Protocol Solutions Group
3385 Scott Blvd., Santa Clara, CA 95054
Tel: +1/408.727.6600
Fax: +1/408.727.6622

UWB*Trainer*™ Exerciser

Generation Script Language

Reference Manual

Manual Version 1.02

For UWB*Tracer*/UWB*Trainer* Software Version 3.02

November 2007

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.

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

Trademarks and Servicemarks

CATC Trace, UWBTracer, *UWBTrainer*, and *BusEngine* are trademarks of LeCroy.

All other trademarks are property of their respective companies.

Copyright

Version

This is version 1.02 of the *UWBTrainer Generation Script Language Reference Manual*.
This manual applies to UWB*Trainer* software version 3.02 and higher.

Contents

1 INTRODUCTION 1
1.1 Declaration Conventions 1
1.1.1 Parentheses 1
1.1.2 Brackets 1
1.2 Script Example Highlighting 1

2 SCRIPT LANGUAGE STRUCTURE 2
2.1 Generation Script Structure 2
2.2 Main Procedure and Other Procedures 2

3 COMMENTS 3
3.1 Line Comment 3
3.2 Block Comment 3

4 FILE-INCLUDING DIRECTIVES 4
4.1 Inline Directive 4
4.2 Include Directive 4

5 CONSTANT DECLARATIONS 5
5.1 Predefined Constants 5
5.2 Constant Definition Examples 5

6 DATA PATTERN DECLARATIONS 6
6.1 Constants and Data Patterns in Declarations 6
6.2 Leading Zeroes 6

7 GLOBAL GENERATION SETTINGS 7
7.1 Generation Settings 7
7.2 Simulation Settings 10
7.3 Generation Settings Syntax 11
7.4 Generation Settings outside Procedures 11
7.5 Generation Settings inside Procedures 12
7.6 Arithmetic Expressions in Constants 12

8 FRAME AND STRUCTURE TEMPLATE DECLARATIONS 13
8.1 Predefined Frame/Structure Templates 13
8.2 Field Definitions 14
8.2.1 Defining a Field at a Specific Offset 14
8.2.2 Defining a Field at the Current Offset 15
8.2.3 Defining a Field with Variable Length 16
8.2.4 Defining a Default Field Value 17
8.2.5 Specifying Byte Order in Field Definitions 18
8.2.6 Using Byte Stream Literals in Field Assignments 18
8.2.7 List of Possible Field Values Attribute 19
8.2.8 Defining Subfields 20
8.3 Constants/Arithmetic Expressions in Template Declarations 22
8.4 Frame Template Inheritance 24
8.4.1 Frame Template Single Inheritance 24
8.4.2 Frame Template Multiple Inheritance 26
8.4.3 Frame Template Insert Directive 28
8.4.4 Complex Frame Templates 30

8.5 Frame Template Multi-byte Field Byte Order Attribute 32
8.5.1 Big Endian Byte Order 32
8.5.2 Another Example 33
8.6 Structure Declaration Examples 35

9 GENERATION PROCEDURES 36

9.1 Send Frame Instruction 36
9.2 Using Local Fields in Send Frame Instructions 36
9.3 Instruction Parameters 38
9.4 Structure Variable Syntax 42
9.4.1 Omitting the Send Keyword 42
9.5 Changing a Generation Setting 43
9.6 Register Condition Instructions 44
9.6.1 Register Rx Frame Header+Payload Condition Instruction 44
9.7 Timer Instructions 50
9.7.1 Start Timer Instruction 50
9.7.2 Stop Timer Instruction 51
9.7.3 Reset Timer Instruction 51
9.7.4 Sleep Instruction 52
9.8 Revoke Condition Instruction 53
9.9 Reset Condition Instruction 54
9.10 Wait Instructions 55
9.11 Until Instructions 56
9.12 Wait for the Next SuperFrame Instruction 58
9.13 TxSleep Instruction 59
9.14 Wait Frame Shortcut Instructions 60
9.15 If Instructions 62
9.16 Loop Instruction 64
9.17 BreakLoop Instruction 65
9.18 Exit Instruction 66
9.19 Analyzer Control Instructions 67
9.19.1 StartRecording Instruction 67
9.19.2 StopRecording Instruction 68
9.19.3 TriggerAnalyzer instruction 69
9.19.4 Trace Instruction 69
9.19.5 Trace_B Instruction 70

10 ADVANCED SCRIPT PARSER FEATURES 71

10.1 Local Numeric Parser Variables 71
10.2 Local Structure Parser Variables 72
10.3 Using Local Fields in Structure Variables 74
10.4 Changing Structure Parser Variables 75
10.5 Sending Frames using Structure Variables 79
10.6 Using Special Data Pattern Creators in Field Assignments 80
10.7 Using Structure Variables to Assign Field Values 81
10.8 Using Multipliers to Assign Field Values 82
10.9 Using the Append Operator in Field Assignments 83
10.10 Initializing Struct Variables from Hex Streams 84
10.10.1 Assignments for Variables with Fixed-length Fields 84
10.10.2 Assignments for Variables with Variable Length Fields 85
10.11 Sizeof Operators 87
10.12 Preprocessor Integer Arithmetic 88
10.13 Preprocessor If Operator 89
10.14 Preprocessor Loop Operators 90
10.15 Forward Declarations 91
10.16 RAND Token 92

10.17 RandStream(n) Primitive 92
10.18 Global Numeric and Structure Variables 93
10.19 Using the Call Directive for Generation Procedure Insertions 94
10.19.1 Calling Another Generation Procedure with Parameters 94
10.19.2 Calling a Generation Procedure with No Parameters 98
10.19.3 Nested Calls Using Global Variables 99
10.20 Parser Tracing Functions 100
10.20.1 PTrace( ) : Parser Trace 100
10.20.2 PTraceVar( ) : Parser Trace Variable 101
10.20.3 PtraceVarEx( ) : Parser Trace Variable Extended 102
10.20.4 PtraceTemplate( ) : Parser Trace Template 105
10.21 Name Aliasing 107
10.22 Include Path Directive 108

11 APPENDIX A – GENERATION SCRIPT EXAMPLE 109

12 HOW TO CONTACT LECROY 116

List of Instructions, Comments, Definitions, Directives, Operators, Parameters, Primitives, Procedures, and Settings

AbsTime_parameter ... 38
AddMissingFields_setting ... 8
aliasing ... 107
append operator ... 83
AutoReset_parameter ... 50
block_comment ... 3
BreakLoop ... 73
Burst_parameter ... 38
Call directive ... 94
Commentss ... 3
ConditionRegisterWMRxFrame instruction ... 44
ConditionReset instruction ... 54
ConditionRevoke instruction ... 53
constants instruction ... 5
Count_parameter ... 45
DataPattern instruction ... 6
Delay_parameter ... 38
DelayNs_parameter ... 59
Exit ... 74
fields_definition ... 66
fld_size_operator ... 14
ForceStopRec_parameter ... 87
Frame instruction ... 68
FrameDelay_setting ... 13
If_All instruction ... 62
If_Any instruction ... 62
If_Condition instruction ... 62
%include directive ... 108
%inline directive ... 4
KeepOldTrace_parameter ... 67
line_comment ... 3
Local instruction ... 71
Loop instruction ... 64
Main procedure ... 2
MaxLoopIterCount_parser_setting ... 90
MaxLoopIterCount _setting ... 8
multiplier instruction ... 82
Name_parameter ... 45
Override_parameter ... 38
Packet instruction ... 13
pkt_size_operator ... 87
directive ... 89
PTrace instruction ... 100
PTraceTemplate instruction ... 105
PTraceVar instruction ... 101
PTraceVarEx instruction ... 102
pttn_size_operator ... 87
RAND token ... 92
RandSeed_setting ... 8
RandStream_primitive ... 92
Receive_setting ... 8

RecOpt_parameter ... 67
RegRxFrmCondition instruction ... 45
ResetCondition instruction ... 54
ResetTimer ... 59
RevokeCondition instruction ... 53
RxSimulationMode_setting ... 10
Send instruction ... 36
Set instruction ... 11
SFOffset_parameter ... 38
SimAnalyzerPhy_setting ... 10
SimUseElseBranch_setting ... 10
Sleep instruction ... 52
StartRecording instruction ... 67
StartTimer instruction ... 50
StopRecording instruction ... 68
StopTimer ... 59
Struct instruction ... 13
subfields_definition ... 20
SuperFramePeriod_setting ... 8
TimeAdjNs_parameter ... 38
Timeout_parameter ... 52
TimeVar_parameter ... 50
Trace instruction ... 69
Trace_B instruction ... 70
TraceName_parameter ... 67
TriggerAnalyzer ... 77
TxSleep instruction ... 59
Until instruction ... 56
Until_All instruction ... 56
Until_Any instruction ... 56
UwbRxChannel_setting ... 7
UwbTxChannel_setting ... 7
UwbTxPower_setting ... 7
Wait instruction ... 55
Wait_All instruction ... 55
Wait_Any instruction ... 55
WaitForNextSuperFrame instruction ... 58
WaitForTrace_parameter ... 68
WaitRxFrame instruction ... 60

[blank page]

1 Introduction

The UWB*Trainer*™ Generation Script Language allows you to create UWB traffic generation scenarios for UWB*Trainer* devices. The Generation Script Language allows you to implement even complicated generation scenarios.

Typically, Generation Script 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.

Also, Generation Script Language constructions are not case-sensitive.

This document describes the commands and syntax of the UWB*Trainer* Generation Script Language.

1.1 Declaration Conventions

1.1.1 Parentheses

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

(Packet | Frame | Struct) Template_Name …

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

1.1.2 Brackets

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

(Packet | Frame | 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

Generation Script Language examples in this document show syntax highlighting similar to that in the UWB*Trainer* 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
  - Frame/structure templates
  - Global numeric variables
  - Global structure variables (declare a template for a variable before declaring a variable)
- Generation procedures
  - List of generation instructions

Note: The parser can use previously declared objects in later declarations. In generation procedures, the parser can use objects assigned before their declaration.

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 the 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. Constants are DWORD (unsigned integer) values only.

5.1 Predefined Constants

For information about the predefined constants, see the **genconstants.ginc** file in the **Generation\Include** folder in the **Application** directory.

5.2 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 Hex Data" = 0xCDCDBEBE
const HDR_ERR_SIZE      = 8
const RATE_53MBS = 0

# The parser can use arithmetic operations in constant definitions.
const TX_PAYLOAD_OFFSET = 15 * 8   # Payload offset(in bits) for
                                   # the Tx frame

# Payload offset(in bits) for the Rx frame
const RX_PAYLOAD_OFFSET = TX_PAYLOAD_OFFSET + HDR_ERR_SIZE

# Example of a complex name in an expression
const MY_Data = ["Some Hex Data"] + 12
```

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 }
```

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 Global Generation Settings

A generation script can define global parameters, called "generation settings", that affect aspects of script compilation and execution. The generation settings are set using the **Set** script keyword.

