jPDL2 - Protocol Definition Language Version 2

Table of contents

1. Introduction

The Protocol Definition Language Version 2 (PDL2) and its implementation, jPDL2, is used to decode and encode messages to and from any binary protocol that can be defined in PDL2. The library was initially written to handle the MIDI protocol used by the synthesizer Clavia Nord Modular but is now a general purpose library for defining binary protocols and parsing or generating streams according to the protocol.

The PDL2 library mainly consists of a parser which reads a protocol definition and generates an equivalent document object model (DOM), the DOM implementation, streaming class for bitwise reading or writing, classes for parsing and generating streams according to the protocol.

2. History

- **2008-02-22:** PDL Version 2

3. References

- JPDL2 API

4. Using the PDL2 library

4.1. Parsing a PDL-file

```
import net.sf.nmedit.jpdl2.format.PDL2Parser;
import net.sf.nmedit.jpdl2.dom.PDLDocument;
import net.sf.nmedit.jpdl2.PDLPacketParser;
...

String PDLFileContents =
  "start Sysex; Sysex := 0xF0 MidiData$data 0xF7; " +
  ...; // pdl file contents
PDL2Parser fileParser =
  new PDL2Parser(new StringReader(PDLFileContents)); // create the PDL
parser
fileParser.parse(); // parse the file
PDLDocument pdlDoc =
  fileParser.getDocument(); // get the parsed document
...
```

4.2. Parsing a message

```
// parse a message
BitStream messageStream = getMessageStream(); // get message stream
PDLPacketParser packetParser = new PDLPacketParser(pdlDoc); // create
packet parser
PDLMessage message = packetParser.parseMessage(messageStream); // parse
message stream

// further processing
handle(message);
```

4.3. Generating a message

```
// generate a message
IntStream messageData = getMessageData(); // get message data
PDLPacketParser packetParser = new PDLPacketParser(pdlDoc); // create
packet parser
PDLMessage message = packetParser.parseMessage(messageData); // parse
message stream
BitStream resultStream = packetParser.getBitStream(); // get generated
message stream

// further processing
byte[] byteMessage = resultStream.toByteArray();
handle(byteMessage);
```

5. The Packet Parser

5.1. Parsing Methods

Return-Type	Method	Requires start-Statement
PDLMessage	parseMessage(PDLDataSource input)	yes
PDLMessage	parseMessage(PDLDataSource input, String packetName)	no
PDLPacket	parse(PDLDataSource input)	yes
PDLPacket	parse(PDLDataSource input, String packetName)	no

The packet parser takes instances of IntStream or BitStream as argument (PDLDataSource). For IntStream arguments it generates a BitStream instance containing the message. Parsing starts at the packet with the specified name. If no packet name is specified, parsing starts at the packet referenced in the start-Statement.

5.2. Table of Modifications

Statement	Example	IntStream read	BitStream read	BitStream generate	Packet
Constant	0xF0:8	1 value	read specified number of bits and compare to constant	write constant value with specified number of bits	-
Constant (multiplicity)	n*0x1:8	n values	repeat n-times: read specified number of bits and compare to constant	repeat n-times: write constant value with specified number of bits	-
Variable	v:8	1 value	read specified number of bits	write value with specified number of bits	set variable
Variable-List	n*v:8/0	at most n values, stop behind a terminal value	read at most n-times the specified number of bits, stop behind a terminal value	write at most n-times the specified number of bits, stop behind a terminal value	set variable-list (terminal not included)
Implicit Variable	v:8 = (a+b)	-	read specified number of bits, fails if the function result and the value are not equal	compute function, write the result value with specified number of bits	set variable
Anonymous Variable	%v:8 = (a+b)	-	-	-	compute function, set variable
Packet Reference	Packet$data	-	-	-	parse into new packet/set packet
Inline Packet Reference	Packet$$	-	-	-	parse into current packet

Packet-List	n*Packet$data	-	-	-	parse packet n-times/set packet-list
stringdef	str:="hello"	-	-	-	set string
					Message (Global)
Label	@lblName	-	-	-	store bitstream position
messageId	messageId ("Sysex")	-	-	-	set message id

Note: message (global) properties are mutable and store only the latest value.

