Voyager Exerciser Language

Reference Manual

Manual Version 1.0

December 28, 2020

For USB Protocol Suite version 8.50 Build 3675 and above

Document Disclaimer

The information contained in this document has been carefully checked and is believed to be reliable. However, no responsibility can be assumed for inaccuracies that may not have been detected.

Teledyne LeCroy reserves the right to revise the information presented in this document without notice or penalty.

Trademarks and Servicemarks

CATC Trace, Voyager M3i, Voyager M3x, Voyager M310, Voyager M310C, Voyager M310P, Voyager M4x, Voyager ReadyLink, USB Protocol Suite, and *BusEngine* are trademarks of Teledyne LeCroy.Inc.

All other trademarks are property of their respective companies.

Copyright

1 Contents

1. Introduction

The Voyager Exerciser Language provides rich language and preprocessor capabilities and allows you to implement even complicated generation scenarios.

Typically, Voyager Exerciser Language constructions do not require special separation symbols (such as the semicolon in C languages) to distinguish between different constructions. Where possible, script parsing uses "context-dependency" rather than separation symbols, thus simplifying writing of generation scripts.

Also, Voyager Exerciser Language constructions are not case-sensitive.

This document describes the syntax of the Exerciser Generation Script Language.

You can start the exerciser script editor by clicking that button in the USB Protocol Suite application to display an editor user interface, in which you can create and edit generation script files, as well as reuse codes.

Each running module must have at least one function named **Main**, which starts and ends the generation script.

1.1 Declaration Conventions

1.1.1 Parentheses

In declarations and descriptions of Voyager Exerciser Language instructions,
the format **"(" ")"** specifies a list of keywords that have the same value and can replace each other. The declaration

(Packet | **Struct) Template_Name …**

specifies that you can use the "Packet", or "Struct" keyword for template name declarations.

1.1.2 Brackets

In declarations and descriptions of Voyager Exerciser Language instructions,
the format "**[**" "**]**" specifies optional parts of declarations or instruction parameters. The declaration

(Packet | **Struct) Template_Name [** Ancestor_1, Ancestor_ 2, … **] [** (Attribute list) **]**

specifies that the listed ancestors and attribute lists are optional for template name declarations.

1.2 Script Example Highlighting

Voyager Exerciser Language examples in this document show syntax highlighting similar to that in the Generation Script Editor.

2 Script Language Structure

2.1 Generation Script Structure

Typically, a generation script has the following structure:

- Parser directives
- Declarations
  - Constants
  - Data patterns
  - Global generation settings
  - Packet/structure templates
  - Global numeric variables
  - Global structure variables (declare a template for a variable before declaring a variable)
  - Aliases
- Generation procedures
  - List of generation instructions
  - Global declarations (constants, data patterns, templates, aliases)

Note: The parser can use previously declared objects in later declarations. In generation procedures, the parser can use global objects before their declaration as long as such objects are finally declared outside of generation procedures.

Reminder: The generation parser is NOT case-sensitive.

2.2 Main Procedure and Other Procedures

Although you can create many generation procedures, the major execution entry point is a procedure with the name **Main**. Therefore, you must have a generation procedure named **Main**. You can invoke the other generation procedures in the **Main** generation procedure using the Call directive.

The **Call** directive makes a "dynamic" insertion, in which the included procedure is re-parsed using the new parser variable values and the latest values of global variables.

3 Comments

Comments instruct the script parser to exclude the commented parts of the script file from parsing.

3.1 Line Comment

To comment a line, start the line with the symbol **#**.

To comment the end of a line, put the symbol **#** before the comment. The parser ignores the rest of the line after the **#** symbol.

Example

```
SomeStuff # The text after the # symbol is not parsed.
```

3.2 Block Comment

To comment a block of text, start with the symbol pair **/*** and end with the symbol pair ***/**. The parser ignores the part of the file inside the comment block.

Example

```
/*
    Example of a block of comments
    All text between '/' '*' and '*' '/' is ignored.
*/
```

4 File-Including Directives

To include a file in a generation parsing stream, use the **%inline** or **%include** directive.

4.1 Inline Directive

The **%inline** directive instructs the script parser to insert the content of the named file into the parsing stream when the parser sees this directive, even if the file is already inserted.

Examples

```
%inline "SomeInc1.inc" # Includes the 'SomeInc1.inc' file.
%inline "SomeInc1.inc" # Includes the 'SomeInc1.inc' file again.
```

4.2 Include Directive

The **%include** directive instructs the script parser to insert the content of the named file into the parsing stream only ONCE, the first time the parser sees the directive with the specified file name.

Examples

```
%include "SomeInc1.inc" # Includes the 'SomeInc1.inc' file.
%include "SomeInc1.inc" # Does nothing.
```

5 Constant Declarations

You can declare numeric script constants to use later in assignments or arithmetic operations. The limits of values of constants are from 0 to 0xFFFFFFFF (4294967295 decimal).
Constants are DWORD (unsigned integer) values only.

5.1 Constant Definition Examples

Examples

```
Const SOME_HEX_DATA    = 0xAABBFFEE # Define a hexadecimal constant.
Const SOME_DEC_DATA    = 12         # Define a decimal constant.
Const "SOME DEC DATA" = 64          # Define a decimal constant.
Const SOME_BIN_DATA    = _11110000  # Define a binary constant.

Const "Some Hex Data" = 0xCDCDBEBE
Const LMP_PORT_CFG_ACK = 6

# The parser can use arithmetic operations in constant definitions.
Const TX_PAYLOAD_OFFSET = 16 * 8    # Payload offset(in bits) for
                                    # the Tx packet

# Payload offset(in bits) for the Rx packet
Const SOME_CONST = 16
Const SOME_PAYLOAD_OFFSET = TX_PAYLOAD_OFFSET + SOME_CONST

# Example of a complex name in an expression
Const MY_Data = "Some Hex Data" + 12 - _1111000 /*binary literal*/
```

6 Data Pattern Declarations

Data pattern declarations declare named byte strings to use where you use byte vectors. Data pattern declaration starts with the **DataPattern** keyword.

```
DataPattern Pattern_Name = { hex_stream }
```

Example

```
      # Declare a data pattern containing the byte string:
      # AA BB CC DD DD EE FF 11 22 33 44 55
DataPattern PATTERN_1 = { AA BB CC DD EE FF 11 22 33 44 55 }
      # or
DataPattern PATTERN_1 = { AA BB CC DD EE _11111111 11 22 33 44 55 }
```

6.1 Constants and Data Patterns in Declarations

You can use constants and previously defined data patterns in data pattern declarations. Place constants and data patterns inside a **[]** block. (You can omit a **[]** block for constant insertion, but for code clarity it is recommended that you use a **[]** block.)

Note: When inserting constants into a data pattern, the script parser uses only the least significant byte of the constant. For example, if constant **0xAABBCCDD** is inserted, only the **0xDD** is put into the data pattern.

Examples

```
Const MY_CONST    = 0xCC
Const "MY CONST" = 0xDD

# Declare a data pattern containing the byte string:
# AA AA BB BB CC CC DD DD
DataPattern PATTERN_1 = { AA AA BB BB [MY_CONST] CC DD ["MY CONST"] }

# Declare a data pattern containing the byte string:
# 11 AA AA BB BB CC CC DD DD 88
DataPattern PATTERN_2 = { 11 [ PATTERN_1 ] 88 }
```

6.2 Leading Zeroes

For bytes less than 0x10, it is not necessary to add a leading 0.

Example

```
DataPattern PATTERN_4 = { B B 6 B B } # Is the same as 0B 0B 06 0B 0B.
```

7 Packet and Structure Template Declarations

Packet/structure declarations declare named packet/structure objects. A packet/structure object gives its target byte stream a set of "fields", each having a unique name. You can fully set up a byte stream using packet/structure fields.

Note: Templates can inherit field layouts from other templates through ancestor lists.

(Packet | Struct) Template_Name [: Ancestor_1, Ancestor_ 2, … **] [(**Attribute list**)]**

```
[
{
    Field Definition 1
    Field Definition 2
    ...
}
]
```

Note: For packet/structure template declarations, you can use the **Packet** or **Struct** keyword. However, structures are supposed to be used as building blocks for constructing packet payloads (or similar purposes), rather than for describing full packets.

7.1 Field Definitions

You can define template fields using the following syntax:

Field_Name : [offset,] length [(Byte Order attribute)] [= Field_Value] [[list of possible values]]

or

Field_Name [: length] { Subfield definitions } [(Byte Order attribute) [= Field_Value] [[list of possible values]]

Note 1: Specify all field offsets and lengths in bits. Numeric expressions with declared constants are allowed in field offset and length definitions. All field and subfield names must be unique inside a template.

Note 2: You can define local fields for packet variable instances and when sending a packet.

7.1.1 Defining a Field at a Specific Offset

If the offset parameter is in a field definition, the field is bound to a specific offset.

Examples

```
const F3_OFFSET = 16
const F3_LEN    = 8

# Declare the packet template 'SomeTemplate'.
Packet SomeTemplate
{
    F1 : 0, 16            # Declare the 16-bit field 'F1' at offset 0.
    F2 : 64, 32           # Declare the 32-bit field 'F2' at offset 64.
    F3 : F3_OFFSET, F3_LEN # Declare the 8-bit field 'F2' at offset 16.
    F4 : F3_OFFSET + F3_LEN, 16 # Declare the 16-bit field 'F4' at
                                # offset 16+8.
}
```

7.1.2 Defining a Field at the Current Offset

If the field offset is omitted, the field's offset is calculated based on the lengths of previously declared fields. The initial template offset is always 0.