7.1 Generation Settings

Currently the following settings are supported:

Setting	Description
UwbTxPower	Specifies the level of transmitter power. Possible values are 0 (no transmitting) to 15 (max level). Default value: 15
UwbTxChannel	Specifies the wireless channel that UWB*Trainer*™ uses to transmit WiMedia traffic. The setting value for channel mapping (see Figure 7.1 below) includes: **Band Group** (3 bits) **TF Code** (3 bits) as specified in the WiMedia PHY Specification. Default value: AUTO (UWB*Trainer* uses the currently specified channel.)
UwbRxChannel	Specifies the wireless channel that UWB*Trainer* uses to receive WiMedia traffic. The setting value for channel mapping (see Figure 7.1 below) includes: **Band Group** (3 bits) **TF Code** (3 bits) as specified in the WiMedia PHY Specification. Default value: AUTO (UWB*Trainer* uses the currently specified channel.)

Receive	Allows you to turn OFF(0) or ON(1) the UWB*Trainer* receiver. Default value: OFF (0) If the UWB*Trainer* receiver is on, then UWB*Trainer* receives and processes Rx traffic even if the script scenario is not concerned about it. This can significantly decrease script performance (even for just only Tx generation scenarios), because some time is needed to process Rx traffic. So it is better to turn it off when there is no need to process Rx traffic. **Note**: **Register Rx Frame** condition instruction automatically turns on the receiver. It remains active and processes Rx traffic until it is turned off by this setting: `Set Receive = OFF` or `Set Receive = 0`
SuperFramePeriod	Specifies the **SuperFrame** period in microseconds. Possible values are 512 to 0xFFFFFF. Default value: AUTO (65,536 microseconds)
FrameDelay	Specifies the default delay in nanoseconds between transmitted frames. Default value: 0 (Frames are transmitted without delay.) **Note**: This value can be overridden by the **Delay** parameter in a **Send Frame** instruction.
AddMissingFields	Instructs the script parser to add any missing frame template fields for **Send Frame** instructions. Missing TX frame template fields might include **FCS**. Missing RX frame template fields might include **FCS +Rx** packet end report fields (such as **RSSI** and **LQI**), which are used in conditions. Default value: ON
RandSeed	Allows you to set an integer seed for the pseudo-random generator used by the RAND token.
MaxLoopIterCount	Allows you to change the maximum total number of preprocessor loop iterations in a generation script. Possible values are 1 to 100,000. Default value: 20,000

Blocking	Set the default behavior of **Send Frame** instructions. If set to TRUE, the **Send Frame** instruction is blocked until the frame is sent (unless overridden in the **Send Frame** parameter). If set to FALSE, **Send Frame** will continue executing script instructions, even if the **Send** command is still queued in the output FIFO and not yet executed (unless overridden in the instruction parameter). Default: FALSE

Channel Number (decimal)	Channel Number (octal)	(Band Group, TF Code)	Mandatory / Optional
9 - 15	011 - 017	(1, 1 - 7)	Mandatory
17 - 23	021 - 027	(2, 1 - 7)	Optional
25 - 31	031 - 037	(3, 1 - 7)	Optional
33 - 39	041 - 047	(4, 1 - 7)	Optional
45 - 46	055 - 056	(5, 5 - 6)	Optional

Figure 7.1: Channel Mapping (PHY Spec 1.1 Table 7.7)

7.2 Simulation Settings

This group of settings does not affect generation output for UWB*Trainer* in any way. Only the UWB*Trainer* application trace simulation tool uses these settings. This tool allows you to convert generation scenarios into UWB*Tracer* trace files and preview traffic generated by the UWB*Trainer* device. (For more details about the simulation tool, please refer to the UWB*Trainer* user manual.)

Setting	Description
RxSimulationMode	If set to 1, specifies that the generation scenario should be converted to an RF trace file as it might be recorded by a UWB*Tracer* analyzer recording RF traffic. If set to 0, specifies that the generation scenario should be converted to an MPI trace file as it might be recorded by a UWB*Tracer* analyzer recording MPI traffic. Default value: 1 (RF simulation mode)

SimAnalyzerPhy	Specifies simulation analyzer PHY settings for RF simulation mode. It is used only when **RxSimulationMode** = 1.
	Possible values (constants are defined in **gen_constants.ginc**) are:
	`PHY_WISAIR_REV_B = 0x11`
	`PHY_WISAIR_REV_C = 0x12`
	`PHY_WISAIR_REV_532 = 0x13`
	`PHY_ALEREON = 0x30`
	Default value: PHY_WISAIR_REV_C

SimUseElseBranch	If set to 1, specifies that, whenever the simulation tool sees the **else_condition** branch, it takes those branch instructions for the simulated trace file and ignores **if_condition** branch instructions. If set to 0, specifies that, whenever the simulation tool sees the **if_condition** branch, it takes those branch instructions for the simulated trace file and ignores **else_condition** branch instructions. Default value: 0 (simulation tool uses **if_condition** branch instructions for a simulated trace file)

7.3 Generation Settings Syntax

The syntax for generation numeric and string settings is:

Set NumericSetting = numeric value

Set StringSetting = "string value"

Examples

```
# Default time in nanoseconds after the previous frame
# to send the next frame.
Set FrameDelay   = 2000000
Set UwbTxPower   = 8
Set UwbTxChannel = 9
```

7.4 Generation Settings outside Procedures

Generation settings specified outside the generation procedures are set before the first script generation instruction is executed, no matter where they appear in the script. The two examples below are equivalent.

Example 1

```
# Set a setting before the first instruction is executed.
Set FrameDelay = 20000
Main
{
    # Some generation instructions
}
```

Example 2

```
# Set a setting before the first instruction is executed.
Main
{
    # Some generation instructions
}
# Set before the first instruction is executed,
# though the Set line is below the instruction in the script.
Set FrameDelay = 12000
```

7.5 Generation Settings inside Procedures

Generation settings specified inside a generation procedure are set only during script execution by the UWB*Trainer*™ device.

```
Main
{
    Set FrameDelay = 24000 # Set during runtime.
}
```

7.6 Arithmetic Expressions in Constants

You can use arithmetic expressions, such as

+, -, *, /, %, >>, <<, &, |, ^, and ()

to define setting values.

Example

```
Const SOME_CONST = 5
set FrameDelay = ( 2000000 + 5 ) >> ( 8 + SOME_CONST )
```

8 Frame and Structure Template Declarations

Frame/structure declarations declare named frame/structure objects. A frame/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 frame/structure fields.

Note[1]: Templates can inherit field layouts from other templates through ancestor lists.

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

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

]

Note[2]**:** For frame/structure template declarations, you can use the **Frame** or **Struct** keyword. However, structures are supposed to be used as building blocks for constructing frame payloads (or similar purposes), rather than for describing full frames.

8.1 Predefined Frame/Structure Templates

LeCroy PSG provides some useful WiMedia frame templates that can be used in generation scripts. The predefined frame/structure templates are listed in the file:

UWBTracer\Generation\Include\main_pkt_templates.ginc

8.2 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, register condition instructions, and when sending a frame. See sections “Changing Structure Parser Variables” and “Send Frame Instruction” for more information.

8.2.1 Defining a Field at a Specific Offset

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

Examples

```
const F3_OFFSET = 16
const F3_LEN    = 8

# Declare the frame template 'SomeTemplate.
Frame 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.
}
```

8.2.2 Defining a Field at the Current Offset

If the field offset is omitted, then 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 frame template 'SomeTemplate.
Frame 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.
}
```

8.2.3 Defining a Field with Variable Length

If the field length is set to *****, then 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, then the field's length is set to 0.

Note: When a value is assigned to a variable length field, then 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 frame template 'SomeTemplate'.
Frame 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.
    FCS  : 32 # Declare the 32-bit field 'FCS' at offset 16+32.
}

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

8.2.4 Defining a Default Field Value

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

Note: When a value is assigned to a variable length field, then 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 frame template 'SomeTemplate.
Frame 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.
}
```

8.2.5 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.
Frame Mixed
{
    F1 : 16
    F2 : 32 (MSB)
    F3 : 16 (MSB) = 0xAABB
    F4 : 32       = 0xAABBCCDD
}

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

8.2.6 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 }
```

8.2.7 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 UWB*Trainer*™ Application Development Environment (UWB*Trainer* Script Editor Intellisense and Graphic Scenario Builder) 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 }

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

8.2.8 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, then 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 frame 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 frame with payload: 0A 00 EE
	Send Templ_2
	{
	    F1  = 0xA
	    SF1 = 0xEE
	}
}

Frame PLCP_PART
{
	FrameCtrl
	{
	    Reserved3 : 2
	    Retry     : 1
	    CtrlType  : 4
	    FrameType : 3
	    AckPolicy : 2
	    Secure    : 1
	    Version   : 3
	}
	DestAddr  : 16
	SrcAddr   : 16
}
```

8.3 Constants/Arithmetic Expressions in Template Declarations

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

Examples

```
const BM_LENGTH = 1

# Declare the frame template 'PLCP'.
Frame PLCP
{
    # PHY HEADER
    # Declare the field 'Rate' ( offset : 0, length : 5 bits ), having
    # default value RATE_53MBS ( 0 ).
    Rate      : 5 = RATE_53MBS
    Reserved0 : 3
    Len7_0    : 8
    Scr       : 2
    Reserved1 : 2
    Len11_8   : 4
    BG        : 1
    TFCode    : 3 = 1
    PreType   : 1
    BM        : BM_LENGTH  # Constant in a field definition.
    Reserved2 : 10
```

```
    # MAC HEADER
    #------- Frame Control Multi-byte Field -------
    # Example of a complex multi-byte field.
    # A multi-byte complex field has a subfield layout.
    # Only one level of subfields is currently allowed.
    FrameCtrl
    {
        Reserved3 : 2
        Retry     : 1
        CtrlType  : 4
        FrameType : 3
        AckPolicy : 2
        Secure    : 1
        Version   : 3
        DestAddr  : 16
        SrcAddr   : 16
    }

    #------- Sequence Control -------
    # Another example of a complex multi-byte field.
    SeqCtrl
    {
        Reserved4 : 1
        MoreFrag  : 1
        SeqNum    : 11
        FragNum   : 3
    }

    #------- Access Information -------
    # Another example of a complex multi-byte field.
    AccInfo
    {
        AccMthd   : 1
        MoreFrms  : 1
        Duration  : 14
    }
}
```

8.4 Frame Template Inheritance

You can create a frame/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 frame template after it adds fields from the inherited templates.