5.3. Parsing Behaviour

Statement	Example	Behaviour	cause of failing
if-Statement	if (v>8) { ... }	if the condition is true, parse the following item/block	if the condition is true and the following item/block fails

switch-Statement	switch(v) {
	case 1: ...
	case 2: ...
	...
	default:...
	}
		Selects a case according to the value. Either selects a specific case or if no case matches the value and the default case is present then selects the default case. If the default case is not present, then no case may be selected.	the selected case fails

optional-Statement	?Optional$$	Parses the optional statement, ignores the statement if it failed.	-
choice-Statement	(A$$\|B$$|C$$)	Parses the choosable statements in the specified order. Accepts the first statement which could	each of the choosable statements failed

		be parsed successfully.	
fail-Statement	if(v==2) fail	always fails (in the example the packet fails if the condition v==2 is true)	always

6. PDL2 Syntax

6.1. Comments

```
... // comment until the end of the line
... /* comment over
       multiple
       lines */
...
```

6.2. start-Statement

The PDL File may begin with the optional 'start'-statement. The start-Statement identifies a packet as the packet with that the parser should begin parsing / generating a message.

Syntax:

```
START_DECLARATION := 'start' PACKET_NAME ';'
```

Example:

```
start Sysex;
```

6.3. Packet Declaration

The remaining PDL file consists of a sequence of packet-statements.

```
PACKET_DECL_LIST :=
    PACKET_DECLARATION
  | PACKET_DECLARATION PACKET_DECL_LIST
;
```

Syntax

```
PACKET_DECLARATION :=
  PACKET_DECL_HEADER ':=' ITEM_LIST_OR_NO_ITEMS ';'
;
PACKET_DECL_HEADER :=
    PACKET_NAME
  | PACKET_NAME '%' PADDING
```

```
;
ITEM_LIST_OR_NO_ITEMS :=
    /* no items */
  | ITEM_LIST
;
ITEM_LIST :=
    ITEM
  | ITEM ITEM_LIST
;
ITEM :=
    CONSTANT
  | VARIABLE
  | VARIABLE_LIST
  | IMPLICIT_VARIABLE
  | ANONYMOUS_VARIABLE
  | PACKET_REF
  | INLINE_PACKET_REF
  | PACKET_LIST
  | LABEL
  | MESSAGE_ID
  | FAIL
  | STRINGDEF
  | IF_STATEMENT
  | SWITCH_STATEMENT
  | CHOICE_STATEMENT
  | OPTIONAL_STATEMENT
  | BLOCK
;
BLOCK := '{' ITEM_LIST_OR_NO_ITEMS '}' ;
```

Example

```
start Sysex;

Sysex       := ... ;
Packet2     := ... ;
Packet3 % 8 := ... ;
...
PacketN % 9 := ... ;
```

Packet declarations might have an optional padding value. If the value is not specified, then a default padding value of 1 is used.

A packet statement is either empty or contains one ore more of the following items: choice-Statement, Constant, fail, if-Statement, Label, messageId, optional-Statement, Packet Reference, Inline Packet Reference, Packet-List, switch-Statement, Variable, Variable-List, Implicit Variable, Anonymous Variable, StringDef.

6.4. Constant

A constant value or a list of constant values.

Syntax:

```
CONSTANT :=
  MULTIPLICITY? CONSTANT_VALUE ':' SIZE
;
CONSTANT_VALUE :=
  INTEGER_LITERAL
;
SIZE :=
  INTEGER_LITERAL        // 0 <= SIZE < 32
;
INTEGER_LITERAL :=       // 32-bit integer
    DUAL
  | HEXADECIMAL
  | DECIMAL
;
DECIMAL :=
  0 | [1-9][0-9]*        // example: 1234
;
HEXADECIMAL :=
  "0x" [0-9a-fA-F] {1,8} // example: 0xF0
;
DUAL :=
  [01] {1,32} [dD]       // example: 001d
;
```

Example:

```
Sysex := 0xF0:8 ... 0xF7:8 ;
```

The Sysex packet starts with the 8-bit number 0xF0 and ends with the 8-bit number 0xF7.

