TrgR() will create a configuration trigger response (TRG-R). This is one kind of ICLC message. This macro has to be called after some of ICLC messages in RxData sublink. CRI will be taken care in macro Implementation.

Example:
TrgR() then output = 0x00 (RTI=0 + CRI + CLC_ID="000" + LC Indicator=0) + 0x09 D3+ CRC.

3.6.5 DataFrame

DataFrame(bool IsRTI, uint_8 DLC_ID, chr[] payload, uint_16 SDLC_ID = 0, uint_16 CIL = 0) will be used to create the Data frame. The parameters are:
IsRTI: This field will enable/disable the Retrasmission Indicator.
DLC_ID: Sets the Data Logical Channel ID
SDLC_ID: Shal be utilized for the numbering of additional logical channels.
CIL:(Compression Information Lenght): shall represent the number of bytes in the payload used for the data compression information CI.
CRC will be taken care in macro Implementation.

Example:
DataFrame(IsRTI=false, DLC_ID=0x01,payload={....})
then output = 0x53+Payload+CRC.

3.6.6 TestPattern

TestPattern() will create the test pattern.

3.7 SSIC Macros

M-PHY also supports the SuperSpeed Inter-Chip (SSIC) specification. The Frame Generator provides some macros to send Remote Register Access Protocol (RRAP) messages. While in PWM-BURST mode, communication is done with the following RRAP packets:

3.7.1 WriteCommand

WriteCommand(uint_8 Data, uint_8 LowerAddr, uint_8 UpperAddr)

3.7.2 ReadCommand

ReadCommand(uint_8 LowerAddr, uint_8 UpperAddr)

3.7.3 WriteResponse

WriteResponse()

3.7.4 ReadResponse

LoopBackEnable(bool Enable)

3.7.5 LoopbackEnable

ReadResponse(uint_8 Data)

3.7.6 ScramblingEnable

ScramblingEnable(bool Enable)

3.7.7 LineReset

LineReset(double Time)

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

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 bits, 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 bits.

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;
```

Edition 2.0, May 2016
N5990-91140
www.keysight.com