SDNet PX Programming Language

User Guide

UG1016 (v2017.4) December 20, 2017

XILINX
ALL PROGRAMMABLE™

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
12/20/2017	2017.4	Released with SDNet 2017.4 without changes from the previous version.
11/29/2017	2017.3	Initial Xilinx release.

Table of Contents

SDNet PX Programming Language User Guide

Introduction

PX is the high-level domain-specific programming language provided for the Programmable Packet Processor (PPP) component of Xilinx® SDNet™. The same language can be used for two purposes: to generate synthesizable RTL code for the architecture of a PPP instance and to change firmware for an existing PPP instance. The overall design methodology around PX is described in the *SDNet Packet Processor User Guide* (UG1012).

PX is designed to allow users to focus on the desired packet processing functions, and to not be concerned with the careful implementation details needed to achieve high performance. PX is a declarative language. It is concerned with the *what* rather than the *how* of packet processing. In other words, it describes a problem rather than defines a solution. More specifically, it describes rules to be applied to packets, and not the implementation of these rules. Thus, PX differs from typical software programming languages in that it lacks a temporal characteristic for specifying the order in which rules are applied. However, PX differs from typical hardware description languages in that it lacks a spatial characteristic for specifying the machinery to be used for rule application. The aim is that the user can concentrate on packets and protocols, and not worry about implementation details.

A PX program consists of a collection of rules for packet processing. It is object-oriented, with two basic types of objects: engines, which perform packet processing, and interfaces, which allow communication between engines and the outside world. The PX program specifies the capabilities of engines and systems built by making connections between engines via interfaces.

The core built-in engine classes are ParsingEngine, TupleEngine, EditingEngine, and LookupEngine, augmented with a UserEngine class to allow the integration of user-provided engines. Particular engines are defined as subclasses of these built-in classes. The programmer provides standard methods within the subclasses to specify the desired processing behavior. The core built-in interface classes are Packet and Tuple, used to specify packet-stream or data-group styles of connection between engines, respectively, augmented with a Plain class to allow untyped connections on user-provided engines. The arrival of an input at an engine triggers its processing function, resulting in a corresponding output, with the exact correspondence depending on engine type.

A PX program is expressed as a single declaration block. A declaration block can contain three types of declaration statements, arranged in any order that is logical for the user:

- Class Declarations, page 6
- Constant Declarations, page 16
- Structure Declarations, page 17

Declaration blocks can occur locally within class declarations. In some contexts, these inner declaration blocks can contain map declarations (Map Declarations, page 13) or object declarations (Object Declarations, page 17). Identifiers, constants, and expressions used in declarations are defined in Identifiers, Constants, and Expressions, page 18.

The class declarations are the heart of the PX program, and always describe subclasses of built-in classes. The permitted subclass types depend on the surrounding context of the declaration block. They can include engine subclasses or interface subclasses. Certain engine subclasses include subclasses of the built-in Section class. This class is used to describe a section of a packet. The top-level class declarations can also include subclasses of the built-in System class. This class is used to describe an engine that is formed from a collection of sub-engines and connections between these sub-engines via interfaces.

Object declarations are used to declare objects belonging to a declared class, though in many cases objects are not declared explicitly, as object creation and destruction detail is not necessary because of the overall declarative programming style.

A complete PX program can be split over multiple files, arranged in any way that is logical for the user. Explanatory comments can be inserted in a PX program as in C, either as single lines beginning with "//" or as any text (maybe over multiple lines) enclosed between "/*" and "*/".

Appendix 1 contains a complete example PX description of a PPP instance.

A full description of using the PX compiler is contained in the *SDNet Packet Processor User Guide* (UG1012).

Class Declarations

An SDNet class declaration has the following general format:

```
class  classIdentifier  :: parentClassIdentifier parameterList {
   Statements
}
```

The *classIdentifier* is the name of the class being declared, and the *parentClassIdentifier* is the name of its built-in parent class. If present, the *parameterList* is a bracketed list of parameters that are specific to the parent class: (*Param1, ... , ParamN*). These constitute configuration parameters for the subclass instance.