6.5. Variable

A variable defines a value which can be queried after parsing or can be used in an integer expression in the PDL file.

Syntax:

```
VARIABLE :=
  VARIABLE_NAME ':' SIZE
;
VARIABLE_NAME :=
  NAME
;
```

Example:

```
Sysex := 0xF0:8 v:8 ... 0xF7:8 ;
```

Defines the 8-bit variable 'v'.

6.6. Variable-List

A variable-list defines a list of values of the same size which can be queried after parsing.

Syntax:

```
VARIABLE_LIST :=
    MULTIPLICITY VARIABLE
  | MULTIPLICITY VARIABLE '/' TERMINAL
;
TERMINAL :=
  INTEGER_LITERAL
;
```

Example:

```
Sysex := 0xF0:8 16*string1:8 16*string2:8/0 ... 0xF7:8 ;
```

Defines 16*8-bit variablelist 'string1' and the 16*8-bit variable list string2 which additional allows the terminal symbol '0'.

6.7. Implicit Variable

As the name indicates the value of this variable is implied. The value must be equal to the value of the specified integer expression. The implicit variable is intendet to be used to for checksum values.

Syntax:

```
IMPLICIT_VARIABLE :=
  VARIABLE '=' '(' INTEGER_EXPRESSION ')'
;
```

Example:

```
Sysex := 0xF0:8 ... 0:1 v:7=(2*3)  0xF7:8 ;
```

Variable with an integer expression (function) assigned.

6.8. Anonymous Variable

A anonymous variable is not part of the stream but only added to the packet.

Syntax:

```
ANONYMOUS_VARIABLE :=
  '%' IMPLICIT_VARIABLE
;
```

Example:

```
Sysex := 0xF0:8 ... a:1 b:1 %ab:2=((a<<1) | b)  0xF7:8 ;
```

6.9. Multiplicity

Syntax:

```
MULTIPLICITY :=
  VARIABLE_NAME '*'
| CONSTANT_VALUE '*'
;
```

Example:

```
Sysex := 2*0:8 2*var_list:8 v:8 v*0:8 v*var_list2:8 ;
```

6.10. Label

A label stores the current bit-position. The label may be updated and stores only the latest bit-position.

Syntax:

```
LABEL := '@' LABEL_NAME ;
```

Example:

```
Sysex := 0xF0:8 ... @lblEnd 0:1 v:7=(@lblEnd&0x7F)  0xF7:8 ;
```

6.11. Packet Reference

A reference to another packet declaration.

Syntax:

```
PACKET_REF := PACKET_NAME '$' BINDING ;
```

Example:

```
Sysex := 0xF0:8 DataPacket$data 0xF7:8 ;
DataPacket := ... ;
```

The example shows the reference of the packet 'DataPacket'. The reference is associated with the packet binding value 'data'.

6.12. Inline Packet Reference

A reference to another packet declaration. The items of the referenced parsed as if they were declared in the referencing packet declation. No new data-packet will be created. The declaration allows to reuse certain parts of the protocol without changing the data-packet structure.

Syntax:

```
INLINE_PACKET_REF := PACKET_NAME '$$';
```

Example:

```
Sysex := 0xF0:8 Inline$$ 0xF7:8 ; Inline := d:8 ;
is the same as
Sysex := 0xF0:8 d:8 0xF7:8 ;
```

6.13. Packet-List

Like with constants or variables it is possible to define a list of packets.

Syntax:

```
PACKET_LIST := MULTIPLICITY PACKET_REF;
```

Example:

```
Sysex := 0xF0:8 v:8 v*DataPacket$data 0xF7:8 ;
DataPacket := ... ;
```

The example shows the reference of the packet 'DataPacket'. 'DataPacket' is repeated 'v'-times. The packet-list is then associated with the packet binding value 'data'.