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

8.4.1 Frame Template Single Inheritance

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

Examples

```
Frame TX_FRAME : PLCP
{
    # Example of a field with variable length having a default value.
    Data     : * = { 00 00 00 01 }

    # The parser can use a declared data pattern name.
    FCS      : 32
    # This field is calculated automatically if autocorrect settings
    # are on in the SendFrame instruction.
}
# Other examples of frame templates:
# PHY-MAC header for RX frame + header error fields
Frame PLCP_RX : PLCP
{
    HDERsvd : 3
    HDError : 5
}
# Template for generic TX frame
Frame TX_FRAME : PLCP_TX
{
    Data : *
    FCS  : 32  # calculated automatically
}
```

```
# Template for generic RX frame
Frame RX_FRAME : PLCP_RX
{
    Data    : *
    FCS     : 32  # Calculated automatically
    RSSI    : 8
    LQI     : 8
    RXERsvd : 3
    RXError : 5
}

# Example of declaration of some field at some offset and
# declaration of some consecutive fields after that.
const OPCODE_OFFSET = 128
Frame TX_FRAME_1 : TX_FRAME
{
    # The field 'FrameType' now has a different default value.
    # For example, the constant DATA = 3
    FrameType = DATA
    # Declare the field 'Opcode' ( offset : 128  , length : 8  )
    # having default value 0x2A.
    Opcode : OPCODE_OFFSET, 8 = 0x2A

    # Declare the field 'LBA'(offset : 128+8, length : 64)
    LBA : 64
}
# Example of using arithmetic and data pattern names
# in field definitions.
Frame MY_TX : TX_FRAME
{
    # Declaration of the field at offset: 18 * 8 bits, length: 8 bits
    MyLen : TX_PAYLOAD_OFFSET + (3 * 8), 8

    # Setting a field value using a previously declared data pattern.
    Data = PATTERN_4
}
```

8.4.2 Frame Template Multiple Inheritance

You can create a frame/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

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

Frame Templ_0
{
    FieldT0_8  : 8
    FieldT0_16 : 16
}

Frame Templ_1
{
    FieldT1_24 : 24
    FieldT1_32 : 32
}

Frame 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

```
# Frame payload fields
struct FramePldFields
{
    Data : *
    FCS  : 32
}

# RX frame header error fields
Struct HdrErrFields
{
    HDERsvd : 3
    HDError : 5
}

# RX frame payload error fields
Struct RxErrFields
{
    RSSI    : 8
    LQI     : 8
    RXERsvd : 3
    RXError : 5
}

# Template for generic TX frame
Frame TX_FRAME : PLCP, FramePldFields
{
}

# Template for generic RX frame
Frame RX_FRAME : PLCP, HdrErrFields, FramePldFields, RxErrFields
{
}
```

8.4.3 Frame 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

```
Frame Base
{
    F1 : 16
    F2 : 8
    F3 : 32
}
Frame Templ_0
{
    FieldT0_8  : 8
    FieldT0_16 : 16
}
Frame Templ_1
{
    FieldT1_24 : 24
    FieldT1_32 : 32
}
Frame Combined : Base
{
    Cmb_F1 : 8
    insert Templ_0 # Insert fields from frame template Templ_0.
    Cmb_F2 : 16
    insert Templ_1 # Insert fields from frame 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

```
Frame Combined : Base
{
    Cmb_F1 : 8
      : Templ_0  # insert fields from frame template Templ_0
    Cmb_F2 : 16
      : Templ_1  # insert fields from frame template Templ_1
    Data   : 32
}
```

8.4.4 Complex Frame Templates

Template insertions can simplify construction of complex templates.

Examples

```
# Frame payload fields
struct FramePldFields
{
    Data : *
    FCS  : 32
}

# RX frame header error fields
Struct HdrErrFields
{
    HDERsvd : 3
    HDError : 5
}

# RX frame payload error fields
Struct RxErrFields
{
    RSSI    : 8
    LQI     : 8
    RXERsvd : 3
    RXError : 5
}

# Template for generic TX frame
Frame TX_FRAME : PLCP
{
    : FramePldFields
}

# Template for generic RX frame
Frame RX_FRAME : PLCP
{
    : HdrErrFields
    : FramePldFields
    : RxErrFields
}
```

```
# Or use only template insertions instead of template inheritance
# Template for generic TX frame
Frame TX_FRAME
{
    : PLCP
    : FramePldFields
}

# Template for generic RX frame
Frame RX_FRAME
{
    : PLCP
    : HdrErrFields
    : FramePldFields
    : RxErrFields
}
```

8.5 Frame 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

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

The byte stream based on this template is:

`FF EE DD CC BB AA`

8.5.1 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

```
Frame 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.

8.5.2 Another Example

Example

```
struct IPv4Header_ (MSB)
# Instructs that all short fields have MSB to LSB byte order
# when assigned values.
{
    Version        : 4 = 4
    IHL            : 4
    TypeOfSrv      : 8
    TotalLen       : 16
    Identification : 16
    Flags          : 3
    Offset         : 13
    TTL            : 8
    Protocol       : 8
    HdrCheckSum    : 16
    SourceAddr     : 32
    DestAddr       : 32
}
```

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

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

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

Frame 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)
```

8.6 Structure Declaration Examples

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

Note: Also see the examples using the **Struct** keyword to assign frame 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
}
```

9 Generation Procedures

A generation (or instruction) procedure typically contains a list of generation scenario instructions and script parser "preprocessor" directives.

- Generation scenario instructions are translated into generation instructions executed by the UWB*Trainer*™.device (for example, **Send Frame**).
- Script parser "preprocessor" directives are executed during parsing and are not sent to the UWB*Trainer* (for example, parser arithmetic expressions and loops).

Note: The procedure with the name **Main** is the entry point of the generation stream and must be present.

9.1 Send Frame Instruction

To send a frame, write the **Send** keyword followed by the name of the frame template or frame variable in the generation procedure and provide a list of parameters that should override the default instruction parameters, if any.

Note: For information about using frame variables, see the "Declaring frame variables" section.

The script parser constructs a **Send Frame** instruction based on the frame template or frame variable, plus the provided parameters, and sends it to UWB*Trainer*.

Format

```
# Basic Send instruction
Send (Frame Template name | Structure variable name )
[
    {
        Field = Value
        ...
    }

    ( frame delay, superframe offset, absolute time,
      nanosecond time adjustment, burst mode switch,
      override bitmap, time variable index, blocking )

]
```

9.2 Using Local Fields in Send Frame Instructions

You can add local fields to **Send** instructions. You can use them to add to the existing frame template or structure variable or to override specified offsets within the declared structures. These local fields only exist for the specific instance of the **Send** instruction in which they are declared. The frame template or structure variable remains unmodified and does not retain the local fields created. (To learn how to retain local fields, see the section “Using Local Fields in Structure Variables”) With local fields, you may quickly assign a portion of a larger field using bit offsets or append information to a specific packet without having to redefine a template. (Local fields are also applicable to Register Condition Instructions.)

Format

```
# Send instruction with a local field appended.
Send (Frame Template name | Structure variable name )
[
    ( frame delay, superframe offset, absolute time,
      nanosecond time adjustment, burst mode switch,
      override bitmap, time variable index, blocking )

    {
        Field = Value
        ...
        # Declare local field “Lfield” 48 bits long at the packet end.
        LField : 48 = { 01 02 03 04 05 06 }
    }

]

# Send instruction with a local field overwriting an offset.
Send (Frame Template name | Structure variable name )
[
    {
        Field = Value
        ...
        # Declare local field “FirstByte” 8 bits long at offset 0.
        FirstByte : 0,8 = { 01 }
    }

    ( frame delay, superframe offset, absolute time,
      nanosecond time adjustment, burst mode switch,
      override bitmap, time variable index, blocking )

]

# Send instruction with local fields appended and overwrite an offset.
Send (Frame Template name | Structure variable name )
[
    ( frame delay, superframe offset, absolute time,
      nanosecond time adjustment, burst mode switch,
      override bitmap, time variable index, blocking )

    {
        Field = Value
        ...
        # Declare local field “Lfield” 48 bits long at packet end.
        LField : 48 = { 01 02 03 04 05 06 }
        # Declare local field “FirstByte” 8 bits long at offset 0.
        FirstByte : 0,8 = { 01 }
    }
]
```

Note[1]: The instruction parameters **“(…)”** and frame/structure field assignments **“{ … }”** are interchangeable in order within the **Send** instruction.

Note[2]: Declare local fields within the **Send** instruction in the same way that frame template fields are declared. They exist only for the instruction in which they are declared.

9.3 Instruction Parameters

The parser searches the template list first.

- If a template with the specified name is not found, it searches the local generation procedure structure variable list.
- If a local generation procedure structure variable is not found, the parser searches the global structure variable list.
- If a structure variable is not found, the system reports a parser error.

The named instruction parameters are:

- **Delay**: Delay time (in **micro**seconds) to wait before sending the frame.
- **SFOffset**: Offset (in **micro**seconds) of the frame to be sent, in **SuperFrame**.
- **AbsTime**: Absolute time (in **micro**seconds) at which the frame is to be sent. **Note:** When using the AbsTime time mode parameter, make sure that the first frame sent has a time value greater than or equal to 10 microseconds, to ensure correct timing between the first few frames of the scenario.
- **TimeAdjNs**: Time adjustment (in **nano**seconds) for **Delay, SFOffset, or AbsTime.**
- **Burst**: BURST_BEGIN specifies the start of a burst sequence. BURST_CONTINUE specifies that the burst sequence does not end. BURST_END specifies the end of a burst sequence. IGNORE means not to use a burst sequence.
- **Override**: Specifies a bitmap that instructs the UWB*Trainer* to set values for the **TFC/BG, FCS, Scr**, and **Length** fields from the frame template, rather than calculating them automatically.