Example

```
const F3_OFFSET = 64
const F3_LEN    = 8

# Declare the packet template 'SomeTemplate'.
Packet SomeTemplate
{
    F1 : 16               # Declare the 16-bit field 'F1' at offset 0.
    F2 : 32               # Declare the 32-bit field 'F2' at offset 16.
    F3 : F3_OFFSET, F3_LEN # Declare the 8-bit field 'F2' at offset 64.
    F4 : 16             # Declare the 16-bit field 'F4' at offset 64+8.
}
```

7.1.3 Defining a Field with Variable Length

If the field length is set to *, the field's length is defined by the value that is assigned to the field. If no value is assigned to the variable length field, the field's length is set to 0.

Note: When a value is assigned to a variable length field, the field's length is changed based on the difference between the previous and current values. In this case, the offsets of following fields not bound to fixed offsets are shifted by the difference.

Examples

```
# Declare packet template 'SomeTemplate'.
Packet Some Template
{
    F1   : 16 # Declare the 16-bit field 'F1' at offset 0.
    F2   : 32 # Declare the 32-bit field 'F2' at offset 16.
    Data : *  # Declare the variable-length field 'Data'. The field
              # length is now 0.
    CRC32  : 32 # Declare the 32-bit field 'CRC32' at offset 16+32.
}

# Declare packet template 'SomeTemplate1', inheriting fields from
# 'SomeTemplate'.
Packet Some Template1 : Some Template
{
    Data = 0xAABB  # Assign a value to the 'Data' field.
                   # Now the field has length 16 bits and the
                   # field 'CRC32' offset is shifted by 16 bits =
                   # 16+32+16.
}
```

7.1.4 Defining a Default Field Value

When defining a field, you can specify a default field value. If the default value is not provided, the field is filled with zeros based on the field length.

Note: When a value is assigned to a variable length field, the field's length is changed based on the difference between the previous and current values. In this case, the offsets of following fields not bound to fixed offsets are shifted by the difference.

Example

```
const F3_OFFSET = 64
const F3_LEN    = 8

# Declare the packet template 'SomeTemplate.
Packet SomeTemplate
{
    F1 : 16          # Declare the 16-bit field 'F1' at offset 0.
    F2 : 32 = 123456 # Declare the 32-bit field 'F2' with default
                     # value 123456.

    F3 : * = {AA BB} # Declare a variable length field and assign
                     # hex value {AA BB} to it.
                     # Now its length is 16 bits.
}
```

7.1.5 Specifying Byte Order in Field Definitions

You can specify the byte order for integer fields (length <= 32 bits) using the **Byte Order** field attribute. The **Byte Order** field attribute indicates how numeric values are assigned to integer fields. By default, the byte order for integer fields is Big Endian: MSB->LSB. For example, the integer value **0xAABBCCDD** is assigned as the **{AA BB CC DD}** byte stream.

Example

```
# Specify byte order for some fields of a template.
Packet Mixed
{
    F1 : 16
    F2 : 32 (MSB)
    F3 : 16 (MSB) = 0xAABB
    F4 : 32       = 0xAABBCCDD
}

# Template with the same field layout as the template above
Packet MSBMixed (MSB)
{
    F1 : 16 (LSB)
    F2 : 32
    F3 : 16       = 0xAABB
    F4 : 32 (LSB) = 0xAABBCCDD
}
```

7.1.6 Using Byte Stream Literals in Field Assignments

You can specify the byte order explicitly using byte stream literals:

Example

```
Field_32 : 32 = { AA BB CC DD }
```

7.1.7 List of Possible Field Values Attribute

You can specify a list of possible field values (defined by constant or data pattern names) for declared template fields. The Voyager USB 3.1 Exerciser Application Development Environment (Voyager USB 3.1 Exerciser Script Editor Intellisense) uses this list to quickly assign field values.

Note: The list does not affect compilation or traffic generation.

Example

```
Const MyConst = 10

DataPattern MyPattern = { AA BB CC DD _11110000 }

Packet MY_TEMPLATE
{
    Field_1 : 16 [MyConst, MyPattern]         # possible value list
    Field_2 : 32 = 0xAABBCCDD [MyPattern]     # possible value list
    Field_3 : 32 (MSB) = [MyConst]            # possible value list
}
```

7.1.8 Defining Subfields

You can define named subfields for top-level template fields. Subfields allow you to set a field value using DWORD little-endian bit order. Subfield and field names must be unique inside a template. You can also assign field values using field names directly.
Note: You cannot define subfields for lower-level template fields.

Syntax

Field_Name [: length]
{
SubField [: length] [(Byte Order attribute) [= Field_Value] [[list of possible values]]
SubField [: length] [(Byte Order attribute) [= Field_Value] [[list of possible values]]
SubField [: length] [(Byte Order attribute) [= Field_Value] [[list of possible values]]
…
}
[(Byte Order attribute) [= Field_Value] [[list of possible values]]

If the length of a parent field is less than the total length of its subfields, the total length of the subfields defines the parent field length.

Note: The subfields always use their parent field as a DWORD little-endian buffer. For example, if the parent field has a 64-bit length for subfield assignments, the subfields use it as two little-endian DWORDs.

Example

```
struct Templ_1
{
    F1 : 8
    F2
    {
        SF1 : 8 = 0xCC
        SF2 : 16
        SF3 : 8
    }
}

struct Templ_2
{
    F1 : 8
    F2 : 16      # Declare parent field length.
    {
        SF1 : 8
    }
}
Main
{
      # Send a packet with payload: 00 EE AA BB CC
      Send Templ_1
      {
          SF2 = {AA BB}
          SF3 = 0xEE
      }
      # The same as above
      Send Templ_1
      {
          # Use direct parent field assignment instead of subfields.
          F2 = {EE AA BB CC}
```

```
        }
        # Send a packet with payload: 0A 00 EE
        Send Templ_2
        {
            F1  = 0xA
            SF1 = 0xEE
        }
}
```

7.2 Constants/Arithmetic Expressions in Template Declarations

You can use constants and arithmetic expressions in both field definitions and value declarations.

Examples

```
const DPH = _1000 # 0x8

struct LinkCtrlWord
{
    Hseq    : 3
    LcwRsvd : 3
    HDepth  : 3
    D1      : 1
    D2      : 1
    CRC_5   : 5 # Auto calculated
}

const TYPE_LEN = 5

Packet DPH
{
      #Dword 0
      Type     : TYPE_LEN = DPH
      RouteStr : 20
      DevAddr  : 7

      #Dword 1
      SeqNum   : 5
      Rsvd1    : 1
      Delayed  : 1
      EOB      : 1
      Endp     : 4
      Rsvd2    : 3
      Setup    : 1
      DataLen  : 16

      #Dword 2
      StreamID : 16
      Rsvd3    : 11
      PP       :  1
      Rsvd4    :  4

      #Dword 3
      CRC_16   : 16          # Auto calculated
      : LinkCtrlWord         # structure template insertion
}

const CRC32_LEN = 32

Packet DataPacket : DPH
{
      Data     : *
      CRC32    : CRC32_LEN
}
```

7.3 Packet Template Inheritance

You can create a packet/struct template that inherits field layouts defined in another template. The created template has all the fields defined in the inherited templates plus its own fields. All fields must have unique names.

Note: The parser adds the fields defined in the created packet template after it adds fields from the inherited templates.

You can change the default field values of the inherited fields.

7.3.1 Packet Template Single Inheritance

A new template inherits the field layouts from another template using an ancestor list.

Examples

```
Packet MY_DataPacket : DataPacket
{
    # Example of a field with variable length having a default value.
    Data : * = { 00 00 00 01 }
}

# More examples of template inheritance
const LMP = 0
const LMP_SET_LINK_FUNC = 1

Packet USB3Common
{
    Type : 5
}


Packet LMPCommon : USB3Common # Derive a template from a USB3Common
                              # template.
{
    Type = LMP
    SubType : 4
}

Packet LMP_SetLinkFunction : LMPCommon
{
    SubType = LMP_SET_LINK_FUNC

    Resv1                 :  1;
    Force_LinkPM_Accept   :  1;
    Resv2                 :  5;
    Resv3                 : 16;

    Resv4                 : 32;
    Resv5                 : 32;

    CRC_16                : 16 # Auto calculated
    : LinkCtrlWord             # structure template insertion
}
```

7.3.2 Packet Template Multiple Inheritance

You can create a packet/struct template that inherits field layouts defined in several templates. The created template has all the fields defined in the inherited templates plus its own fields. All fields must have unique names.

Multiple inheritance can simplify the construction of complex templates.

Note: The parser adds the fields defined in the created template after it adds fields from the inherited templates. The parser adds the fields defined in the inherited templates in order from the left-most ancestor to the right-most ancestor.