6.14. if-Statement

Syntax:

```
IF_STATEMENT := 'if' '(' BOOLEAN EXPRESSION ')' ITEM ;
```

Example:

```
Packet := v:8 if (v >= 2) { v2:8 } ;
```

6.15. switch-Statement

Syntax:

```
SWITCH_STATEMENT :=
  'switch' '(' INTEGER EXPRESSION ')'
  '{' SWITCH_CASE_LIST DEFAULT_CASE? '}'
;
SWITCH_CASE_LIST :=
    CASE_STATEMENT
  | CASE_STATEMENT SWITCH_CASE_LIST
;
CASE_STATEMENT :=
  INTEGER_LITERAL ':' ITEM
;
DEFAULT_CASE :=
  'default' ':' ITEM
;
```

Example:

6.16. fail

Syntax:

```
FAIL_STATEMENT := 'fail' ;
```

Example:

```
Packet := v:8
  switch (v)
  {
    case 0x01 : 0:8
    case 0x02 : v:8
    default   : fail
  }
;
```

6.17. stringdef

Syntax:

```
STRINGDEF := STRING_NAME ':=' STRING_VALUE;
STRING_NAME := STRING_LITERAL;
STRING_VALUE := STRING_LITERAL;
```

Example:

```
Packet := manufacturer_id:8
  if (manufacturer_id == 0x33) manufacturer := "Clavia Digital Instruments"
;
```

6.18. choice-Statement

Syntax:

```
CHOICE_STATEMENT := '(' CHOICE_LIST ')' ;
CHOICE_LIST :=
    ITEM
  | ITEM '|' CHOICE_LIST
;
```

Example:

```
Packet := ( { 1:8 v:8 } | A$a | B$b ) ;
A := a1:8 a2:8 a3:8;
B := 0:8 b1:8 b2:8 b3:8;
```

6.19. optional-Statement

Syntax:

```
OPTIONAL_STATEMENT := '?' ITEM ;
```

Example:

```
Packet := v:8 ?OptionalPacket$p;
OptionalPacket := v:8 if (v>4) fail;
```

6.20. messageId

Syntax:

```
MESSAGE_ID := 'messageId' '(' STRING_LITERAL ')';
```

Example:

```
Packet := ... id:2 messageId('MessageD')
  switch(id)
  {
    case 0: { messageId('MessageA') packetA$data }
    case 1: { messageId('MessageB') packetB$data }
    case 2: { messageId('MessageC') v:4 }
    case 3: { } // 'MessageD'
  }
;
```

Associates a message with an identifier (ID). Each time a messageId-Statement is found, the global messageId value is set to the specified value.

6.21. Expression

Syntax:

```
EXPRESSION :=
    INTEGER_LITERAL
  | VARIABLE_NAME
  | LABEL
  | '-' EXPRESSION
  | '~' EXPRESSION
  | '!' EXPRESSION
  | '(int)' EXPRESSION
  | '(boolean)' EXPRESSION
  | EXPRESSION '+'   EXPRESSION
  | EXPRESSION '-'   EXPRESSION
  | EXPRESSION '*'   EXPRESSION
  | EXPRESSION '/'   EXPRESSION
  | EXPRESSION '%'   EXPRESSION
  | EXPRESSION '&'   EXPRESSION
  | EXPRESSION '|'   EXPRESSION
  | EXPRESSION '^'   EXPRESSION
  | EXPRESSION '<<'  EXPRESSION
  | EXPRESSION '>>'  EXPRESSION
  | EXPRESSION '>>>' EXPRESSION
  | EXPRESSION '=='  EXPRESSION
  | EXPRESSION '!='  EXPRESSION
  | EXPRESSION '<'   EXPRESSION
  | EXPRESSION '<='  EXPRESSION
  | EXPRESSION '>'   EXPRESSION
  | EXPRESSION '>='  EXPRESSION
  | '(' EXPRESSION ')'
  | '$'
  | STREAM_OPERATOR
;
INTEGER_EXPRESSION :=
  EXPRESSION
;
BOOLEAN_EXPRESSION :=
  EXPRESSION
;
STREAM_OPERATOR :=
  '[' ST_OPERATOR ';' ST_START ';' ST_END ';' ST_SIZE ';' ST_FIELD ']'
;
ST_OPERATOR :=
    '+'
  | '|'
  | '&'
  | '^'
  | '*'
;
ST_START := INTEGER_EXPRESSION ;
ST_END   := INTEGER_EXPRESSION ;
ST_SIZE  := INTEGER_EXPRESSION ;
ST_FIELD := INTEGER_EXPRESSION ; // only here the '$' placeholder is
allowed
```