Send Frame Override Field Instruction Parameters (Found in "gen_constants.ginc")	
TFC/BG	OVR_TFC = 0x01
Length	OVR_LEN = 0x02
Scrambler	OVR_SCR = 0x04
FCS	OVR_FCS = 0x08
No Override	OVR_NONE = 0
Override All	OVR_ALL = (OVR_TFC | OVR_LEN | OVR_SCR | OVR_FCS)

- **TimeVar:** Specifies the **Time** variable index associated with a condition registered in a Register Condition instruction. The **Time** variable value **+** the **Delay** is the absolute time the frame will be sent. The TimeVar value range is from 0 to 255 and must have been declared explicitly in a previous **RegRxFrmCondition, ConditionRegisterWMRxFrame, StartTimer**, or **WaitRxFrame** instruction.
- **Blocking:** If TRUE, the frame will be sent out over the air before the following instruction is executed. If FALSE, the **Send Frame** instruction will be output into a FIFO, but instruction execution of the script will continue even if the **Send Frame** instruction is still pending. This allows queuing a series of **SendFrame** commands with little latency between them. The normal default value is FALSE, but the default can be overridden by the **Blocking** setting.

The default for all the parameters is UNDEFINED (0xFFFFFFFF).

Note[1]: Set **Burst** to IGNORE if no bursts are being sent. Do not set it to OFF, because the parser interprets OFF as the end of a Burst sequence and generates parse errors if a Frame was never sent with **Burst** ON.

Note[2]: You must associate a **Time** variable with a condition before using it in a **Send Frame** instruction. The script parser generates a parsing error if the **TimeVar** parameter contains an index of a **Time** variable not declared in any **Register Condition** instruction.

The purpose of associating a **Time** variable with a condition is that, when the condition triggers, the condition event timestamp is stored in the **Time** variable associated with the condition. Afterwards, the timestamp provides precise scheduling frame transmission because it specifies the delay after the event occurred.

Delay, **SFOffset**, and **AbsTime** are all mutually exclusive. Set only one of these three timing parameters to a value in a **Send** instruction, or a parse error will be generated.

Delimit instruction parameters using a comma, as shown in the following examples:

To send a TX_Frame with Delay = 0, Indicate End of a Burst, and No Override:

Send TX_FRAME(Delay = 0, Burst = OFF, Override = OVR_NONE)

To send a TX_Frame with SFOffset = 10 us, Indicate End of a Burst, and No Override:

Send TX_FRAME(SFOffset = 10, Burst = OFF, Override = OVR_NONE)

To send a TX_Frame with AbsTime = 250 us, Indicate End of a Burst, and No Override:

Send TX_FRAME(AbsTime = 250, Burst = OFF, Override = OVR_NONE)

To send a TX_Frame with Delay = 200 us, Indicate End of a Burst, and No Override:

Send TX_FRAME(200, UNDEFINED, UNDEFINED, 0, OFF, OVR_NONE)

To send a TX_Frame with Delay = 400 us:

Send TX_FRAME(400)

To send a TX_Frame and block until it has been sent out over the air:

Send TX_FRAME(Blocking = TRUE)

To send a TX_Frame with Time Variable Index = 1 and Delay = 400 us:

Send TX_FRAME(TimeVar = 1, Delay = 400)

To send a TX_Frame with Delay = 20 us + 100 ns and Override Scrambler:

Send TX_FRAME(Delay = 20, TimeAdjNs = 100, Override = OVR_SCR)

To send a TX_Frame with Delay = 20 us + 100 ns and Override Length and FCS:

Send TX_FRAME(20, UNDEFINED, UNDEFINED, 100, UNDEFINED, (OVR_LEN | OVR_FCS))

To send a TX_Frame with SFOffset = 170 us without any Nanosecond adjustment, Indicate the beginning of a Burst, and Override TFC/BG:

Send TX_FRAME(UNDEFINED, 170, UNDEFINED, 0, ON, OVR_TFC)

To send a TX_Frame with Absolute Time = 250 us and Override All:

Send TX_FRAME(AbsTime = 250, Override = OVR_ALL)

Examples

```
Main
{
    # Send a frame based on the template 'TX_FRAME'
    # using default values.
    Send TX_FRAME

    # Send a frame based on the template 'TX_FRAME'
    # using default values overriding some default
    # instruction parameters.
    Send TX_FRAME ( Delay = 100 )
    Send TX_FRAME( 100 )  # the same as the previous instruction

    # Send a frame based on the template 'TX_FRAME_1'
    # using overridden values and changing the default delay setting.
    Send TX_FRAME_1
    {
        DestAddr  = 0xAABB
        SrcAddr   = 0xEFBE
        Data      = { AA BB CC DD 12 34 56 78 [PATTERN_2] }
    }

    ( Delay = 100 )  # Set the 'Delay' to 100000.

    # Send a frame and use provided FCS instead of
    # automatic calculation.
    Send TX_FRAME
    {
        DestAddr = 0xAABB
        SrcAddr  = 0xEFBE
        Data     = { AA BB CC DD }
        FCS      = 0x11223344  # This value is used as FCS
                               # for this frame.
    }

    ( Override = OVR_FCS )
```

```
    # Register an RxFrame condition and keep the timestamp of
    # the condition event in Time variable #1.
    RegRxFrmCondition( HDR_COND, TimeVar = 1 )
    {
        # Header Mask/Match
        PLCP
        {
             DestAddr = 0xAABB
             SrcAddr  = 0xBEEF
        }
    }

    # Register a Timer condition and keep the timestamp of
    # the condition event in Time variable #1.
    StartTimer ( 2000, TimeVar = 1 )

    # Wait for any condition to occur.
    Wait ( HDR_COND, TIMER )

    # Send frame 100 us after either a timer or RxHdr event occurs.
    # Note that both conditions specify the same Time variable index:
    # No matter which condition event happens first, the Time variable
    # keeps the timestamp of either of them.
    Send TX_FRAME( TimeVar = 1, Delay = 100 )
    {
        DestAddr = 0xAABB
        SrcAddr  = 0xEFBE
    }

}
```

9.4 Structure Variable Syntax

Structure variables have a special syntax.

Example

```
# This syntax instructs the parser to send a frame based on
# some structure variables, avoiding the search for templates.
Send $(Structure variable name)
[
    {
        Field = Value
        ...
    }

    ( parameter1, parameter2, ... )

]
```

Example

```
Main
{
    # Declare a structure variable based on the template 'TX_FRAME'.
    $X = TX_FRAME
    # Send a frame based on the template 'TX_FRAME'
    # using default values.
    Send $X
}
```

9.4.1 Omitting the Send Keyword

You can omit the **Send** keyword if needed (but this is not recommended).

Example

```
Main
{
    # Send a frame based on the template 'TX_FRAME'
    # using default values.
    TX_FRAME
}
```

9.5 Changing a Generation Setting

The parser can change some generation settings during script execution.

Format

```
Set Setting = Value
```

Example

```
Main
{
    Send TX_FRAME
    Send TX_FRAME

    Set TxPower = 8  # Change TxPower after sending two frames.

    Send TX_FRAME
    Send TX_FRAME
```

9.6 Register Condition Instructions

You can register different types of "conditions" for which to wait or to use in flow elements, such as the **If** element.

Important: At any place in a script, up to 15 user-defined register conditions may be in use simultaneously. (**Note:** In addition, the timer register condition named TIMER is always in use.) After you have defined 15 register conditions in a script, to define a new register condition you must first revoke a register condition that is no longer in use. To revoke a register condition that is no longer in use, use the **RevokeCondition** instruction.

9.6.1 Register Rx Frame Header+Payload Condition Instruction

You can register a condition for both the **Rx Frame** header and a payload.

Note[1]: Checking just the header condition works much faster than checking the combined condition.

Note[2]: The **Register Rx Frame** condition instruction automatically turns on the receiver. It remains active and processes Rx traffic until it is turned off by this setting.

Format

```
ConditionRegisterWMRxFrame ( condition_name, [auto_reset], [counter],
                             [Time variable index] )
[
    {
        Mask/Match pattern or frame variable
        Additional condition parameters
    }
]
```

Format in Short Form

```
RegRxFrmCondition ( condition_name, [auto_reset], [counter],
                    [Time variable index] )
[
    {
        Mask/Match pattern or frame variable
        Additional condition parameters
    }
]
```

This registers the **Rx Frame** (header + payload) condition so that the options are automatically reset when the condition is hit. It is triggered only if it occurs the number of times specified by the **counter** parameter.

Note[3]: If no **Mask/Match** pattern is specified, the system uses any **Rx frame**.

The instruction named parameters are:

Name: Condition name.

AutoReset: Specifies that the condition is automatically reset after it is triggered.

Count: Specifies the number of times the condition event occurs before the condition is triggered.

TimeVar: Specifies the index of the **Time** variable in which to keep the condition timestamp when it is triggered. Values can range from 0 to 255. No value indicates that this parameter is ignored.

By default, **AutoReset** is OFF, **Count** = 1, and **TimeVar** = UNDEFINED (timestamp is not saved).

Note[4]: If mask/match size is less or equal to the size of PHY-MAC header then condition-matching check is performed considerably faster.

Note[5]: You may declare local fields for frame templates and variable masking/matching. See section “Using Local Fields in Send Frame Instructions” section for more details.

9.6.1.1 Additional Rx Frame Header+Payload Condition parameters

You can specify additional matching parameters inside the Register Rx Frame Header +Payload Condition Instruction block that are not part of the PHY-MAC header and frame payload. Such parameters include RSSI, LQI, Frame duration ranges, and errors.