Examples

```
Packet Base
```

```
{
    F1 : 16
    F2 : 8
    F3 : 32
}

Packet Templ_0
{
    FieldT0_8  : 8
    FieldT0_16 : 16
}

Packet Templ_1
{
    FieldT1_24 : 24
    FieldT1_32 : 32
}

Packet Combined : Base, Templ_0, Templ_1
{
}
```

The Combined template above has the fields:

- `F1 : 16 # Base`
- `F2 : 8 # Base`
- `F3 : 32 # Base`
- `FieldT0_8 : 8 # Templ_0`
- `FieldT0_16 : 16 # Templ_0`
- `FieldT1_24 : 24 # Templ_1`
- `FieldT1_32 : 32 # Templ_1`
- `Data : * # Combined`

Examples

```
const DPH = _1000 # 0x8

struct LinkCtrlWord
{
    Hseq    : 3
    LcwRsvd : 3
    HDepth  : 3
    D1      : 1
    D2      : 1
    CRC_5   : 5 # Auto calculated
}

struct DPH_DWORD_0
{
     Type     : 5 = DPH
     RouteStr : 20
     DevAddr  : 7
}

struct DPH_DWORD_1
{
     SeqNum   : 5
     Rsvd1    : 1
     Delayed  : 1
     EOB      : 1
     Endp     : 4
     Rsvd2    : 3
     Setup    : 1
     DataLen  : 16
}

struct DPH_DWORD_2
{
     StreamID : 16
     Rsvd3    : 11
     PP       :  1
     Rsvd4    :  4
}

Packet DPH : DPH_DWORD_0, DPH_DWORD_1, DPH_DWORD_2 # Multiple template
                                                   # inheritance
{
    CRC_16    : 16 # Auto calculated
    : LinkCtrlWord # structure template insertion
}
```

7.3.3 Packet Template Insert Directive

You can insert field layouts from another template after a specific field in a template. Use the **insert** or **':'** directive.

Example 1

```
Packet Base
{
    F1 : 16
    F2 : 8
    F3 : 32
}

Packet Templ_0
{
    FieldT0_8  : 8
    FieldT0_16 : 16
}

Packet Templ_1
{
    FieldT1_24 : 24
    FieldT1_32 : 32
}

Packet Combined : Base
{
    Cmb_F1 : 8
    insert Templ_0 # Insert fields from packet template Templ_0.
    Cmb_F2 : 16
    insert Templ_1 # Insert fields from packet template Templ_1.
    Data   : 32
}
```

The **Combined** template above has the fields:

```
F1          16  # Base
F2           8  # Base
F3          32  # Base
Cmb_F1       8  # Combined
FieldT0_8    8  # Templ_0
FieldT0_16  16  # Templ_0
Cmb_F2      16  # Combined
FieldT1_24  24  # Templ_1
FieldT1_32  32  # Templ_1
Data        32  # Combined
```

Example 2

```
Packet Combined : Base
{
    Cmb_F1 : 8
    : Templ_0  # Insert fields from packet template Templ_0.
    Cmb_F2 : 16
      : Templ_1  # Insert fields from packet template Templ_1.
    Data   : 32
}
```

7.3.4 Complex Packet Templates

Template insertions can simplify construction of complex templates.

Examples

```
const DPH = _1000 # 0x8

struct LinkCtrlWord
{
    Hseq    : 3
    LcwRsvd : 3
    HDepth  : 3
    D1      : 1
    D2      : 1
    CRC_5   : 5 # Auto calculated
}

struct DPH_DWORD_0
{
     Type     : 5 = DPH
     RouteStr : 20
     DevAddr  : 7
}

struct DPH_DWORD_1
{
     SeqNum   : 5
     Rsvd1    : 1
     Delayed  : 1
     EOB      : 1
     Endp     : 4
     Rsvd2    : 3
     Setup    : 1
     DataLen  : 16
}

struct DPH_DWORD_2
{
     StreamID : 16
     Rsvd3    : 11
     PP       :  1
     Rsvd4    :  4
}




Packet DPH
{
    : DPH_DWORD_0
    : DPH_DWORD_1
    : DPH_DWORD_2 # Use template insertion instead of multiple
                  # inheritance.
```

```
        CRC_16    : 16 # Auto calculated
        : LinkCtrlWord # Another template insertion
    }
```

7.4 Packet Template Multi-byte Field Byte Order Attribute

By default, for fields up to 32 bits, the parser uses the Little Endian (LSB -> MSB) byte order. For example, if a 32-bit field has the value 0xAABBCCDD, it is written into the byte stream as:
`DD CC BB AA`

Example

```
Packet MY_TEMPLATE
{
    Field_16 : 16 = 0xEEFF
    Field_32 : 32 = 0xAABBCCDD
}
```

The byte stream based on this template is:

`FF EE DD CC BB AA`

7.4.1 Big Endian Byte Order

You can require that all template fields have the Big Endian (MSB -> LSB) byte order.
Adding **(MSB)** after the ancestor list in the template declaration instructs the script parser to choose the MSB -> LSB byte order.

Example

```
Packet MY_TEMPLATE (MSB)
{
    Field_16 : 16 = 0xEEFF
    Field_32 : 32 = 0xAABBCCDD
}
```

The byte stream based on this template is **`EE FF AA BB CC DD.`**

Note: This attribute is needed only for assignments in which numeric literals (such as 0xAABBCCDD or 12323344) are used.

7.4.2 Another Example

Example

```
struct UsbReq {LSB} # Using LSB attribute for integer field assignments
                    # instructs that all short fields have
                    # LSB to MSB byte order when assigned values.
{
      bmType    : 8
      bReq      : 8
      wValue    : 16
      wIndex    : 16
      wLen      : 16
      Data      : *   # variable length field
}
```

Note: The **Byte Order** attribute applies only for fields defined inside a template and does not affect fields specified in inherited or inserted templates.

Examples

```
Packet Base_0 (MSB)
{
    F1 : 16
}

Packet Base_1  # LSB to MSB byte order by default
{
    F2 : 32
}

Packet Combined : Base0, Base_1 (MSB)
# The attribute goes after the template ancestor list.
{
    Cmb_Fld : 32
}

# Combined template has the fields:
# F1      : 16 (MSB)
# F2      : 32 (LSB)
# Cmb_Fld : 32 (MSB)
```

7.5 Structure Declaration Examples

To the parser, structures are the same as packet templates. However, you should use structures to assign packet field values, rather than packet layouts in **Send Packet** instructions.

Note: Also see the examples using the **Struct** keyword to assign packet field values, in later sections.

Examples

```
struct S1
{
    F16 : 16 = 0xAABB
    F8  :  8 = 0xFE
    F32 : 32 = 0xABCD1234
}

struct S2
{
    S8  : 8  = 0xDA
    S32 : 32 = 0x56781122
    S16 : 16 = 0xBEEF
}
```

8 Advanced Script Parser Features

The script parser has some advanced features that simplify creation of complicated generation scenarios. Such features include:

- Using local and global integer variables
- Using local and global structure variables
- Using special data pattern creators in field assignments
- Using multipliers with structure variables in field assignments
- Using **= +** in field assignments
- Initializing structure variables from hex streams
- Supporting arithmetic operations with parser variables
- Supporting concatenation operations for structure variables
- Using parser **sizeof** operators
- Using parser **while, for**, and **if-else** operators
- Calling generation procedures with parameters
- Allowing users to trace/debug the parser and follow variable and template construction
- Supporting name aliasing for constants, settings, data patterns, templates, variables, instructions and named instruction parameters.
- Using **include** paths to specify additional folders in which to look for included files

8.1 Local Numeric Parser Variables

You can declare a local numeric parser variable, which is seen only inside the generation procedure, and use it in field assignments.

Format

```
[ Local ] var = expression
```

Note: The **Local** keyword is required if you already have a global variable with the same name. Using this keyword explicitly instructs the script parser to declare a local variable with the same name.

Example

```
y = 10 # Declare a global numeric variable with name 'y'.
Main
{
    x = 0x10    # Declare a local variable 'x'.
    Local y = 1 # Explicitly declare a local variable 'y'

    Send TX_PACKET
    {
      DevAddr = x       # Use the previously assigned variable 'x'
      Endp    = y + 2   # Use the previously assigned variable 'y' + 2
      SeqNum  = z       # Create a new variable 'z' with value 0.
    }
}
```

8.2 Local Structure Parser Variables

You can declare a local "structure" parser variable, which is seen only inside the generation procedure, and use it in a **Send Packet** instruction and field assignments.

Format

```
[ Local ] $var = template_or_var_name
[ Local ] $var = $variable_name
```

Note: The **Local** keyword is required if you already have a global variable with the same name. Using this keyword explicitly instructs the script parser to declare a local variable with the same name.

Example

```
Main
{
    # Declare a variable X as a packet of type MY_TX_PACKET.
    $X = MY_TX_PACKET
    # Note: Packet variables can be declared/redeclared and
    # used many times.

    # Declare a variable Y and change the default packet.
    $Y = MY_TX_PACKET
    # Template field values
    {
       DevAddr = 0x10
       Endp    = 2
    }

    # Declare a variable Z using the packet variable Y as a prototype.
    $Z = Y
    {
       DevAddr = 0x10
       Endp    = 2
    }

    Local $Z = Y # Explicitly declare a local variable.

    # Declare a structure variable having the same name as one of the
    # templates.
    $MY_TX_PACKET = MY_TX_PACKET
    {
       DevAddr = 0x10
       Endp    = 2
    }

    # Explicitly instruct the script parser to create a new
    # structure variable based on the other structure variable rather
    # than a template.
    $W = $MY_TX_PACKET

    # Send a packet based on the structure variable.
    Send $W
}
```

8.3 Using Local Fields in Structure Variables

You can declare **local fields** within a structure variable. Use local fields to reassign bit offsets or append additional data to a structure variable.