6.21.1. Operators

Operators in order of precedence from highest to lowest

Precedence	Operators	Operation	Associativity
1	-	unary minus	right
	~	bitwise NOT	
	!	boolean (logical) NOT	
	(int), (boolean)	type cast	
2	* / %	multiplication, division, remainder	left
3	+ -	addition, substraction	left
4	<<	signed bitshift left	left
	>>	signed bitshift right	
	>>>	unsigned bitshift right	

5	< <=	less than, less than or equal to	left
	> >=	greater than, greater than or equal to	
6	==	equal to	left
	!=	not equal to	
7	&	bitwise AND	left
	&	boolean (logical) AND	
8	^	bitwise XOR	left
	^	boolean (logical) XOR	
9	|	bitwise OR	left
	|	boolean (logical) OR	

6.21.2. Stream Operator

The PDL2 format contains a special stream operator which is intended to be used to compute checksum values.

The operator is a function

```
(int, int, int, function:(int) # int) # int
f(start_bit_position, end_bit_position, size, function:(int) # int) # int
```

The operator can be translated to following equivalent java code.

```
function streamOperator(
  BitStream bitstream;   // the message
  char operator;         // the operator, one of [+-*^]
  IntExpression e_start; IntExpression e_end; IntExpression e_size;
IntExpression e_field
)
{
  int start = e_start.computeInt();
  int end   = e_end  .computeInt();
  if (start >= end) error();

  int size  = e_size .computeInt();
  if (size <= 0) error();

  int result = 0;
  for ( int pos = start ; pos<end ; pos+=size )
  {
    bitstream.setPosition(pos);
    int field = bitstream.getInt(size);
    field = e_field.computeInt(field); // '$' refers to this field argument

    if (pos == start) { result = field; continue; }

    switch (operator)
    {
      case '+': { result = result + field; break; }
      case '*': { result = result * field; break; }
      case '^': { result = result ^ field; break; }
      case '|': { result = result | field; break; }
    }
  }
  return result;
}
```

6.21.2.1. Checksum Example

Here an example using the stream operator.

```
MidiMessage :=
  0xF0                                       // start byte
  MidiData$data                              // message contents
  @lblEnd                                    // label containing bit
position
  0:1 checksum = ( [+;0,@lblEnd,8,$] % 128 ) // checksum
  0x7F                                       // last byte
;
```

Let's take a look what the cryptic looking checksum function does. The stream operator has following arguments:

- + : compute the sum of the field values
- 0 : start bit position (before the first byte 0xF0)
- @lblEnd : end bit position (right before the checksum byte)
- 8 (bit) : field size of 1 byte
- $: the field value is not modified

The checksum is the result of the stream operator modulo 128.

```
byte[] message = { 0xF0, ... /* MidiData */, XX, 0x7F }; // XX = checksum
value
BitStream bitstream = BitStream.wrap(message);
int start = 0;
int end   = bitstream.getSize()-(2*8); // 2nd last byte, right before XX
int size  = 8; // 1 byte

int checksum = 0;
```

```
for (int pos=start ; pos < end ; pos += size )
{
  if (pos == start) { checksum  = bitstream.getInt(size); }
  else              { checksum += bitstream.getInt(size); }
}
checksum = checksum % 128;
// now XX == checksum
```

7. Cookbook

7.1. Reuse checksum

Checksum declaration:

```
Checksum :=
  @CHKSUM_END
  0:1 checksum:7 = ([+;0;@CHK_SUM_END;8;$]&0x7F)
;
```

Reuse checksum:

```
Packet1 := ... Checksum$$ 0xF7;
Packet2 := ... Checksum$$ 0xF7;
...
PacketN := ... Checksum$$ 0xF7;
```