Format:

```
RxFrameInfo
{
    Parameter 1
    …
    Parameter N
}
```

The format of additional condition parameters is defined by the RxFrameInfo structure (defined in the **Generation\Include\ RxFrameInfo.ginc** file):

```
struct RxFrameInfo
{
    RSSI_Lo     : 8  # RSSI lower  limit
    RSSI_Hi     : 8  # RSSI upper  limit

    LQI_Lo      : 8  # LQI  lower limit
    LQI_Hi      : 8  # LQI  upper limit

    Duration_Lo : 32 # Frame duration lower limit in microseconds
    Duration_Hi : 32 # Frame duration upper limit in microseconds

    Reserved_0  : 8  # reserved value, should not be used

    Errors           # error flags
    {
        Reserved_1 : 2
        FrmAborted : 1 # Frame Aborted error flag
        FCSError   : 1 # FCS error flag
        LenError   : 1 # Frame Length error flag
        HdrError   : 1 # Frame Header error flag
        PldError   : 1 # Frame Payload error flag
        UnsptdRate : 1 # Unsupported Rate error flag
    }
}
```

Examples

```
Main
{
    # Register the Rx Frame condition named 'COND_1'.
    RegRxFrmCondition ( COND_1 )
    {
        WM_FRAME
        {
            # Header Mask/Match
            DestAddr = 0xAABB
            SrcAddr  = 0xBEEF
            # Payload Mask/Match
            Data     = {AA BB CC DD}
        }
    }

    # Register the Rx Frame condition named 'COND_2' using
    # the named parameters.
    RegRxFrmCondition ( COND_2,
                        AutoReset = YES,
                        Count     = 3,

                        TimeVar   = 1 )
    {
        WM_FRAME
        {
            # Header Mask/Match
            DestAddr = 0xAABB
            SrcAddr  = 0xBEEF
            # Payload Mask/Match
            Data     = {AA BB CC DD}
        }
    }

    # Register the RX Frame condition named "COND_3" with the same
    # parameters as in the previous instruction.
    RegRxFrmCondition ( COND_3, YES, 3, 1 )
    {
        WM_FRAME
        {
            # Header Mask/Match
            DestAddr = 0xAABB
            SrcAddr  = 0xBEEF
            # Payload Mask/Match
            Data     = {AA BB CC DD}
        }
    }
```

```
$X = WM_FRAME # Declare a frame variable.
{
  # Header Mask/Match
  DestAddr = 0xAABB
  SrcAddr  = 0xBEEF
  # Payload Mask/Match
  Data     = {AA BB CC DD}
 }

RegRxFrmCondition ( COND_2, 4, YES )
{
    X # Use a frame variable as mask/match.
}

# Register Rx Frame condition: Any Rx Frame with RSSI from 2 to 8.
RegRxFrmCondition( RX_FRM_COND )
{
    RxFrameInfo
    {
      RSSI_Lo = 2
      RSSI_Hi = 8
    }
}

# Register Rx Frame condition: Any Rx Frame with duration from
# 15 us to 18 us.

RegRxFrmCondition( RX_FRM_COND )
{
    RxFrameInfo
    {
        Duration_Lo = 15
        Duration_Lo = 18
     }
 }

# Register Rx Frame condition: Any Rx Frame with error FCS error
# or unsupported rate error.
RegRxFrmCondition( RX_FRM_COND )
{
    RxFrameInfo
   {
        Errors = ERR_FCS | ERR_UNSUPPORTED_RATE
   }
}
```

```
    # Register Rx Frame condition: Any Rx Frame with error FCS error
    # or Unsupported rate error and a lower duration of 65536-131072 us
    # ( the lower duration is checked using a declared local variable,
    # Upper_Duration_Lo, which is set to the upper byte of Duration_Lo)
    RegRxFrmCondition( RX_FRM_COND )
    {
       RxFrameInfo
       {
          FCSError   = 1
          UnsptdRate = 1
          Upper_Duration_Lo: 48,16 = 0x0001
       }
    }

    # Register Rx Frame condition: Any Data Frame with duration from
    # 15 us to 18 us.

    RegRxFrmCondition( RX_FRM_COND )
    {
        # RxFrameInfo and Frame hdr-payload mask/match pattern can be
        # in any order.

        RxFrameInfo
        {
            Duration_Lo = 15
            Duration_Hi = 18
        }

        # hdr-payload mask/match pattern
        PLCP
        {
           FrameType = DATA
        }
    }

}
```

9.7 Timer Instructions

You can register and manipulate the UWB*Trainer* microsecond resolution timer using Timer instructions.

9.7.1 Start Timer Instruction

This instruction registers a special timer condition named “TIMER” and starts the timer if the timer was already started with the new **timeout** value.

Format

```
StartTimer ( timer_value, [auto_reset], [Time variable index] )
```

Note[1]: If the **auto_reset** parameter is set to 1, then the timer starts counting again. If the **auto_reset** parameter is set to 0, then use a **ResetTimer** instruction to reset the timer.

The instruction named parameters are:

Timeout: Specifies the timer timeout in microseconds.
AutoReset: Specifies that the condition is automatically reset after it is triggered.
TimeVar: Specifies the index of the **Time** variable in which to keep the condition timestamp when it is triggered. Values can range from 0 to 255.
No value indicates that this parameter is ignored.

By default, **AutoReset** is OFF, **Timeout** = UNDEFINED (**Timeout** parameter must be set to a non-zero value to register a **Timer** condition), and **TimeVar** = UNDEFINED (timestamp is not saved).

Note[2]: The timer used in this instruction allows jitter of several microseconds. If you need a precise timer for scheduling frames, then use **TxSleep** instruction.

Example

```
Main
{
    # Start the timer with timeout 2000 microseconds.
    StartTimer ( 2000 )

    Wait( TIMER ) # Wait for the timer to elapse.
}
```

9.7.2 Stop Timer Instruction

This instruction stops the UWB*Trainer* microsecond resolution timer and revokes the “TIMER” condition.

Format

StopTimer

Example

```
Main
{
    # Start the timer with timeout 2000 microseconds and the auto_reset
    # option.
    StartTimer ( 2000, 1 )

    Wait( TIMER ) # Wait for the timer. The timer will be restarted.

    StopTimer  # Stop the timer and revoke the “TIMER” condition.
}
```

9.7.3 Reset Timer Instruction

This instruction resets the UWB*Trainer* microsecond resolution timer.

Format

ResetTimer

Example

```
Main
{
    # Start the timer with timeout 2000 microseconds.
    # The timer has to be reset after elapsing.
    StartTimer ( 2000 )

    Wait( TIMER ) # Wait for the timer. The timer has to be restarted.

    ResetTimer  # Reset the timer.

    Wait( TIMER ) # Wait for the timer again.
}
```

9.7.4 Sleep Instruction

This instruction instructs UWB*Trainer* to **Sleep** for the specified number of microseconds before processing the next instruction.

Format

Sleep (**sleep_value**)

The instruction named parameter is:

Timeout: Specifies the timer timeout in microseconds.

Note: The **sleep** instruction is a shortcut to using Timer and Wait instructions (see the Wait Instructions). The **sleep** instruction is exactly the same as the following set of instructions:

```
StartTimer ( timer_value )
Wait ( TIMER )
StopTimer
```

Example

```
Main
{
    # Sleep for 65538 microseconds.
    # The following instructions are executed afterwards:
    Sleep ( 65538 )

    Send FrameXYZ

    # This set of StartTimer, Wait, and StopTimer Instructions is the
    # same as the Sleep instruction above.

    StartTimer( 65538 ) # Start the timer with timeout 2000 usecs.

    Wait( TIMER )       # Wait for the timer. [Condition “TIMER”]

    StopTimer           # Stop the timer and revoke the “TIMER”
                        # condition.

    Send FrameABC
}
```

9.8 Revoke Condition Instruction

You can instruct the UWB*Trainer* to release all resources associated with some condition.

Especially do this if several conditions are not being used.

Format

ConditionRevoke(**condition_name**)

or

RevokeCondition(**condition_name**)

The instruction named parameter is:

Name: Condition name.

Example

```
Main
{
    # Register the RX Frame condition named "HDR_COND_1".
    RegRxFrmCondition ( HDR_COND_1 )
    {
        # Header Mask/Match
        PLCP
        {
            DestAddr = 0xAABB
            SrcAddr  = 0xBEEF
        }
    }

    # Do something.

    # Revoke condition from the UWBTrainer condition list.
    RevokeCondition ( HDR_COND_1 )
}
```

9.9 Reset Condition Instruction

You can reset the conditions registered as non auto-reset and change its state from "triggered" to "charged".

Format

ConditionReset(**condition_name**)

or

ResetCondition(**condition_name**)

The instruction named parameter is:

Name: Condition name.

Example

```
Main
{
    RegRxFrmCondition ( HDR_COND )
    {
        # Header Mask/Match
        PLCP
        {
            DestAddr = 0xAABB
            SrcAddr  = 0xBEEF
        }
    }

    Wait( HDR_COND ) # Wait for condition to be triggered.

    # Do something.

    ResetCondition( HDR_COND ) # Reset condition after it is triggered.

    Wait( HDR_COND ) # Again wait for condition to be triggered.
}
```

9.10 Wait Instructions

You can wait for registered conditions.

Formats

Wait for ANY one of the condition(s) specified:

```
Wait( condition_name_1 [, condition_name_2, ...] )
```

Wait for ANY one of the condition(s) specified (same as **Wait**):

```
Wait_Any( condition_name_1 [, condition_name_2, ...] )
```

Wait for ALL the condition(s) specified:

```
Wait_All( condition_name_1 [, condition_name_2, ...] )
```

Note: Wait / **Wait_Any** perform a logical **OR** on the conditions specified. **Wait_All** performs a logical **AND** on the conditions specified.

Examples

```
Main
{
    # Register a Header condition.
    RegRxFrmCondition( HDR_COND, YES )
    {
        # Header Mask/Match
        PLCP
        {
            DestAddr = 0xAABB
            SrcAddr  = 0xBEEF
        }
    }

    # Start the timer and register the auto_reset “TIMER” condition.
    StartTimer( 2000, YES )

    # Wait for condition 'HDR_COND' OR condition 'TIMER'
    # to be triggered.
    Wait( HDR_COND, TIMER )

    # Same as Wait but clearer for multiple conditions.
    Wait_Any( HDR_COND, TIMER )

    # Wait for condition 'HDR_COND' AND condition 'TIMER'
    # to be triggered.
    Wait_All( HDR_COND, TIMER )

}
```

9.11 Until Instructions

You can execute some code while waiting for some conditions.

Formats

Execute until ANY one of the condition(s) specified triggers:

```
Until( condition_name_1 [, condition_name_2, ...] )
```

Execute until ANY one of the condition(s) specified triggers (the same as **Until**):

```
Until_Any( condition_name_1 [, condition_name_2, ...] )
```

Execute until ALL condition(s) specified trigger:

```
Until_All( condition_name_1 [, condition_name_2, ...] )
```

Note: Until / **Until_Any** perform a logical **OR** on the conditions specified. **Until_All** performs a logical **AND** on the conditions specified.

Examples

```
Main
{
    # Register a Header condition.
    RegRxFrmCondition( HDR_COND, YES )
    {
        # Header Mask/Match
        PLCP
        {
            DestAddr = 0xAABB
            SrcAddr  = 0xBEEF
        }
    }

    # Start the timer and register the auto_reset “TIMER” condition.
    StartTimer( 2000, YES )


    # Send frames during 2000 microseconds.
    Until( TIMER )
    {
        # Send frames.
        Send WM_FRAME ( Delay = 1000 )
    }
```

```
        # Send frames until condition 'HDR_COND' OR condition 'TIMER'
        # triggers.
        # Same as Until but clearer for multiple conditions.
        Until_Any( HDR_COND, TIMER )
        {
            # Send frames.
            Send WM_FRAME ( Delay = 1000 )
        }

        # Send frames until condition 'HDR_COND' AND condition 'TIMER'
        # trigger.
        Until_All( HDR_COND, TIMER )
        {
            # Send frames.
            Send WM_FRAME ( Delay = 1000 )
        }

    }
```

9.12 Wait for the Next SuperFrame Instruction

This instruction instructs UWB*Trainer* to wait until the beginning of the next superframe(s) before processing the next instruction.