These fields are only valid in the structure variable in which they are declared, not the packet template.

You can declare new structure variables based on the structure variable with local fields. However, it is illegal to assign an existing structure variable to a structure variable that contains declared local fields. This may or may not generate parse errors. If no parse errors are generated, the local fields are not in the assigned structure variable.

Note: See section Changing Structure Parser Variables for more examples.

Example

```
struct S1
{
    F16 : 16 = 0xAABB
    F8  :  8 = 0xFE
    F32 : 32 = 0xABCD1234
}
Main
{
    # Declare a structure variable based on struct S1.
    $Pkt_Var1 = S1
    # Declare local field F24 in structure variable $Pkt_Var1. The
    # local field is from offset 0 and is 24 bits long (it has the same
    # offsets as F16 and F8).
    $Pkt_Var1
    {
        F24 : 0,24 = { 11 22 33 }
    }

    # Declare a structure variable based on struct S1.
    $Pkt_Var2 = S1
    # Declare local field F24 in structure variable $Pkt_Var2. The
    # local field is 120 bits long at offset 56 (end of the structure
    # variable).
    $Pkt_Var2
    {
        F128 : 120 = { 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF }
    }

    # Declare structure variable $Pkt_Var3 (assigned from $Pkt_Var1).
    $Pkt_Var3 = $Pkt_Var1

    # Declare a structure variable based on struct S1.
    $Pkt_Var4 = S1
    # Bad assignment $Pkt_Var2 to $Pkt_Var4 (data NOT copied)
    $Pkt_Var4 = $Pkt_Var2
}
```

8.4 Changing Structure Parser Variables

You can change a structure variable in a generation procedure by changing a field value, length, offset, or hex stream assignment, or you can add to or edit the structure variable by declaring local fields. You can also redeclare the variable.

Note 1: If you change a structure variable by a hex stream assignment and the structure variable has some variable-length fields, then only the first variable-length field is filled with data and all other variable-length fields have zero length.

Note 2: If you declare local fields within a structure variable, they are only valid in that structure variable, not the packet template. New structure variables can be declared based on the structure variable with local fields. However, it is illegal to assign an existing structure variable to a structure variable that contains declared local fields, and this generates parse errors.

Examples

```
Struct MY_STRUCT
{
    V1 : 16
    V2 : 8
    V3 : 8
    V4 : 32
}

Struct MY_STRUCT_2
{
    F1 : 8
    F2 : 32
    F3 : * # Variable length
    F4 : * # Variable length
    F5 : 32
}

Main
{
    # Change some fields in a structure variable.
    $X { DevAddr = 0x12 }

    # Assign some values at some offsets:
    # The format is $PktVar[offset, length] =
    # appropriate field assignment (numeric value, data pattern, etc.)
    $X[16,8] = 0xAA
    $X[16]   = 0xAA




    # Note: If the length value is omitted, the default is 8 bits.
    offset = 16 # Preprocessor variable keeps the offset.
    len    = 8  # Preprocessor variable keeps the length.
    $X[offset, len] = 0xAA
    $X[offset]      = 0xAA
```

```
$Y = MY_STRUCT

# Change the structure variable from a hex stream.
$Y = { 11 22 33 44 55 66 77 88 }

# After the above change, the variable Y fields have the
# following values:
#  Y.V1 = { 11 22 }
#  Y.V2 = 0x33
#  Y.V3 = 0x44
#  Y.V4 = { 55 66 77 88 }

# Change $Y from the above to {AA BB CC DD 55 66 77 88} and
# add local field F6 to variable X.
$Y { V_123 : 0, 32 = { AA BB CC DD }

$Z = MY_STRUCT_2

# Attempt to copy $Y to $Z. This will NOT work and may or may not
# generate a parse error.
# If no error is generated, the parser skipped this instruction.
# (Because $Z is derived from MY_STRUCT_2 and $Y is no longer
# purely MY_STRUCT_2 with its local variables,
# this assignment is not valid).
$Z = $Y

# Copy $Y to a newly instantiated $W structure variable.
# This is legal because $W does not have a packet template
# associated with it, because it is being declared.
$W = $Y

# Change $Y from the above to
# {AA BB CC DD 55 66 77 88 00 00 00 00 FF FF}
# and add local variable V5 at offset 96 with length 16.
# Offset 64-95 is padded with zeroes.
$Y { V5 : 96, 16 = { FF FF } }




# Send a packet of {FF 00 FE FD FC 66 77 88 00 00 00 00 FF FF} by
# modifying $Y from the above and
# instantiate local variables V3_1 and V3_2 for use
# in this send instruction only. [ No $ = send ]
Y
{
  V1 = 255
  V2 = 254
  V3_1 : 16, 8 = 253
  V3_2 : 24, 8 = 252
}
```

```
# COMPLEX LOCAL FIELD MANIPULATION EXAMPLES

$X = MY_STRUCT_2
# Change the structure variable from a hex stream.
$X = { 11 AA BB CC DD 22 33 44 BE EF BE EF }

# After the above change, the variable X fields have the
# following values:
#  X.F1 = 0x11
#  X.F2 = { AA BB CC DD }
#  X.F3 = { 22 33 44 }
#  X.F4 = empty
#  X.F5 = { BE EF BE EF }

# Change $X from the above to
# {11 AA BB CC DD 22 33 44 BE F0 0D F0 0D}
# and add local field F6 to variable X.
# The variable length fields F3 & F4 are treated as empty
# and F5 is treated as having offset 40-71,
# hence F6 occupies offset 72-103.
$X { F6 : 32 = { F0 0D F0 0D } }

# Change $X from the above to
# {11 02 01 04 03 22 33 44 BE F0 0D F0 0D}
# and add local fields F2Lower and F2Upper to variable X.
$X{
     F2Lower :  8, 16 = { 02 01 }
     F2Upper : 24, 16 = { 04 03 }
  }

# Change $X from the above to
# {C0 01 C0 DE 11 22 AA EE BE F0 0D F0 0D}
# and add local field F1_F2 and B3 to variable X.
# B3 is only 16 bits long, hence only AA EE is taken as data,
# and FF BB is truncated.
$X
{
  F1_F2 : 0, 40  = { C0 01 C0 DE 11 }
  B3    : 48, 16 = { AA EE FF BB } #Note: B3 is only 16 bits long.
}

# Send a packet of {C0 01 C0 DE 11 22 AA EE 44 55 22 33 44} by
# modifying $X from the above.
# Existing local field F6 is used and
# F7 is instantiated for use in this Send instruction only.
# F6 fills in from offset 72-103,
# because that is where it became declared
```

```
    # when created in $X earlier.
    # F7 fills in offset 64-79,
    # because variable length fields F3 and F4 still exist,
    # and there is a hole between offset 63-72.
    Send $X{
              F6 = { 11 22 33 44 }
              F7 : 16 = { 44 55 }
           }
}
```

8.5 Using Special Data Pattern Creators in Field Assignments

You can use special pattern constructors to simplify creation of data patterns.

Examples

```
Main
{
    # Examples of using special data pattern declarations while
    # assigning field values.
    Send TX_PACKET
    {
        Endp = 0xEFBE
        Data = { 01 02 03 00 0A }  # Simple byte stream
        Data = PATTERN_2           # Use declared data pattern.

        # Use combined byte stream.
        Data = { AA BB CC _11110000 12 34 56 78 [PATTERN_2] }

        Data = 12, 0xAA # Specify data payload of 12 bytes and
                        # fill it by 0xAA: AA AA AA AA ... AA

        # Specify data payload of 12 bytes and fill it starting
        # from 1, incrementing each byte by 2: 01 03 05 07 09 ...
        Data = 12, 1, 2
    }
}
```

8.6 Using Structure Variables to Assign Field Values

You can use structure variables in field assignments.

Examples

```
Main
{
    # Use structure variables to assign field values.
    # Declare a 'structure' instance based on the template S1.
    $S = S1

    # Declare a 'structure' instance based on the template S2
    # and change the default value for the S16 field.
    $W = S2 { S16 = 2 }

    Send TX_PACKET
    {
        Data = $S  # Data field contains the payload of structure S.
    }

    Send TX_PACKET
    {
        # Example of concatenation of structures.
        # Data field contains combined payload of structures S and W
        Data = $S + $W
    }
}
```

8.7 Using Multipliers to Assign Field Values

You can use a "multiplier" to assign repeated data and create complex assignments. This multiplier only works on structure variables when assigning fields.

Note: Though this multiplier uses the symbol *, this multiplier is not the arithmetic multiplier.