The *statements* forming the body of the class declaration consist of declaration blocks and method definitions, arranged in any order that is logical for the user. A method definition has the following general format:

```
method methodIdentifier = methodBody ;
```

The permitted local class declarations and method definitions depend on the parent class. The *methodIdentifier* is always a built-in identifier associated with the parent class. In a few special cases for certain defined methods, the = *methodBody* part is optional.

Engine Subclass Declarations

There are five built-in parent engine classes: ParsingEngine, TupleEngine, EditingEngine, LookupEngine, and UserEngine.

ParsingEngine Subclass Declarations

A section is any contiguous part of a packet, typically but not necessarily a header that is at the beginning. A parsing engine works by a forward traversal of sections in a packet.

The *classIdentifier* in a ParsingEngine subclass declaration can be used in System subclass declarations elsewhere in the PX description, and also indicates the module name given to the corresponding implemented parsing engine.

The ParsingEngine class has the following *parameterList*:

```
(maxPacketRegion,maxSectionDepth,firstSection)
```

This gives configuration parameters for the parsing engine instance. The first two parameters are constant positive integer values specifying the maximum size (in bits) of the initial part of the packet analyzed by the engine and the maximum number of packet sections traversed by the engine. The third parameter is the name of a Section subclass declared within this declaration which is the first section to be traversed.

The body of a ParsingEngine subclass declaration contains a declaration block, and no method definitions. The declaration block can contain any relevant constant or structure declarations. It also contains Section subclass declarations and Tuple subclass object declarations.

Section subclass declarations describe packet sections that are processed. Each subclass declaration describes the format of a section, and also defines methods for parsing that section and extracting data from it. Dynamic objects of the Section subclass are implicitly created during parsing of a packet.

Tuple subclass object declarations describe groups of data that accompany each packet and can be input, accessed, or updated by the parsing engine, and output. Each subclass object declaration defines the format of a data group, and then fields within this object can be read and written during the processing of packet sections.

The parsing engine is told the type of the first packet section, and thus a first Section subclass object to be created and applied. The methods defined within each Section subclass determine the type of the next section, and hence the next Section object to be applied. This process continues until a Section object method determines that parsing is complete, or that an error has occurred. At this point, any output Tuple objects will be populated and ready for output. For example, when processing standard Ethernet packets, successive Section objects might handle Ethernet, VLAN, IP, and TCP, headers, and an output Tuple object might contain a standard TCP/IP five-tuple for lookup.

TupleEngine Subclass Declarations

A tuple engine is similar to a parsing engine, except that it does not follow sections of a packet. It only inputs and outputs tuples, and works by traversal of sections of tuple processing.

The *classIdentifier* in a TupleEngine subclass declaration can be used in System subclass declarations elsewhere in the PX description, and also indicates the module name given to the corresponding implemented tuple engine.

The TupleEngine class has the following *parameterList*:

```
(maxSectionDepth,firstSection)
```

This gives configuration parameters for the tuple engine instance. The first parameter is a constant positive integer value specifying the maximum number of sections of tuple processing traversed by the engine. The second parameter is the name of a Section subclass declared within this declaration which is the first section to be traversed.

The body of a TupleEngine subclass declaration contains a declaration block, and no method definitions. The declaration block can contain any relevant constant or structure declarations. It also contains Section subclass declarations and Tuple subclass object declarations.

Section subclass declarations describe sections of tuple processing. Each subclass declaration defines methods for the tuple processing. Dynamic objects of the Section subclass are implicitly created during the traversal process.

Tuple subclass object declarations describe groups of data that are input, accessed, or updated by the tuple engine, and output. Each subclass object declaration defines the format of a data group, and the fields within this object can be read and written during the sections of tuple processing.

The tuple engine is told the type of the first section, and thus a first Section subclass object to be created and applied. The methods defined within each Section subclass determine the type of the next section, and hence the next Section object to be applied. This process continues until a Section object method determines that tuple processing is complete, or that an error has occurred. At this point, the output Tuple objects are populated and ready for output.

EditingEngine Subclass Declarations

A section is any contiguous part of a packet, typically but not necessarily a header that is at the beginning. An editing engine works by a forward traversal of sections in a packet.

The *classIdentifier* in an EditingEngine subclass declaration can be used in System subclass declarations elsewhere in the PX description, and also indicates the module name given to the corresponding implemented parsing engine.