Note: This instruction does not delay script execution. It instructs the UWB*Trainer* transmitter to wait for the specified number of superframes before sending frames in its Tx queue. It means that the instruction following **WaitForNextSuperFrame** will be processed without any delay.

Format

WaitForNextSuperFrame([number_of_superframes])

The instruction named parameter is:

SFCount: Number of super frames to skip.

If this parameter is set to 0 (the default), UWB*Trainer* waits for the next superframe.

By default, **SFCount** is 0.

Example

```
Main
{
    # Wait for the next superframe.
    WaitForNextSuperFrame()

    # Send the frame 10 microseconds after the beginning of
    # the superframe.
    Send WM_FRAME( Delay = 10 )

    # Wait for the next superframe, then skip 3 superframes.
    WaitForNextSuperFrame( 3 )

    Send WM_FRAME
}
```

9.13 TxSleep Instruction

This instruction instructs the UWB*Trainer* transmitter to pause for a specified timeout before sending a next frame in its Tx queue.

Note: This instruction does not delay script execution as the **Sleep** instruction does. It means that the instruction following **TxSleep** will be processed without any delay. The TxSleep instruction is a more precise “sleep” timer than Sleep instruction but it affects only frame scheduling in Tx queue.

Format

TxSleep(**[** **delay_microseconds, delay_nanoseconds** **]**)

The instruction named parameter is:

Delay : Delay part in microseconds.
DelayNs: Delay part in nanoseconds.

Note: The total delay is calculated as **Delay** + **DelayNs**.

Example

```
Main
{
    # Send some frame
    Send WM_FRAME

    # Pause Tx queue for 10 microseconds.
    TxSleep( 10 )

    # Send the next frame.
    Send WM_FRAME

    # Pause Tx queue for 10 microseconds + 100 ns.
    TxSleep( 10,100 )

    # Send the next frame.
    Send WM_FRAME
 }
```

9.14 Wait Frame Shortcut Instructions

The **Wait** shortcut instructions save some time because they register conditions "behind the scenes".

Note: These instructions are not in Version 1.0.

Format

```
WaitRxFrame ( [timeout], [counter], [Time variable index] )
[
    {
        Mask/Match frame template or frame variable
        Additional condition parameters
    }
]
```

WaitRxFrame registers the **Rx Frame** and **Timer** conditions and waits until any of them triggers. The **Rx Frame** condition is triggered only if it occurs the number of times specified by the **counter** parameter.

Note: If no **Mask/Match** template is specified, the system waits for any Rx frame.

The instruction named parameters are:

Timeout: Specifies the timer timeout in microseconds.

Count: Specifies the number of times the **Rx Frame** condition event occurs before the condition is triggered.

TimeVar: Specifies the index of the **Time** variable in which to keep the condition timestamp when it is triggered. Values can range from 0 to 255. No value indicates that this parameter is ignored.

By default, **Timeout** = 0xFFFFFFFF, **Count** = 1, and **TimeVar** = UNDEFINED (timestamp is not saved).

For more details about **`Additional condition parameters`** please refer to topic 9.6.1.1 **Additional Rx Frame Header+Payload Condition parameters.**

Examples

```
Main
{
    # 1. Register a Rx Frame condition.
    # 2. Register a Timer condition (if specified).
    # 3. Wait for conditions.
    WaitRxFrame( 3000 )
    {
        RX_FRAME
        {
            # Header Mask/Match
            DestAddr = 0xAABB
            SrcAddr  = 0xBEEF
            # Payload Mask/Match
            Data     = {AA BB CC DD}
        }
    }
}
```

9.15 If Instructions

You can check whether some condition is hit or not.

Format

Test one OR several conditions:

```
If_Condition( condition_name_1 [, condition_name_2, ...] )
{
    <instructions>
}
[
else_condition
{
    <instructions>
}
]
```

Format

Test one OR several conditions (the same as **If_Condition**):

```
If_Any( condition_name_1 [, condition_name_2, ...] )
{
    <instructions>
}
[
else_condition
{
    <instructions>
}
]
```

Format

Test ALL conditions:

```
If_All( condition_name_1 [, condition_name_2, ...] )
{
    <instructions>
}
[
else_condition
{
    <instructions>
}
]
```

Examples

```
Main
{
    ConditionRegisterWMRxFrame( HDR_COND )
    {
        # Header Mask/Match
        PLCP
        {
            DestAddr = 0xAABB
            SrcAddr  = 0xBEEF
        }
    }

    # Wait for condition.
    Wait( HDR_COND, HDR_COND1, 2000 )

    # Check that ANY condition in the condition list is true.
    If_Condition( HDR_COND )
    {
        # Do something.
    }

    # Check that ANY condition in the condition list is true.
    # Same as If but clearer for multiple conditions
    If_Any( HDR_COND, HDR_COND )
    {
        # Do something.
    }

    # Check that ALL conditions in the condition list are true.
    If_All( HDR_COND, HDR_COND2 )
    {
        # Do something.
    }

    # Example with 'else' statement
    if_condition( HDR_COND )
    {
        # Do something.
    }
    else_condition
    {
        # Do something else.
    }
}
```

9.16 Loop Instruction

You can run some code in a loop, a limited or unlimited number of times.

Format

```
Loop [( counter )]
[
      {
          instruction_1
          ...
          instruction_n
      }
]
```

Note: If the **counter** parameter is omitted or set to **INFINITE,** the loop is executed infinitely. You can use **BreakLoop** instruction to break loop code execution and jump the next after Loop instruction.

Example

```
Main
{
    Loop( 100 ) # Run the instructions below 100 times.
    {
        Send TX_FRAME
        Send TX_FRAME
    }

# Run an infinite loop. User interaction is required to break it.
    Loop
    {
        Send TX_FRAME
        Send TX_FRAME
    }
}
```

9.17 BreakLoop Instruction

You can break **Loop** instruction code execution and jump to the next after **Loop** instruction.

Format

BreakLoop

Example

```
Main
{
    # Start 100 us timer
    StartTimer( 100 )

    # Run an infinite loop.
    Loop
    {
        # Break the loop after timer expires.
        If_Condition( TIMER ) { BreakLoop }

        Send TX_FRAME
    }

}
```

9.18 Exit Instruction

You can stop script execution at any time.

Format

```
Exit
```

Example

```
Main
{
    # Start the timer with timeout 10 seconds.
    StartTimer( 10 * 1000000 )

    # Run an infinite loop. User interaction is required to break it.
    Loop
    {
        Send TX_FRAME
        Send TX_FRAME

        If_Condition( TIMER )
        {
            # Stop script execution after sending frames for 10
            # seconds.
            Exit
        }
    }
}
```

9.19 Analyzer Control Instructions

You can control the UWB*Tracer*™ from the script.

9.19.1 StartRecording Instruction

You can instruct the application to start recording on UWB*Tracer*.

Format

```
StartRecording ([ recording_options, trace_file, keep_old_trace ] )
```

recording_options: Optional parameter specifying the path to the file containing the recording options. If it is omitted, the current UWB*Tracer* recording options are used.

trace_file: Optional parameter specifying a file name for the recorded trace. If it is omitted, the file name provided in the recording options is used.

keep_old_trace: Optional parameter specifying that the application should not overwrite a trace with the same file name. It should use a similar but different new file name. If it is omitted, the old trace file is overwritten.

The instruction named parameters are:

RecOpt: See **recording_options** parameter for details.

TraceName: See **trace_file** parameter for details.

KeepOldTrace: See **trace_file** parameter for details.

Example

```
Main
{
    StartRecording( "C:\\my_rec.rec", "C:\\UWB\Test1.uwb" )

    Send TX_FRAME
    Send TX_FRAME

    StartRecording( RecOpt = "C:\\my_rec.rec" )
}
```

9.19.2 StopRecording Instruction

You can instruct the application to stop recording on UWB*Tracer*.

Format

`StopRecording ( [ wait_for_trace, force_stop_recording ] )`

`wait_for_trace:` Optional parameter specifying that the UWB*Trainer* should not proceed until the application uploads the recorded trace. If it is omitted, the application stops recording and starts the uploading process, and UWB*Trainer* proceeds without waiting until the trace file is fully uploaded.

`force_stop_recording:` Optional parameter specifying that the application should stop recording and start the uploading process even if the recording was not started by the script. If it is omitted, the application stops recording and starts the uploading process only if the script started the recording.

The instruction named parameters are:

WaitForTrace: See **`wait_for_trace`** parameter for details**.**

ForceStopRec: See **`force_stop_recording`** parameter for details**.**

Examples

```
Main
{
    # Stop recording and wait for trace even if the script didn’t start
    # it.
    StopRecording( 1, 1 )


    StartRecording( "C:\\my_rec.rec", "C:\\Test1.uwb" )
    Send TX_FRAME
    Send TX_FRAME
    StopRecording( 1 ) # Stop recording and wait for trace.

    StartRecording( TraceName = "C:\\Test2.uwb" )
    Send TX_FRAME
    Send TX_FRAME
    StopRecording( 1 ) # Stop recording and wait for trace.
}
```

9.19.3 TriggerAnalyzer instruction

You can instruct the application to trigger UWB*Tracer*.

Format

`TriggerAnalyzer`

Example

```
Main
{
    StartRecording( "C:\\my_rec.rec", "C:\\Test1.uwb" )
    Send TX_FRAME
    Send TX_FRAME
    TriggerAnalyzer # Send a trigger signal to the analyzer.
    Send TX_FRAME
    Send TX_FRAME
}
```

9.19.4 Trace Instruction

You can instruct the application to display a trace message in the application output window.

Format

`Trace( trace_message )`

`trace_message:` Trace message to display.

Note: This instruction does not block script execution. If many trace messages are sent in a short period of time, some of them may be dropped and not displayed. Use the **Trace_B** instruction (see below) to guarantee trace message displaying.

Examples

```
Main
{
    Send TX_FRAME
    Send TX_FRAME

    Trace( "2 frames were sent." ) # Does not block script execution.

    Send TX_FRAME
    Send TX_FRAME

    Trace( "4 frames were sent." ) # Does not block script execution.
}
```

9.19.5 Trace_B Instruction

You can instruct the application to display a trace message in the application output window. The script will stop and will not proceed until the application displays the trace message .

Format

`Trace_B( trace_message )`

`trace_message:` Trace message to display.

Note: This instruction blocks script execution to guarantee trace message displaying. However, this can break precise timing sequences. Use the **Trace** instruction (see above) if you do not want to block script execution.