Examples

```
Generic $X
Generic $Y
Main
{
    # Use multipliers to assign field values.
    # Declare a packet variable of 'structure' instance Generic with
    # data FB 01 02.
    $TestIE = Generic { Data = { FB 01 02 } }

    # Declare a 'structure' instance based on the template S2
    # and change the default value for the S16 field.
    $S2_Var = S2 { S16 = 2 }

    Send TX_PACKET
    {
        Data = 10*$TestIE # Data field contains the sequence of $TestIE
                          # repeated 10 times:
                          # { FB 01 02 FB 01 02 FB 01 02 FB 01 02 FB 01
                          #   02 FB 01 02 FB 01 02 FB 01 02 FB 01 02 }
    }

    # Declare Packet Variable $X of structure Generic with data FF.
    $X = Generic { Data = 0xFF }

    # Packet Variable $Y contains the sequence of $TestIE repeated
    # three times and $X repeated two times:
    # { FB 01 02 FB 01 02 FB 01 02 FF FF }
    $Y { Data = 3*$TestIE + 2*$X }
    Num_of_Y = 5          # Local Numeric Variable
    Send TX_PACKET
    {
        # Example of concatenation and multiplication of structures.
        # Data field contains combined payload of:
        # $S2_Var + $S2_Var + $X + $X + $X + $X + $X + $X + $X + $X + $X + $X
        # + $X + $X + $X + $X + $Y + $Y + $Y + $Y + $Y
        Data = $S2_Var*2 + 16*$X + Num_of_Y*$Y
    }
}
```

8.8 Using the Append Operator in Field Assignments

You can append data to fields using **= +** (append operator) in field assignments. Use the append operator to generate dynamic packets and avoid restating previously assigned values. For example, if a packet variable field **SomeField** contains “01 02 03”, to add “04 05 06” you can assign SomeField = +{04 05 06}), rather than assigning SomeField = {01 02 03 04 05 06}.

Note: Do not confuse this operator with the += operator, or a parse error will be generated! You can write the append operator as **= +** or **=+** (without a space), but it is recommended to use the space to avoid confusion with the **+=** operator.

Examples

```
Packet BPOIE
{
    ElemID           : 8 = 1
    ElemLen          : 8
    BPLen            : 8
    Occupancy_Bitmap : *
    DevAddrList      : *
}
BPOIE $Beacon3_BPOIE
BPOIE $Beacon11_BPOIE

Main
{
    $Beacon3_BPOIE
    {
        BPLen = 16
        Occupancy_Bitmap = { 00 55 55 55 01 }
        DevAddrList = { 04 00 }
    }
    # Generate DevAddrList for Addrs 5-16 : DevAddrList = {04 00} -->
    # DevAddrList = { 04 00 05 00 06 00 07 00 08 00 09 00 0A 00
    #                 0B 00 0C 00 0D 00 0E 00 0F 00 10 00 }
    for( i=5, i<17, i++)
    {
        $Beacon3_BPOIE{ DevAddrList = +{ i 00 } }
    }
    $Beacon11_BPOIE
    {
        BPLen = 16
        Occupancy_Bitmap = { 10 55 15 55 01 }
        DevAddrList = { 03 00 }
    }
    # Generate DevAddrList for Addrs 4-10 and 12-16 :
    # DevAddrList = {03 00} -->
    # DevAddrList = { 03 00 04 00 05 00 06 00 07 00 08 00 09 00
    #                 0A 00 0C 00 0D 00 0E 00 0F 00 10 00 }
    for(i=4, i<17, i++)
    {
        if (i!=11)
        {
            $Beacon11_BPOIE{ DevAddrList = +{ i 00 } }
        }
    }
}
```

8.9 Initializing Struct Variables from Hex Streams

You can assign structure variables using hex streams. Rather than defining each field within a structure variable, set the structure variable equal to a hex/byte stream. The fields within the structure variable are then assigned based on the offset.

8.9.1 Assignments for Variables with Fixed-length Fields

For structure variables composed of fixed length fields (so that the structure variable has fixed length), assignments by hex streams fill the fields in offset order. Any overflow from the hex stream is truncated. For example, assigning an eight-byte structure variable to a twelve-byte hex stream loses the last four bytes of the hex stream.

Example

```
Packet GenericSample
{
    Field_1 : 8
    Field_2 : 32
    Field_3 : 16
    Field_4 : 4
    Field_5 : 3
    Field_6 : 1
}

GenericSample $Struct_Var

Main
{
    $Struct_Var = { 01 02 03 04 05 06 07 08 }

    # The following is the same as the line above.
    $Struct_Var{
            Field_1 = 1
            Field_2 = { 02 03 04 05 }
            Field_3 = { 06 07 }
            Field_4 = 0
            Field_5 = 4
            Field_6 = 0
    }
}
```

8.9.2 Assignments for Variables with Variable Length Fields

For structure variables that have one variable length field:

1. The hex stream fills all fixed-length fields prior to the first variable-length field, using a top-down approach.
2. The hex stream fills all fixed-length fields following the variable-length field using a bottom-up approach, in which the required number of bytes comes from the end of the hex stream and the fixed-length fields are filled in order.
3. The hex stream fills the variable-length field with the remaining bytes from the middle of the hex stream.

If a structure variable has more than one variable-length field (including interleaved variable-length and fixed-length fields), only the first variable-length field receives data. Later variable-length fields receive no data and are blank. For these types of structure variables, it is not recommended to explicitly modify fields after assignment from a hex stream, because data is not logically assigned. Essentially, a structure variable with more than one variable-length field becomes a structure variable with only one variable-length field, the first one listed in offset order.

Example

```
Packet SF_Beacon
{
    PHY : 40
    MAC : 80
    BH  : 64
    IEs : *
    DATA: *
}

Packet GenericSample
{
    First3_Bytes    : 24
    More_Bytes      : *
    Some_More_Len   : 8
    Some_More_Bytes : *
    Even_More_Len   : 8
    Even_More_Bytes : *
    Total_Len       : 16
}

SF_Beacon $BeaconSlot2
GenericSample $X

Main
{
    $BeaconSlot2 = { 00 0C 00 D0 00 00 00 FF FF 02 00 00 00 00 00
                     98 76 54 32 10 FC 02 00 01 02 03 00 }

    # Equivalent to
    $BeaconSlot2{
        PHY = { 00 0C 00 D0 00 }
        MAC = { 00 00 FF FF 02 00 00 00 00 00 }
        BH  = { 98 76 54 32 10 FC 02 00 }
        IEs = { 01 02 03 00 }
        # DATA is Empty.
    }

    $X = { 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 }

    # Equivalent to
    $X{
        First3_Bytes    = { 01 02 03 }
        More_Bytes      = { 04 05 06 07 08 09 0A 0B 0C 0D 0E }
        Some_More_Len   = 0x15
        # Some_More_Bytes is empty.
        Even_More_Len   = { 10 }
        # Even_More_Bytes is empty.
        Total_Len       = { 11 12 }
    }
}
```

8.10 Sizeof Operators

Several kinds of **sizeof** operators are currently supported:

- **fld_size (field_name):** Returns the length of the field in bits.
- **pkt_size (template)**: Returns the length of the template payload in bits.
- **pkt_size ($pkt_var_name)**: Returns the length of the structure variable payload in bits.
- **pttn_size (data_pattern):** Returns the length of the data pattern payload in bits.

Note: Fields that are not initialized with a variable length (declared as **f : ***) have 0 length.

Examples

```
Main
{
    # Examples of using 'sizeof' operators.
    # Declare a 'structure' instance based on the template S1.
    $S = S1

    # Declare a 'structure' instance based on the template S2,
    # changing the default value for the S16 field.
    $W = S2 { S16 = 2 }

    Send TX_PACKET
    {
       # Set the Len to the size of
       # combined payload + the size of the 'Len' field.
       Len  = ( fld_size( Len ) + pkt_size( $S ) + pkt_size( $W ) ) / 8

       # Same as the previous assignment.
       Len  = ( fld_size( Len ) + pkt_size( S1 ) + pkt_size( S2 ) ) / 8

       # Data field contains combined payload of structures S and W.
       Data = $S + $W
    }
}
```

8.11 Preprocessor Integer Arithmetic

You can declare a preprocessor DWORD variable, make arithmetic operations with it, and use it in field and setting/parameter assignments.

Note: Arithmetic expressions are allowed only in numeric variable assignments. Some legal expressions are.

- x = y + 2
- TX_PACKET (Delay = (x+y)*7)
- TX_PACKET (x + 12)

Examples

```
Const TX_PAYLOAD_OFFSET = 16 * 8
Const "Some Hex Data" = 0xAABB

Main
{
    x = 2
    y = ( z = 12 ) + ( TX_PAYLOAD_OFFSET + 36 ) / 8
    z = 0x1 << 5
    s = "Some Hex Data" # Constants may be used in operations.
    x++
    y--
    z += ( x + y )
    x = ( ( y & 0xFF ) >> 5 ) / 12

    Send TX_PACKET ( Delay = x )

    Send TX_PACKET
    {
        Endp =  0xEEEE

        # Example of a data payload assignment that uses
        # integer variables, constants, hex literals, binary literals,
        # and a data pattern.
        Data = { y y y 7a7a7a "Some Hex Data"
                 "Some Hex Data" 8b 8b _11110000 _00001111 z z z
                  [PATTERN_3] }
    }

    ( Delay = y ) # Use an integer variable for a parameter setting.
}
```

8.12 Preprocessor If Operator

A special preprocessor If directive includes/excludes different parts of generation code depending on some condition.