The EditingEngine class has the following *parameterList*:

```
(maxPacketRegion,maxSectionDepth,firstSection)
```

This gives configuration parameters for the editing engine instance. The first two parameters are constant positive integer values specifying the maximum size (in bits) of the initial part of the packet analyzed by the engine and the maximum number of packet sections traversed by the engine. The third parameter is the name of a Section subclass declared within this declaration which is the first section to be traversed.

The body of an EditingEngine subclass declaration contains a declaration block, and no method definitions. The declaration block can contain any relevant constant or structure declarations. It also contains Section subclass declarations and Tuple subclass object declarations.

Section subclass declarations describe packet sections that are processed. Each subclass declaration describes the format of a section, and also defines methods for editing that section by updating, inserting, or removing, data. Dynamic objects of the Section subclass are implicitly created during editing of a packet.

Tuple subclass object declarations describe groups of data that accompany each packet, and can be input, accessed, or updated by the editing engine, and output. Each subclass object

declaration defines the format of a data group, and fields within this object can be read and written during the processing of packet sections.

The editing engine is told the type of the first packet section, and thus a first Section subclass object to be created and applied. The methods defined within each Section subclass determine the type of the next section, and hence the next Section object to be applied. This process continues until a Section object method determines that editing is complete, or that an error has occurred. At this point, any output Tuple objects are populated and ready for output. For example, when processing standard Ethernet packets, successive Section objects might update Ethernet and IP headers, and insert or remove VLAN headers.

LookupEngine Subclass Declarations

The *classIdentifier* in a LookupEngine subclass declaration can be used in System subclass declarations elsewhere in the PX description, and also provides a prefix for the module name given to the corresponding implemented lookup engine.

The LookupEngine class has the following *parameterList*:

```
(lookupType,capacity,keyWidth,valueWidth,responseType,external)
```

This gives selection and configuration parameters for the lookup engine instance. The first parameter must be present, and specifies the type of lookup: EM (exact match, i.e., binary content addressable memory), LPM (longest-prefix match), TCAM (ternary content addressable memory), or DIRECT (random access memory). The second parameter must also be present, and is a constant positive integer value specifying the maximum number of entries stored in the lookup engine.

If present, *keyWidth* is a constant positive integer value specifying the width, in bits, of the search key for a lookup. If present, *valueWidth* is a constant positive integer value specifying the width, in bits, of the search result from a lookup. If either parameter is absent, the required width is inferred from the method definitions within the rest of the declaration.

If present, *responseType* is a constant non-negative integer value, in the range 0 to 2, specifying the format of the response returned by the lookup engine instance. 0 indicates value only, 1 indicates a hit/miss bit followed by the value, and 2 indicates a hit/miss bit followed by the value followed by the key. If this parameter is absent, the default is a hit/miss bit followed by value.

If present, *external* is a constant non-negative integer value, either 0 or 1, specifying whether the lookup engine instance implementation is external to the overall PPP system instance. 0 indicates non-external and 1 indicates external. If this parameter is absent, the default is non-external.

The body of a LookupEngine subclass declaration contains a declaration block and two method definitions. The declaration block can contain any relevant constant, structure, or Tuple subclass declarations. It also contains Tuple subclass object declarations. One input

Tuple and one output Tuple are used to describe the groups of data that are input to, and output from, the lookup engine, respectively.

The lookup engine gives requests to a lookup block instance generated from the PPP standard library, and takes responses from this lookup block instance. One method definition describes the lookup request, and the other method definition describes the lookup response.

The first method has the following form:

```
method send_request = {
    key = identifier
}
```

The *identifier* is a local input Tuple subclass object identifier. The second method has the following form:

```
method receive_response = {
    identifier = value
}
```

The *identifier* is a local output Tuple subclass object identifier. Note that the format of the group of data associated with the output Tuple subclass object has semantics corresponding to the *responseType* parameter (or its default, if absent), that is, it covers the full response format not just the value returned from the lookup.

UserEngine Subclass Declarations

The *classIdentifier* in a UserEngine subclass declaration can be used in System subclass declarations elsewhere in the PX description, and also indicates the module name of the corresponding implemented user engine, provided either by an SDNet library or directly by the user.