Examples

```
Main
{
    Send TX_FRAME
    Send TX_FRAME

    Trace_B( "2 frames were sent." ) # Blocks script execution.

    Send TX_FRAME
    Send TX_FRAME

    Trace_B( "4 frames were sent." ) # Blocks script execution.
}
```

10 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 struct variables in field assignments
- Using = + in field assignments
- Initializing struct 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

10.1 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 = 0xAABB   # Declare a local variable 'x'.
    Local y = 20 # Explicitly declare a local variable 'y'

    Send TX_FRAME
    {
      DestAddr = x      # Use the previously assigned variable 'x'
      SrcAddr  = y + 10 # Use the previously assigned variable 'x' + 10
      SeqCtrl  = z      # Create a new variable 'z' with value 0.
    }
}
```

10.2 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 Frame** 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 frame of type MY_TX_FRAME.
    $X = MY_TX_FRAME
# Note: Frame variables can be declared/redeclared and used many times.

    # Declare a variable Y and change the default frame.
    $Y = MY_TX_FRAME
    # Template field values.
    {
       DestAddr = 0xAABB
       SrcAddr  = 0xEFBE
    }

    # Declare a variable Z using the frame variable Y as a prototype.
    $Z = Y
    {
       DestAddr = 0x1122
       SrcAddr  = 0x3344
    }

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

    # Declare a structure variable having the same name as one of the
    # templates.
    $MY_TX_FRAME = MY_TX_FRAME
    {
       DestAddr = 0xAABB
       SrcAddr  = 0xEFBE
    }
```

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

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

10.3 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 frame 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 is 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
}
```

10.4 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 frame 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 will generate 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 { DestAddr = 0x1234 }

    # 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 assigment is not valid).
$Z = $Y

# Copy $Y to a newly instantiated $W structure variable.
# This is legal because $W does not have a frame 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 frame 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 frame 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 }
           }
}
```

10.5 Sending Frames using Structure Variables

You can send a frame based on some frame variable.

Examples

```
Main
{
# Send a frame based on the frame variable X using overridden values.
# Note: Frame variable X field values are NOT changed.
    # '$' may be omitted after the 'Send' keyword,
    # but if there is a template X, it is used instead of
    # the structure variable X.
    Send X
    {
        FrameType = 2
        DestAddr  = 0xAABB
        SrcAddr   = 0xEFBE
        Data      = { AA BB CC DD 12 34 56 78 [PATTERN_2] }
    }

    # '$' explicitly indicates that you want to send a frame
    # based on the structure variable X.
    Send $X
    {
        FrameCtrl = 0xC018 # Assign the whole multi-byte field.
        SeqCtrl   = 0x1234 # Assign the whole multi-byte field.
    }
    # Short syntax.
    # Send a frame based on the frame variable X having delay 50 us
    X ( 50 )
}
```

10.6 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_FRAME
    {
        SrcAddr = 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 DD 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
    }
}
```

10.7 Using Structure Variables to Assign Field Values

You can use structure variables in field assignments. Also See Section Using Multipliers to Assign Field Values.

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

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

10.8 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_FRAME
    {
        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_FRAME
    {
        # 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
    }
}
```

10.9 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

```
Frame 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 } }
        }
    }
    ... ... ... ...
}
```

10.10 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.

10.10.1 Assignments for Variables with Fixed-length Fields

For structure variables composed of fixed length fields (so that the 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

```
Frame 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
    }
}
```

10.10.2 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 come 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 will not be 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

```
Frame SF_Beacon
{
    PHY : 40
    MAC : 80
    BH  : 64
    IEs : *
    DATA: *
}
Frame 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
set SuperFramePeriod = 65538 # Set Length of SuperFrame (65,538 us).
```

```
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 }
    }
}
```

10.11 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_FRAME
    {
       # 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
    }
}
```

10.12 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_FRAME (Duration = (x+y)*7)
- TX_FRAME (x + 12)

Examples

```
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_FRAME ( Delay = x )

    Send TX_FRAME
    {
        SrcAddr =  0xEEEE

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

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

10.13 Preprocessor If Operator

A special preprocessor 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_FRAME
        Send MMC_FRAME( 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 WINET_FRAME
        {
            DestAddr  = 0xAABB
            SrcAddr   = 0xEFBE
            Data      = { AA BB CC DD 12 34 56 78 [PATTERN_2] }
        }
    ( Delay = 100000 )
    }
    # 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_WUSB_FRAME
        Send MMC_FRAME( Delay = 2000 )
    }
    else
    {
    # This block is parsed in a special way.
    # Only "{" "}" are taken into account.
    # All other stuff is ignored.
        Send MY_WUSB_FRAME
    }
}
```

10.14 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_FRAME
        {
            SrcAddr  = 0xEEEE
            DestAddr = 0xBBBB
            # 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_FRAME
        {
            SrcAddr  = 0xEEEE
            DestAddr = 0xBBBB
            # Data pattern uses preprocessor variables.
            Data     = { k k k k k }
       }
    }
}
```

10.15 Forward Declarations

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

Example

```
Main
{
    # The frame 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' frame template used above.
    Frame SOME_PKT_TEMPLATE
    {
        Field0 : 32 = 0xFEFEFEFE
    }
}

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

10.16 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 (for the **RandSeed** setting description, see the "Generation Setting" topic).

Example

```
Main
{
    # The frame 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.
}
```

10.17 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 (preceding).

Example

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

10.18 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_FRAME'.
TX_FRAME $Z

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

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

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

Main
{
    Send $Z  # Send a frame 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.

10.19 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.

10.19.1 Calling Another 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 frame template.
Frame SOME_PKT_TEMPLATE
{
    Field0 : 32 = 0xFEFEFEFE
}

# Declare a generation procedure with two numeric parameters.
Block1( x, y )
{
# Send a frame 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 frame based on the structure variable parameter
    # to 'pkt_param'.
    Send pkt_param
    {
        Field0 =  x + y
    }
}

# Call another generation procedure.
Main
{
    Send TX_FRAME # Send frame based on the template 'TX_FRAME'.

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

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

    # Payload
    Data      : *
    CRC       : 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 frame 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 frame based on 'SOME_PACKET' template
                         # and Data = { BE EF DA CD BE EF DA CD }

}
```

10.19.2 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 frame 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()
}
```

10.19.3 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 frame 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 frame 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 frame based on the local structure variable 'p'.
    Send Z  # Send frame based on the global structure variable 'Z'.
}
```

10.20 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 UWB*Trainer* 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.

10.20.1 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 WM_FRAME
54:     {
55:         Data = {AA BB CC DD}
56:     }
57:     PTrace ("Sending Second WM_FRAME")
58:     Send WM_FRAME
59:     {
60:         Data = {EE FF 00 11}
61:     }
62:     PTrace ()
63: }
```

Output Window:

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

10.20.2 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 will generate parse errors.

Note: Numeric variables are DWORDS.

Example

```
51: frame 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.uwbg. Operation Starts at time 09:29:17 AM -
Parsing...  c:\sample_ptracevar.uwbg
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.uwbg - 0 error(s)
C:\Sample_PTraceVar.uwbg - Generation Blocks : 1, Total number of instructions : 2
- Parsing Finished: File: C:\Sample_PTraceVar.uwbg. Operation ends at time 09:29:17 AM -
```

10.20.3 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 will 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: Frame Test_IE_ChSelect : IE_INFO  # Create Frame Test_IE_ChSelect.
57: {
58:     ElemID  = 251
59:     ElemLen = 2
60:     TModeSubType : 8 = 1
61:     PHYChNum     : 8
62: }
63: Frame BPOIE : IE_INFO            # Create Frame 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.uwbg. Operation Starts at time 05:50:10 PM
Parsing...  c:\sample_ptracevarex.uwbg
$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.uwbg - 0 error(s)
C:\Sample_PTraceVarEx.uwbg - Generation Blocks : 1, Total number of instructions : 4
- Parsing Finished: File: C:\Sample_PTraceVarEx.uwbg. Operation ends at time 05:50:10 PM
```

10.20.4 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, frame, and packet names, which will generate parse errors.

Example

```
50: Main
51: {
52:     PTraceTemplate ( IE_INFO_ALLOCATION, BEACON_PARAMS )
53:     Send WM_FRAME { DATA = { 01 08 0C 00 05 00 04 00 05 00 } }
54:     PTraceTemplate ( RX_HDR_ERRORS )
55: }
```

Output Window:

```
Parsing started:File: C:\Sample_PTraceTemplate.uwbg. Operation Starts at time 02:36:26 PM
Parsing...
c:\sample_ptracetemplate.uwbg
------------------------------------ ← PTraceTemplate ( IE_INFO_ALLOCATION, … ) line 52
Template : IE_INFO_ALLOCATION          ←
Template group : Generic               ←

Fields:                                ←
Field : ZONEBITMAP                     ←
    index = 0, offset  = 0, length = 16  ←
Field : MASBITMAP                      ←
    index = 1, offset  = 16, length = 16 ←
------------------------------------   ←

------------------------------------ ← PTraceTemplate ( …, BEACON_PARAMS ) line 52
Template : BEACON_PARAMS               ←
Template group : Generic               ←

Fields:                                ←
Field : MACADDRLOWER                   ←
    index = 0, offset  = 0, length = 16  ←
Field : MACADDRUPPER                   ←
    index = 1, offset  = 16, length = 32 ←
Field : SLOTNUMBER                     ←
    index = 2, offset  = 48, length = 8  ←
Field : SECURITYMODE                   ←
    index = 3, offset  = 56, length = 2  ←
Field : RESERVED                       ←
    index = 4, offset  = 58, length = 4  ←
Field : SIGNALINGSLOT                  ←
    index = 5, offset  = 62, length = 1  ←
Field : MOVABLE                        ←
    index = 6, offset  = 63, length = 1  ←
--------------------------------       ←
```

```
-----------------------------------   ← PTraceTemplate ( RX_HDR_ERRORS ) line 54
Template : RX_HDR_ERRORS               ←
Template group : Generic               ←

Fields:                                ←
Field : HDERSVD                        ←
    index = 0, offset = 0, length = 3  ←
Field : HDERROR                        ←
    index = 1, offset = 3, length = 5  ←
-------------------------------        ←

Parsing is done...
C:\Sample_PTraceTemplate.uwbg - 0 error(s)
C:\Sample_PTraceTemplate.uwbg - Generation Blocks : 1, Total number of instructions : 1
Parsing Finished: File: C:\Sample_PTraceTemplate.uwbg. Operation ends at time 02:36:26 PM
```