Examples

```
Main
{
    x = 180
    y = 30
    # Short if operator with only a 'then' clause
    if( x > y )
    {
    # This block is parsed the usual way and all instructions
    # are added to the current instruction block.
        Send TX_PACKET
        Send TX_PACKET ( Delay = 2000 )
    }
    if( x < y )
    {
    # This block is parsed in a special way:
    # Only the "{" "}" are taken into account.
    # All other stuff is ignored.
        Send DATA_PACKET
        {
            DevAddr  = 0x02
            Endp     = 0
            Data      = { AA BB CC DD 12 34 56 78 [PATTERN_2] }
        }
    ( Delay = 1000 )
    }
    # Full if operator with both 'then' and 'else' clauses.
    if( x > y )
    {
    # This block is parsed the usual way and all instructions
    # are added to the current instruction block.
        Send MY_USB3_PKT
        Send MY_DATA_PACKET ( Delay = 2000 )
    }
    else
    {
    # This block is parsed in a special way.
    # Only "{" "}" are taken into account.
    # All other stuff is ignored.
        Send MY_USB3_PKT
    }
}
```

8.13 Preprocessor Loop Operators

Note: Loop operators can produce a huge number of instructions. Therefore, you should set a maximum allowed limit for the total number of loop iterations in a generation script. To do this, use the **MaxLoopIterCount** parser setting.

Examples

```
Main
{
    k = 0
    # 'while' loop operator.
    while( k < 4 )
    {
        k++
        # Skip the remaining part of the loop iteration.
        if( k == 2 ) { skip_iteration }
        # Stop the loop.
        if( k == 3 ) { stop_loop }

        Send TX_PACKET
        {
            Endp   = 2
            DevAddr = 0x10
            # Data pattern uses preprocessor variables.
            Data     = { k k k k k }
        }
    }

    # 'for' loop operator.
    for( k = 0, k < 10, k++ )
    {
        # Skip the remaining part of the loop iteration.
        if( k == 2 ) { skip_iteration }
        # Stop the loop.
        if( k == 4 ) { stop_loop }

        Send TX_PACKET
        {
            Endp   = 0
            DevAddr = 0xB
            # Data pattern uses preprocessor variables.
            Data     = { k k k k k }
       }
    }
}
```

8.14 Forward Declarations

You can use declared items in generation procedures before their declarations.

Example

```
Main
{
    # The packet template 'SOME_PKT_TEMPLATE' is declared later.
    Send SOME_PKT_TEMPLATE
    {
        Field0 = 0xAABBCCDD
    }

    Call Block1() # The procedure 'Block1' is declared later.

    # Declare the 'SOME_PKT_TEMPLATE' packet template used above.
    Packet SOME_PKT_TEMPLATE
    {
        Field0 : 32 = 0xFEFEFEFE
    }
}

# Declare the procedure 'Block1' used above.
Block1
{
    Send TX_PACKET
}
```

8.15 RAND Token

You can use a **RAND** token in places where numeric literals are used to insert pseudo-random numbers in the range 0 to 0x7fff.

Note: By default, **RAND** uses a different integer seed value every time the script is compiled. You can set the seed using the **RandSeed** setting. Ffor the **RandSeed** setting description, see the Generation Setting topic.

Example

```
Main
{
    # The packet template 'SOME_PKT_TEMPLATE' is declared later.
    Send SOME_PKT_TEMPLATE
    {
        Field0 = { 00 RAND RAND RAND RAND 00 } # Set random hex
                                               # stream.
    }

    x = RAND # Assign a random value to the numeric variable.
}
```

8.16 RandStream(n) Primitive

The **RandStream(n)** primitive is a utility, based on the **RAND** token, that produces a random byte stream, where **n** is the number of bytes in the stream.

Note: For a description of random seeding, see the Note in the Rand Token section.

Example

```
Main
{
    # The packet template 'SOME_PKT_TEMPLATE' is declared later.
    Send SOME_PKT_TEMPLATE
    {
        Fieldof32bytes = RandStream(32) # Set a random 32-byte stream.
    }
}
```

8.17 Global Numeric and Structure Variables

You can declare global numeric and structure variables that can be used in different generation procedures during parsing. Such global variables can be changed in one generation procedure, and then the changes are used in other generation procedures.

Example

```
Const MY_CONST = 77
x = 0xAA      # Declare global numeric variable 'x'.
y = 0x12      # Declare global numeric variable 'y'.
z = MY_CONST  # Declare global numeric variable 'z'.

# Declare a global structure variable 'Z' based on the template
# 'TX_PACKET'.
TX_PACKET $Z

# Declare a global structure variable 'Y' based on the template
# 'TX_PACKET'and change the default template field values.
TX_PACKET $Y
{
    DevAddr = 0xAABB
}

# Another way to declare global structure variables:
# Declare the global structure variable 'Z' based on the template
#'TX_PACKET'.
$Z = TX_PACKET

# Declare the global structure variable 'Y' based on the template
# 'TX_PACKET' and change the default template field values.
$Y = TX_PACKET
{
    DevAddr = 0xAABB
}

Main
{
    Send $Z  # Send a packet based on the global structure variable Z.

    Send SOME_PKT_TEMPLATE
    {
        Field0 = y # Use the global numeric variable 'y' to change the
                   # default template field values.
    }
}
```

For more examples of using global variables, see the examples for the **CALL** directive in the next section.

8.18 Using the Call Directive for Generation Procedure Insertions

You can declare a generation procedure with parameters and then "call" the generation procedure using the **Call** directive. The **Call** command inserts the other procedure instructions and takes into account the passed parameters or global variables at every call. This is called "dynamic" insertion, as opposed to the "static" insertion implemented by the **insert** directive.

Note: The **Call** directive copies instructions from the generation procedure. The **Insert** directive does not copy instructions from the generation procedure. Rather, the generation parser inserts a special **Insert** instruction with a reference to the "inserted" generation procedure.

8.18.1 Calling a Generation Procedure with Parameters

Using the **Call** directive, the parser can insert instructions from another generation procedure and, before insertion, provide some values for local parameters for use during parsing of that procedure.

Example 1

```
# Declare a packet template.
Packet SOME_PKT_TEMPLATE
{
    Field0 : 32 = 0xFEFEFEFE
}

# Declare a generation procedure with two numeric parameters.
Block1( x, y )
{
# Send a packet based on the template 'SOME_PKT_TEMPLATE'.
    Send SOME_PKT_TEMPLATE
    {
        # Procedure parameters 'x' and 'y' are used to override
        # the default template field values.
        Field0 = x + y
    }
}
# The generation procedure now has one structure variable parameter and
# two numeric parameters.
Block2( SOME_PKT_TEMPLATE $pkt_param, x, y )
{
    # Send a packet based on the structure variable parameter
    # to 'pkt_param'.
    Send pkt_param
    {
        Field0 =  x + y
    }
}

# Call another generation procedure.
Main
{
    Send TX_PACKET # Send packet based on the template 'TX_PACKET'.

    $X = SOME_PKT_TEMPLATE # Declare a variable X as a packet of type
                           # SOME_PKT_TEMPLATE.
    w = 1
    u = 0xAABB
    # "Call" the procedure 'Block2' with parameters.
    # Note: The actual structure variable parameter
```

```
    # must have '$' before the name of a variable.
    Call Block2 ( $X, w, u )

    $X # Change the local structure variable 'X'.
    {
         Field0 = 5096
    }

    w = 1024 # Change the local numeric variable 'w'.
    u = 12   # Change the local numeric variable 'u'.

    Call Block2 ( $X, w, u ) # Call the procedure 'Block2' with new
                             # parameters.
# Note: The parser checks the variable structure parameters when
# processing the Call directive and allows only structure variables
# derived from the templates specified in the procedure declaration.

    $X = DIFFERENT_TEMPLATE # Redeclare a structure variable using a
                    # template not derived from 'SOME_PKT_TEMPLATE'.

    Call Block2 ( $X, w, w ) # !!! yields a parsing error.
}
```

Example 2

```
struct Generic
{
    Data : *
}

Packet SOME_PACKET
{
    # Header
    HdrField0 : 16
    HdrField1 : 16
    HdrField2 : 32

    # Payload
    Data      : *
    CRC32     : 32
}

Block( Generic $X, Generic $Y )
{
    Send SOME_PACKET
    {
        HdrField0 = 0xAAAA
        HdrField1 = 0xBBBB
        HdrField2 = 0xABCDBEEF
        Data      = $X + $Y  # Combined payload
    }
}
Main
{
    $X = Generic
    {
        Data = { AA BB CC DD }
    }

    $Y = Generic
    {
        Data = { 00 11 22 33 }
    }

    Call Block( $X, $Y ) # Send a packet based on 'SOME_PACKET'
                         # template and Data = { AA BB CC DD 00 11 22 33 }

    $X # Change the structure variable 'X'.
    {
        Data = { BE EF DA CD }
    }

    $Y # Change the structure variable 'Y'.
    {
        Data = { BE EF DA CD }
    }
```

```
        Call Block( $X, $Y ) # Send a packet based on 'SOME_PACKET'
                             # template and Data = { BE EF DA CD BE EF DA CD }
    }
```

8.18.2 Calling a Generation Procedure with No Parameters

You can "call" a generation procedure and omit some of its parameters, because procedure parameters have default values.

Example

```
Block2( SOME_PKT_TEMPLATE $pkt_param, x, y )
{
# Send a packet based on the structure variable parameter 'pkt_param'.
    Send pkt_param
    {
        Field0 =  x + y
    }
}

Generation
{
    # Call the procedure 'Block2' with no parameters.
    # In this case, use the default values for procedure parameters.
    # Note: '(' ')' is mandatory after the procedure's name.
    Call Block2()
}
```

8.18.3 Nested Calls Using Global Variables

You can call another generation procedure inside a generation procedure. This is called "nested calls". During such calls, the current values of global variables are used when the called generation procedure is parsed.