The UserEngine class has the following *parameterList*:

```
(maxLatency,controlWidth)
```

This gives information parameters about the user engine instance. The first parameter must be present, and is a constant positive integer value specifying the latency, measured in cycles relative to a clock specified at compile time, between arrival of the latest input and availability of the latest output. If present, *controlWidth* is a constant non-negative integer value specifying the width of the control interfaces of the engine in bits, with 0 indicating no control interface. If this parameter is absent, the default is no control interface.

The body of a UserEngine subclass declaration contains a declaration block, and no method definitions. The declaration block can contain any relevant constant, structure, or Packet or Tuple subclass declarations. It also contains Packet subclass object declarations, Tuple subclass object declarations, and/or Plain subclass object declarations. Packet subclass object declarations describe packet streams that are input to or output from the user engine. There can be at most one input packet stream and one output packet stream. Tuple

subclass object declarations describe the format of groups of data that are input to or output from the user engine. Plain subclass object declarations describe the direction and width of other, untyped, input or output interfaces exposed by the user engine.

The Packet, Tuple, and Plain subclass object declarations correspond explicitly to the input and output interfaces of the implemented user engine. The name of each interface must match the identifier used for the corresponding declared object.

Interface Subclass Declarations

There are three built-in parent interface classes: Packet, Tuple, and Plain.

Packet Subclass Declarations

A Packet subclass declaration describes a packet stream that can be input to an engine or subsystem, accessed or updated by an engine or subsystem, and/or output by an engine or subsystem.

The Packet class has the following *parameterList*:

```
(direction)
```

The *direction* parameter can be `in` (input-only), `out` (output-only), or `inout` (input and output). A Packet subclass declaration has an empty body. This is because operations on packets are achieved indirectly via Section subclasses.

Within the *ParsingEngine* and *EditingEngine* classes, there is a predeclared anonymous Packet subclass object with *direction* parameter `inout` that represents the packet stream that passes through the parsing and editing engines. This corresponds to the interfaces of the implemented engine packet_in (input) and packet_out (output).

Tuple Subclass Declarations

A Tuple subclass declaration describes a data stream that can be input to an engine or subsystem, accessed or updated by an engine or subsystem, and/or output by an engine or subsystem.

The Tuple class has an optional *parameterList*:

```
(direction)
```

If this *parameterList* is present, the *direction* parameter can be `in` (input-only), `out` (output-only), or `inout` (input and output). If it is absent, the tuple is neither input nor output and the *direction* parameter can only be used locally within an engine.

The body of a Tuple subclass declaration contains a declaration block, and no method definitions. The declaration block can contain any relevant constant declarations, and must contain a single structure declaration. The structure declaration describes the format of the data group forming the tuple. The fields within the structure can be directly accessed as

operands or result destinations, by preceding their names with the name of an object of the Tuple subclass. For example, if a tuple object is named "key" and the structure contains a field called "priority", this field can be accessed using the identifier `key.priority`.

Plain Subclass Declarations

A Plain subclass declaration describes an untyped interface to a user engine or subsystem.

The Plain class has the following *parameterList*:

```
(direction,width)
```

The *direction* parameter is either `in` (input-only) or `out` (output-only). The width parameter is a constant positive integer value specifying the width of the interface in bits.

A Plain subclass declaration has an empty body. This is because no further information about the interface is specified in the PX description.

Section Subclass Declarations

The built-in Section class represents the section of a packet that is a contiguous sequence of bits within the packet. Typically, a section is a header of a packet or a number of adjacent headers, but it can also be a trailer of a packet or some arbitrary part of the payload of a packet. As a special case, a section can correspond to a null part of a packet when it contains processing not directly related to the packet.

The *classIdentifier* in a Section subclass declaration can be used elsewhere in the PX description to refer to this class.

The Section class has an optional *parameterList*:

```
(levels)
```

If this *parameterList* is present, it explicitly specifies the section traversal levels where this section type can occur during any input packet's sequence of sections. This extra information can lead to more resource-efficient implementations. The qualifier is a list of single level numbers, or ranges of level numbers. Two examples are: `(1,2)` and `(2,4:6,8)`, where 4:6 is just a shorthand for the range 4,5,6.