10.21 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 TxPower  = UwbTxPower
%alias RegTimer = ConditionRegisterTimer
%alias WM_FRAME = WIMEDIA_LONG_TEMPLATE_NAME
Set TxPower = 8  # Use an alias for the 'UwbTxPower' setting.
Frame WIMEDIA_LONG_TEMPLATE_NAME
{
    Data : *
}
Main
{
    # Use an alias for the 'ConditionRegisterTimer' instruction.
    # Register the auto-reset timer condition named 'TIMER_COND'.
    RegTimer ( TIMER_COND, 2000 )
    Send WIMEDIA_LONG_TEMPLATE_NAME
    {
        Data = {AA BB CC DD}
    }
    # Use an alias for the 'WIMEDIA_LONG_TEMPLATE_NAME' template.
    # Send a frame based on the 'WIMEDIA_LONG_TEMPLATE_NAME' template.
    # Using aliases for frame/structure templates allows you to
    # use short names instead of the long names defined in
    # large template libraries.
    Send WM_FRAME
    {
        Data = {AA BB CC DD}
    }
}
```

10.22 Include Path Directive

This feature allows you to specify additional folders where the parser should look for included files. By default, the script parser tries to include a file using a name specified in the **%include** directive. If it can't 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:\UwbGeneration\"

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

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

Main
{
    # Do something.
}
```

11 Appendix A – Generation Script Example

```
#*********************************************************
#                WUSBMassSCSIInquiry.uwbg                 #
#*********************************************************
# This file contains a generation script which generates
# WUSB Transfer that transfers Mass Storage class SCSI Inquiry
# command over Wireless USB.
#*********************************************************

# Include main definitions.
# Some WUSB templates are defined in this include file.
%include "Generation\Include\UwbGeneration.ginc"


#================================================#
#                 Constants                      #
#================================================#

#================================================#
#                 Data Patterns                  #
#================================================#


DataPattern SetAddrReq   = { 00 05 00 00 00 00 00 00 }

DataPattern INQUIRY_CDB  = { 55 53 42 43 C8 81 4F 81 24 00 00 00 80 00 06 12
                             00 00 00 24 00 00 00 00 00 00 00 00 00 00 00 00 }

DataPattern INQUIRY_RESP = { 00 80 00 01 75 00 00 00 4D 2D 53 79 73 20 20 20
                             44 69 73 6B 4F 6E 4B 65 79 20 20 20 20 20 20 20
                             32 2E 30 31 }

#================================================#
#       Frame and structure templates            #
#================================================#
struct DeliveryID
{
    Sel : 1 = 1 # Stream Index
    Val : 3
}
```

```
#================================================#
#           Main Generation Procedure            #
#================================================#
Main
{
    # parser numeric variables
    dest_address = 0x0080
    host_address = 0xBEEF
    endpoint     = 4
    direction    = IN

    # parser for loop
    for( i = 0, i < 2, i++ )
    {

        # parser if directive
        if( i == 1 )
        {
            host_address = 0xABCD
            endpoint = 5
        }

        # parser Call of another generation procedure
        Call Inquiry( host_address, dest_address, endpoint, direction )
    }

}
```

```
#=====================================================#
#   Second generation procedure 'Inquiry'             #
#=====================================================#
# The four parameters are host_addr, dest_addr, endpoint, and direction.
# More parameters could be added here.
#=====================================================#
Inquiry( host_addr, dest_address, endpoint, direction )
{
    # Prepare payload for MMC frame Request data.
    $req = USB_REQ  # structure variable based on template 'USB_REQ'
    {
        Request = SET_CONFIG
        Value   = 1
    }

    # Create WDRCTA structure.
    $dr_cta   = WDRCTA_SETUP
    {
        Start = 154
        DevID = 128
        SData = $req
    }
    $dn_cta   = WDNTSCTA { Start = 215  NumSlots = 16 }
    $dt_cta   = WDTCTA   { Start = 599 }
    $eol_cta  = WCTA_EOL

    # Create CTA.
    $cta_ie   = WUSB_IE
    {
        IE_ID = WCTA_IE

        # Complex length calculation done during preprocessor stage
        Length = ( fld_size( Length  ) + fld_size( IE_ID   ) +
                   pkt_size( $dr_cta ) + pkt_size( $dn_cta ) +
                   pkt_size( $dt_cta ) + pkt_size( $eol_cta ) ) / 8   # = 34

        # Build complex payload from stuctures.
        Data  = $dr_cta + $dn_cta + $dt_cta + $eol_cta
    }

    $ca_block = CA_BLOCK
    $ca_ie    = WUSB_IE
    {
        IE_ID  = WCONNECTACK_IE
        Length = ( fld_size( Length ) + fld_size( IE_ID ) +
                   pkt_size( $ca_block ) ) / 8   # = 20
        Data   = $ca_block
    }

    $host_info_ie = WHOSTINFO_IE
    {
        ConnAvail = CA_LIMITED
        StreamIdx = 2
        CHID      = { ED B4 24 3F 68 23 CC BD 5D 44 57 27 6B 4B FC 85 }
    }
```

```
##############################################################################
    # Send MMC frame from Host to Device.
    Send WUSB_MMC
    {
        DestAddr     = 0x00FE
        SrcAddr      = host_addr  # Use a generation procedure parameter.
        NextMMCTime  = 3
        WUSBTimeStmp = 25

        # Combined payload containing several IEs
        Data = $cta_ie + $ca_ie + $host_info_ie
    }

    # Send Data OUT frame from Host to Device.
    Send WUSB_DATA
    {
        MoreFrms = 1

        DestAddr = dest_address  # Use a generation procedure parameter.
        SrcAddr  = host_addr     # Use a generation procedure parameter.
        Data     = { 00 01 }
    }

    $dlvry_id = DeliveryID { Val = 3 }

    # Wait for Data HNDSHK frame from Device to Host with ACK code.
    WaitRxFrame()
    {
        WUSB_HNDSHK_RX
        {
            CtrlType = $dlvry_id

            DestAddr = host_addr    # Use a generation procedure parameter.
            SrcAddr  = dest_address # Use a generation procedure parameter.
            Flags    = HNC_ACK
        }
    }
```

```
##############################################################################
# OUT TRANSFER
##############################################################################
    $dr_cta  = WDRCTA  # Create a WDRCTA structure.
    {
        Start = 154
        DevID = 128
        EndP  = endpoint
    }

    $cta_ie = WUSB_IE
    {
        IE_ID  = WCTA_IE
        Length = ( fld_size( Length  ) + fld_size( IE_ID   ) +
                   pkt_size( $dr_cta ) + pkt_size( $dn_cta ) +
                   pkt_size( $dt_cta ) + pkt_size( $eol_cta ) ) / 8   # = 34
        Data   = $dr_cta + $dn_cta + $dt_cta + $eol_cta
    }

    # Send MMC frame from Host to Device.
    Send WUSB_MMC
    {
        DestAddr     = 0x00FE
        SrcAddr      = host_addr

        NextMMCTime  = 3
        WUSBTimeStmp = 25

        # Combined payload containing several IEs
        Data = $cta_ie + $ca_ie + $host_info_ie
     }

    # Send Data OUT frame from Host to Device.
    Send WUSB_DATA
    {
        MoreFrms = 1

        DestAddr = dest_address
        SrcAddr  = host_addr
        EndP     = endpoint
        EDir     = direction

        Data     = INQUIRY_CDB  # SCSI Inquiry CDB
    }
```

```
    # Wait for Data HNDSHK frame From Device to Host with ACK code.
    WaitRxFrame()
    {
        WUSB_HNDSHK
        {
            EndP     = endpoint
            EDir     = direction
            CtrlType = $dlvry_id
            DestAddr = host_addr
            SrcAddr  = dest_address
            AckCode  = 2
        }
    }

################################################################################
# IN TRANSFER
################################################################################
    $dt_cta  = WDTCTA  # Create a WDTCTA structure.
    {
        Direction = IN
        Start     = 154
        DevID     = 128
        EndP      = endpoint
        DINAck    = 1  # You need the first segment.
    }

    $cta_ie = WUSB_IE
    {
        IE_ID  = WCTA_IE
        Length = ( fld_size( Length  ) + fld_size( IE_ID   ) +
                   pkt_size( $dt_cta ) + pkt_size( $dn_cta ) +
                   pkt_size( $eol_cta ) ) / 8

        Data   = $dt_cta + $dn_cta + $eol_cta
    }

    # Send MMC frame from Host to Device.
    Send WUSB_MMC
    {
        DestAddr = 0x00FE
        SrcAddr  = host_addr

        NextMMCTime  = 3
        WUSBTimeStmp = 25

        Data = $cta_ie + $ca_ie + $host_info_ie # combined payload containing
                                                # several IEs.
    }
```

```
    # Wait for Data IN frame from Device to Host: SCSI Inquiry Response.
    WaitRxFrame()
    {
        WUSB_DATA_TX
        {
            MoreFrms = 1
            DestAddr = host_addr
            SrcAddr  = dest_address
            EndP     = endpoint
            Data     = INQUIRY_RESP
        }
    }

    $blank_dt_cta  = WDTCTA # Create a blank WDTCTA structure confirming that
                            # you received the segment #0.
    {
        DevID     = 128
        Direction = IN
        EndP      = 4

        DINAck    = 2  # 2 (bit 1 is set) means that you received the segment
                       # 0 and are ready for the segment #1, if any.
    }

    $cta_ie = WUSB_IE
    {
        IE_ID  = WCTA_IE
        Length = ( fld_size( Length ) + fld_size( IE_ID ) +
                   pkt_size( $blank_dt_cta ) +
                   pkt_size( $eol_cta ) ) / 8

        Data   = $blank_dt_cta + $eol_cta
    }

    # Send Handshake MMC frame from Host to Device confirming that you
    # received all the data.
    Send WUSB_MMC
    {
        DestAddr = 0x00FE
        SrcAddr  = host_addr # Use a generation procedure parameter as a
                             # local variable.

        NextMMCTime  = 3
        WUSBTimeStmp = 25

        Data = $cta_ie + $ca_ie + $host_info_ie # Combined payload containing
                                                # several IEs
    }

}
```

12 How to Contact LeCroy

Type of Service	**Contact**
Call for technical support…	US and Canada: 1 (800) 909-2282 Worldwide: 1 (408) 653-1260
Fax your questions…	Worldwide: 1 (408) 727-6622
Write a letter …	LeCroy Protocol Solutions Group Customer Support 3385 Scott Blvd. Santa Clara, CA 95054-3115
Send e-mail…	psgsupport@lecroy.com
Visit LeCroy's website…	http://www.lecroy.com/