Example

```
z = 18
SOME_PKT_TEMPLATE $Z # Declare a global structure variable 'Z'
                     # based on the template 'SOME_PKT_TEMPLATE'.
Main
{
    $Y = SOME_PKT_TEMPLATE # Declare a local structure variable 'Y'
    {
        Field0 = z + 12    # Use a global variable 'z'.
    }
    z++     # Increment the global variable 'z'.
    Send Y  # Send packet based on the local structure variable 'Y'.
    Call Block1( $Y, 12, 10 )  # Call 'Block1'.
}

Block1( SOME_PKT_TEMPLATE $p, x, y )
{
    $p # Change the local structure variable 'p'.
    {
       Field0 = 2*(x + y) + z # Use local variables 'x', 'y' and global
                              # variable 'z' here.
    }

    Send Z # Send packet based on the global variable 'Z'.
    $Z     # Change the global structure variable 'Z'.
    {
        Field0 = 0x0000BEEF
    }
    $Z[0, 4] = 0xA     # Change the 4 bits of structure variable 'Z'
                       # at offset 0 bits.
    Call Block2( $p )  # Call 'Block2'.
}

Block2( SOME_PKT_TEMPLATE $p )
{
    Send p  # Send packet based on the local structure variable 'p'.
    Send Z  # Send packet based on the global structure variable 'Z'.
}
```

8.19 Parser Tracing Functions

Parser tracing functions are debug tools that allow you to follow the generation process. These functions track script generation procedures and allow you to observe instruction generation and the changes of values of variables and template structures. Because the Voyager USB 3.1 Exerciser can create sophisticated and dynamic generation scripts, these functions are helpful in debugging.

Note: These functions have no affect on the compiled firmware instruction output. They only follow the paths taken in scripts to create instructions or values of variables and template structures.

8.19.1 PTrace( ) : Parser Trace

Format

PTrace( )

Parser Trace without any supplied parameters outputs the line number from which it was called to the compilation output window.

PTrace(“output_string”)

Parser Trace with a supplied string parameter outputs the supplied string to the compilation output window when called.

Example

```
50: Main
51: {
52:     PTrace ()
53:     Send TX_PACKET
54:     {
55:         Data = {AA BB CC DD}
56:     }
57:     PTrace (“Sending Second TX_PACKET”)
58:     Send TX_PACKET
59:     {
60:         Data = {EE FF 00 11}
61:     }
62:     PTrace ()
63: }
```

Output Window:

```
-- Parsing started: File: C:\Sample_PTrace.usb3g. Operation Starts at time 08:44:07 PM --
Parsing...
c:\sample_ptrace_1.usb3g
***Parser Trace Message(c:\sample_ptrace_1.usb3g, line: 52)    ← PTrace( ) from line 52
Sending Second TX_PACKET                                       ← PTrace( … ) from line 57
***Parser Trace Message(c:\sample_ptrace_1.usb3g, line: 62)    ← PTrace( ) from line 62
Parsing is done...
C:\Sample_PTrace_1.usb3g - 0 error(s)
C:\Sample_PTrace_1.usb3g - Generation Blocks : 1, Total number of instructions : 2
-- Parsing Finished: File: C:\Sample_PTrace.usb3g. Operation ends at time 08:44:08 PM --
```

8.19.2 PTraceVar( ) : Parser Trace Variable

Format

PTraceVar(<variable list *var1*, *var2*, …>)

Parser Trace Variable outputs the values of the supplied comma-separated variables. Numeric variables are output in decimal and hex format. Packet variables are output in byte streams (hex). This primitive does not accept constants or data patterns, which generate parse errors.

Note: Numeric variables are DWORDS.

Example

```
51: packet GenericData
52: {
53:     DATA : *
54: }
55: GenericData $XYZ { DATA = { 00 11 00 11 00 11 00 11 } }
56: global_var = 7
57: Main
58: {
59:     local_var = 12
60:     PTraceVar ( global_var, local_var, $XYZ )
61:     Send $XYZ #Send $XYZ with the following data for the instance.
62:     {
63:         Data = {AA BB CC DD}
64:     }
65:     PTraceVar ( $XYZ )
66:     # Modify the value of $XYZ.
67:     $XYZ { DATA = { EE FF 00 11 global_var local_var } }
68:     local_var = 777777
69:     PTraceVar ( $XYZ, local_var )
70:     Send $XYZ # Send $XYZ.
71:     PTraceVar ( $XYZ )
72: }
```

Output Window:

```
- Parsing started: File: C:\Sample_PTraceVar.usb3g. Operation Starts at time 09:29:17 AM-
Parsing...  c:\sample_ptracevar.usb3g
GLOBAL_VAR = 7 (hex: 0x7)                    ← PTraceVar( global_var, … ) from line 60
LOCAL_VAR = 12 (hex: 0xC)                    ← PTraceVar( …, local_var, … ) from line 60
$XYZ = {00 11 00 11 00 11 00 11} - 8 byte(s) ← PTraceVar( …, $XYZ ) from line 60
$XYZ = {00 11 00 11 00 11 00 11} - 8 byte(s) ← PTraceVar( $XYZ ) from line 67
$XYZ = {EE FF 00 11 07 0C} - 6 byte(s)       ← PTraceVar( $XYZ, … ) from line 69
LOCAL_VAR = 777777 (hex: 0xBDE31)            ← PTraceVar( …, local_var ) from line 69
$XYZ = {EE FF 00 11 07 0C} - 6 byte(s)       ← PTraceVar( $XYZ ) from line 71
Parsing is done... C:\Sample_PTraceVar.usb3g - 0 error(s)
C:\Sample_PTraceVar.usb3g - Generation Blocks : 1, Total number of instructions : 2
- Parsing Finished: File: C:\Sample_PTraceVar.usb3g. Operation ends at time 09:29:17 AM -
```

8.19.3 PTraceVarEx( ) : Parser Trace Variable Extended

Format

PTraceVarEx(<variable list *var1*, *var2*, …>)

Parser Trace Variable Extended outputs the values of the supplied comma-separated variables. **PTraceVarEx( )** is the same as **PTraceVar( )**, except that packet variable values are output in detail at the field level, rather than just as a byte stream. Numeric variables are output in decimal and hex format. This primitive does not accept constants or data patterns, which generate parse errors. In the example below, also note that the language is not case sensitive.

Note: Numeric variables are DWORDS.

Example

```
51: struct IE_Info                      # Create Struct IE_Info.
52: {
53:     ElemID  : 8
54:     ElemLen : 8
55: }
56: Packet Test_IE_ChSelect : IE_INFO # Create Packet Test_IE_ChSelect.
57: {
58:     ElemID  = 251
59:     ElemLen = 2
60:     TModeSubType : 8 = 1
61:     PHYChNum     : 8
62: }
63: Packet BPOIE : IE_INFO             # Create Packet BPOIE.
64: {
65:     ElemID = 1
66:     BPLen       : 8
67:     Occupancy   : *
68:     DevAddrList : *
69: }
70: Test_IE_ChSelect $Pkt_Var { PHYCHNUM = 5 } # Global Packet Variable
71: BPOIE $Pkt_Var2                           # Global Packet Variable
72: {
73:     BPLen       = 12
74:     Occupancy   = { 00 01 00 }
75:     DevAddrList = 0x0004
76:     ElemLen     = 6
77: }
78: alternateCHNum = 7                       # Global Numeric Variable
79: Main
80: {
81:     PTraceVarEx ( $Pkt_Var, $Pkt_Var2 )
82:     Send $Pkt_Var
83:     $Pkt_Var { phychnum = alternatechnum } # Modify $Pkt_Var.
84:     PTraceVarEx ( AlternateCHNum, $Pkt_Var )
85:     Send $Pkt_Var
86:     Send $Pkt_Var2
87:   $Pkt_Var2 = { 01 08 0C 00 05 00 04 00 05 00 } # Modify $Pkt_Var2.
88:     pkt_var2 = 1   # Declare Local Variable pkt_var2 (note no '$').
89:     PTraceVarEx ( $Pkt_Var2, Pkt_Var2 )
90:     if(pkt_var2 != 0) { Send $Pkt_Var2 }
91: }
```