The body of a Section subclass declaration contains a declaration block, and at least one method definition. The declaration block can contain any relevant constant declarations, and can contain a single structure declaration. If present, the structure declaration explicitly describes the format of the packet section. The fields within the structure can be directly accessed within the context of the subclass, as operands or result destinations, using their names. The Section subclass methods describe the handling of a packet section.

Map Declarations

The declaration block can also contain one or more map declarations. These describe content-addressable memories that map keys to results, and can be used in expressions within the context of the subclass. Each key is a non-negative integer constant, and all keys must be unique. The results are either all Section subclass identifiers or all non-negative integer constants, depending on the specific context within which the map is used. Optionally, a default result can be specified, used when no match is made for a key supplied to the map. If this is not present and no match is made, the result of the map is undefined. A declaration has the following form:

```
map mapIdentifier {
    (key1, result1),
    (key2, result2),
      ...
    (keyn, resultn),
    defaultresult
}
```

Update Method

One optional method definition specifies updates to be done to fields of Tuple subclass objects declared in the surrounding parsing or editing engine, or to packet fields declared in the structure declaration in the surrounding editing engine. This method has the following general form:

```
method update = {
    destfield1 = valueexpr1,
    ... ,
    destfieldn = valueexprn
}
```

The left-hand side of each assignment is the identifier of a field, and the right-hand side of the assignment is a value expression. Multiple updates to the same field (or sub-field) have an undefined effect. The other two methods describe the traversal to the next section.

Move_to_Section Method

The move_to_section method must be present. It specifies the Section subclass to be applied at the next traversal step, using a class expression. This expression either provides the name of a Section subclass (which might be the same as the current class) or provides the special `done` class. The latter indicates the end of traversal, and is associated with a non-negative integer value that indicates the final status of the packet traversal, e.g., `done(0)`. This method has the following general form:

```
method move_to_section = classexpr ;
```

Increment_offset, Set_offset, And Set_virtual_offset, Methods

One method, `increment_offset`, `set_offset`, or `set_virtual_offset` (but no more than one of these) can be present.

The `increment_offset` method specifies the number of bits to skip in the packet to move from the current section being handled to the next section to be handled, using a value expression. This expression provides a non-negative integer value. Note that this means that traversal always proceeds in a forward direction in the packet, with no backtracking allowed. The special value expression `rop()` can be used to move to after the final bit of the packet. The method has the following general form:

```
method increment_offset = valueexpr ;
```

The `set_offset` and `set_virtual_offset` methods specify an absolute bit position to skip to in the input packet to move from the current section being handled to the next section to be handled, using a value expression. This expression provides a non-negative integer value. The difference between the two methods is that `set_offset` measures the offset from the beginning of the packet at that instance, whereas `set_virtual_offset` measures the offset from the beginning of the original input packet. These differ when packet editing (insertion and/or removal) has taken place and changed the size of the packet. The value must be greater than or equal to the current offset. This means that traversal always proceeds in a forward direction in the packet, with no backtracking allowed. The special value expression `eop()` can be used to move to after the final bit of the packet. The method has the following general form:

```
method set_offset = valueexpr ;
```

Within parsing engine subclass declarations, the offset modification method is applied after all other section actions are done. If no method is present, the offset is incremented by the size of the section structure, if one is present.

Within editing engine subclass declarations, the offset modification method is applied before any other section actions are done. The offset is also incremented (after all section actions are done) by the size of the removal if a remove method is present, or by the size of the section structure if one is present and no remove or insert method is present.

Additional Methods For Editing Engines

Two additional methods, remove and insert, can be included within editing engine subclass declarations, but not within parsing engine subclass declarations. They specify the removal and/or insertion of a packet data in the section.

The remove method specifies that a number of data bits are to be removed at the current offset in the packet, with a value expression indicating the number of bits. The special value expression `rop()` can be used to remove all remaining bits of the packet. The method has the following general form:

```
method remove = valueexpr ;
```

When a structure is present in the section a short form can be used when the number of bits removed is equal to the size of the structure:

```
method remove ;
```

The insert method specifies that a number of data bits are to be inserted before the current offset in the packet, with a value expression indicating the number of bits. The inserted bits are determined as follows (where struct size is defined as zero if no structure is present in the section):

- If the value of the expression *v* is less than or equal to the struct size, then the initial *v* bits of the structure are used.
- If the value of the expression *v* is greater than the struct size, then the *s* bits of the structure padded with *v-s* trailing zero bits are used.