Output Window:

```
Parsing started: File: C:\Sample_PTraceVarEx.usb3g. Operation Starts at time 05:50:10 PM
Parsing...  c:\sample_ptracevarex.usb3g
$PKT_VAR = {FB 02 01 05} - 4 byte(s)        ← PTraceVarEx( $Pkt_Var, … ) from line 81
Template name: TEST_IE_CHSELECT               ←
Fields:                                       ←
Field : ELEMID                                ←
```

```
    index = 0, offset = 0, length = 8          ←
Value : FB                                     ←

Field : ELEMLEN                                ←
    index = 1, offset = 8, length = 8          ←
Value : 02                                     ←

Field : TMODESUBTYPE                           ←
    index = 2, offset = 16, length = 8         ←
Value : 01                                     ←

Field : PHYCHNUM                               ←
    index = 3, offset = 24, length = 8         ←
Value : 05                                     ←

$PKT_VAR2 = {01 06 0C 00 01 00 04} - 7 byte(s)← PTraceVarEx( …, $Pkt_Var2 ) from line 81
Template name: BPOIE                           ←
Fields:                                        ←
Field : ELEMID                                 ←
    index = 0, offset = 0, length = 8          ←
Value : 01                                     ←

Field : ELEMLEN                                ←
    index = 1, offset = 8, length = 8          ←
Value : 06                                     ←

Field : BPLEN                                  ←
    index = 2, offset = 16, length = 8         ←
Value : 0C                                     ←

Field : OCCUPANCY                              ←
    index = 3, offset = 24, length = 24 (Variable length)   ←
Value : 00 01 00                               ←

Field : DEVADDRLIST                            ←
    index = 4, offset = 48, length = 8 (Variable length)    ←
Value : 04                                     ←
ALTERNATECHNUM = 7 (hex: 0x7)          ← PTraceVarEx( alternatechnum, … ) from line 84
$PKT_VAR = {FB 02 01 07} - 4 byte(s)   ← PTraceVarEx( …, $Pkt_Var2 ) from line 84
Template name: TEST_IE_CHSELECT                ←
Fields:                                        ←
Field : ELEMID                                 ←
    index = 0, offset = 0, length = 8          ←
Value : FB                                     ←

Field : ELEMLEN                                ←
    index = 1, offset = 8, length = 8          ←
Value : 02                                     ←

Field : TMODESUBTYPE                           ←
    index = 2, offset = 16, length = 8         ←
Value : 01                                     ←

Field : PHYCHNUM                               ←
    index = 3, offset = 24, length = 8         ←
Value : 07                                     ←

$PKT_VAR2 = {01 08 0C 00 05 00 04 00 05 00} - 10 byte(s) ← PTraceVarEx($Pkt_Var2, …)
                                                          from line 89
Template name: BPOIE                                      ←
Fields:                                                   ←
Field : ELEMID                                            ←
    index = 0, offset = 0, length = 8                     ←
Value : 01                                                ←

Field : ELEMLEN                                           ←
    index = 1, offset = 8, length = 8                     ←
Value : 08                                                ←

Field : BPLEN                                             ←
    index = 2, offset = 16, length = 8                    ←
Value : 0C                                                ←

Field : OCCUPANCY                                         ←
    index = 3, offset = 24, length = 56 (Variable length) ←
Value : 00 05 00 04 00 05 00                              ←
```

```
Field : DEVADDRLIST                                          ←
    index = 4, offset = 80, length = 0 (Variable length)    ←
Value : FE                                                   ←

PKT_VAR2 = 1 (hex: 0x1)                    ← PTraceVarEx( …, pkt_var2 ) from line 89
Parsing is done...  C:\Sample_PTraceVarEx.usb3g - 0 error(s)
C:\Sample_PTraceVarEx.usb3g - Generation Blocks : 1, Total number of instructions : 4
-Parsing Finished: File: C:\Sample_PTraceVarEx.usb3g. Operation ends at time 05:50:10 PM
```

8.19.4 PtraceTemplate( ) : Parser Trace Template

Format

PTraceTemplate(<template list *Template_1*, *Template_2*, …>)

Parser Trace Template displays the layout of the supplied comma-separated templates. Use it to verify that template declaration was written as intended or to look at what the template type contains when the template has **include** statements. This primitive does not accept parameters other than templates, such as struct, packet, and packet names, which generate parse errors.

Example

```
const DPH = 0x8

Packet USB3Common
{
    Type : 5
}

struct LinkCtrlWord
{
    Hseq    : 3
    LcwRsvd : 3
    HDepth  : 3
    D1      : 1
    D2      : 1
    CRC_5   : 5 # Auto calculated
}

Packet DataPacket : USB3Common
{
      #Dword 0
      Type = DPH
      RouteStr : 20
      DevAddr  : 7

      #Dword 1
      SeqNum   : 5
      Rsvd1    : 1
      Delayed  : 1
      EOB      : 1
      Endp     : 4
      Rsvd2    : 3
      Setup    : 1
      DataLen  : 16

      #Dword 2
      StreamID : 16
      Rsvd3    : 11
      PP       :  1
      Rsvd4    :  4
```

```
        #Dword 3
        CRC_16    : 16 # Auto calculated
        : LinkCtrlWord # structure template insertion
        Data      : *
        CRC32     : 32
}

Main
{
        PTraceTemplate ( DataPacket, LinkCtrlWord )
        Send DataPacket { DATA = { 01 08 0C 00 05 00 04 00 05 00 } }
}
```

Output Window:

```
Parsing C:\Usb3 Gen Scripts\Usb3GenTest1.usb3g ...
--------------------------------------------------
Template : DATAPACKET
Template group : Generic
Top Ancestor Name : USB3COMMON

Fields:
Field : TYPE
    index = 0, offset = 0, length = 5
Value : 08

Field : ROUTESTR
    index = 1, offset = 5, length = 20
Field : DEVADDR
    index = 2, offset = 25, length = 7
Field : SEQNUM
    index = 3, offset = 32, length = 5
Field : RSVD1
    index = 4, offset = 37, length = 1
Field : DELAYED
    index = 5, offset = 38, length = 1
Field : EOB
    index = 6, offset = 39, length = 1
Field : ENDP
    index = 7, offset = 40, length = 4
Field : RSVD2
    index = 8, offset = 44, length = 3
Field : SETUP
    index = 9, offset = 47, length = 1
Field : DATALEN
    index = 10, offset = 48, length = 16
Field : STREAMID
    index = 11, offset = 64, length = 16
Field : RSVD3
    index = 12, offset = 80, length = 11
Field : PP
    index = 13, offset = 91, length = 1
Field : RSVD4
    index = 14, offset = 92, length = 4
Field : CRC_16
    index = 15, offset = 96, length = 16

Field : HSEQ
    index = 16, offset = 112, length = 3
Field : LCWRSVD
    index = 17, offset = 115, length = 3
Field : HDEPTH
    index = 18, offset = 118, length = 3
Field : D1
```

```
    index = 19, offset = 121, length = 1
Field : D2
    index = 20, offset = 122, length = 1
Field : CRC_5
    index = 21, offset = 123, length = 5
Field : DATA
    index = 22, offset = 128, length = 0(Variable length)
Field : CRC32
    index = 23, offset = 128, length = 32
------------------------------------------------------------

------------------------------------------------------------
Template : LINKCTRLWORD
Template group : Generic

Fields:
Field : HSEQ
    index = 0, offset = 0, length = 3
Field : LCWRSVD
    index = 1, offset = 3, length = 3
Field : HDEPTH
    index = 2, offset = 6, length = 3
Field : D1
    index = 3, offset = 9, length = 1
Field : D2
    index = 4, offset = 10, length = 1
Field : CRC_5
    index = 5, offset = 11, length = 5
------------------------------------------------------------


C:\Usb3 Gen Scripts\Usb3GenTest1.usb3g - 0 error(s)
```

8.20 Name Aliasing

You can specify different names for named generation language syntax objects, such as constants, settings, data patterns, templates, variables, generation procedures, instructions, and named instruction parameters.

For example, you can specify a name that is more understandable for a specific user. You can also use the same object in different contexts.

After a name alias is created, for the script parser, the **alias_name** is equivalent to the **alias_value**.

Note: Though you can create many name aliases for the same alias value, you cannot redefine a name alias that was defined before. Also, you cannot create aliases for language keywords, so syntax constructions (like **Send**) can never be named objects.

Format

```
%alias Alias_Name = Alias_Value
```

Examples

```
%alias TxScr   = TxScramble
%alias RxDescr = RxDescramble
%alias DPacket = Usb3LongTemplateNameDataPacket

Set TxScr = 1  # Use an alias for the 'TxScr' setting.

Packet Usb3LongTemplateNameDataPacket
{
    Data : *
}
Main
{
    Send Usb3LongTemplateNameDataPacket
    {
        Data = {AA BB CC DD}
    }
    # Use an alias for the 'Usb3LongTemplateNameDataPacket' template.
    # Send a packet based on the 'Usb3LongTemplateNameDataPacket'
    # template.
    # Using aliases for packet/structure templates allows you to
    # use short names instead of the long names defined in
    # large template libraries.
    Send DPacket
    {
        Data = {AA BB CC DD}
    }

      # It is allowed to declare an alias (the same as constant,
      # data pattern, and template) inside a generation procedure.
      %alias DP = Usb3LongTemplateNameDataPacket
      Send DP
      {
        Data = {11 22 33 44}
      }
}
```

8.21 Include Path Directive

This feature allows you to specify additional folders in which the parser looks for included files. By default, the script parser tries to include a file using a name specified in the **%include** directive. If it cannot find the file, it looks for the file in the **Application** directory.

Format:

```
%include_path [=] "include path"
```

Note: The **include_path** name is supposed to end with **"\"**. If it does not end with **"\"**, the script parser adds **"\"** automatically.

Examples

```
%include_path   "C:\Usb3Generation\"

# You can also use '=' after %include_path.
%include_path = "C:\Usb3Generation\Include\"

%include "MyDefs.ginc" # The parser looks for the file
                       # in the following folders:
                       # 1. Application folder
                       # 2. C:\Usb3Generation\
                       # 3. C:\Usb3Generation\Include\

Main
{
    # Do something.
}
```

9 How to Contact Teledyne LeCroy

Send e-mail…	psgsupport@teledyne.com
Contact support…	teledynelecroy.com/support/contact
Visit Teledyne LeCroy's web site…	teledynelecroy.com
Tell Teledyne LeCroy…	Report a problem to Teledyne LeCroy Support via e-mail by selecting **Help > Tell Teledyne LeCroy** from the application toolbar. This requires that an e-mail client be installed and configured on the host machine.