The method has the following general form:

```
method insert = valueexpr ;
```

When a structure is present in the section a short form can be used when the number of bits inserted is equal to the size of the structure:

```
method insert ;
```

When both remove and insert methods are present, the remove action is done before the insert action. When neither a remove nor an insert method is present, the section has no surgical effect on the packet, but in-place changes can be made by setting fields within an update method.

System Subclass Declarations

The built-in System class represents an engine formed from a collection of sub-engines, and connections between these sub-engines via interfaces. Some or all of the sub-engines can themselves be instances of System subclass objects, allowing hierarchical system organizations.

The *classIdentifier* in a System subclass declaration can be used in System subclass declarations elsewhere in the PX description to refer to this class (representing a sub-engine), and also indicates the module name given to the corresponding implemented system.

The System class does not have a *parameterList*. The body of a System subclass declaration contains a declaration block, and one method definition. The declaration block contains only object declarations.

Two types of objects can be declared. The first, Interface subclass objects, represent the perimeter input and output interfaces of the system. There must be precisely one input Packet subclass object and one output Packet subclass object, in other words, a system has

a packet stream input and a packet stream output. The second, Engine subclass objects or System subclass objects, represent the system's sub-engine components.

The method definition describes connections between system perimeter interfaces and/or component interfaces, specifically connections between Interface subclass objects. This method has the following general form:

```
method connect = {
    destInterface1 = sourceInterface1,
    ... ,
    destInterfacen = sourceInterfacen
}
```

The source and destination interfaces can either be locally declared interface objects, or interface objects within the locally declared engine or system objects. The anonymous Packet interface objects within ParsingEngine and EditingEngine objects are referred to as `packet_in` and `packet_out`, for the input and output interfaces, respectively.

The connections declared within the method must ensure that every locally declared interface object and every interface object within the locally declared engine or system objects are used only one time as a destination if it has in or in-out direction. It can be used one or more times as a source if it has out or in-out direction, and only one time if it is a Packet interface. In other words, no interface is left unconnected, and no input interface is multiply connected.

Constant Declarations

A constant declaration associates an identifier with a constant non-negative integer value. A declaration has the following form:

```
const identifier  = value;
```

See Identifiers, Constants, and Expressions, page 18 for definitions of identifiers, constants, and expressions.

Constant declarations are included as statements in declaration blocks.

Structure Declarations

A structure declaration defines a single data structure format. This format consists of an ordered list of one or more fields, each field consisting of a bit string of a specified width or a nested sub-structure. A declaration can have the following direct form:

```
struct identifier {
    identifier1 : fielddesc1,
    ... ,
    identifiern : fielddescn
}
```

See Identifiers, page 18 for definitions of identifiers. The struct identifier need only be included if the structure is referred to elsewhere in the PX description. The inner block describes the format of the data structure, the fields being declared as a comma-separated list. Each field declaration has an identifier and a descriptor, which is either a width in bits specified by a non-negative integer constant (see Constants, page 19) or the name of a structure declared elsewhere (either locally or in a surrounding context), indicating that the field is a sub-structure.

Alternatively, a structure declaration can have the following indirect form:

```
struct identifier;
```

where the *identifier* is the name of a structure declared elsewhere (either locally or in a surrounding context) using the first structure declaration. Structure declarations are included as statements in declaration blocks.

Object Declarations

An object declaration instances one or more objects of a specific class. In many situations, objects are not explicitly declared, as their creation and existence is implicit in the semantics of the various declarative PX language constructs.

An object declaration has the following form:

```
classIdentifier  objectIdentifier1, …, objectIdentifiern;
```

where *classIdentifier* is the name of a class declared elsewhere (either locally or in a surrounding context). The object identifiers *objectIdentifier1*, ..., *objectIdentifiern* are used for the object instances. Object declarations are included as statements in declaration blocks.

Identifiers, Constants, and Expressions

Identifiers

Identifiers are used for classes, objects, methods, maps, structures, fields within structures, and constants. As such, they must be distinct within the same context. Identifiers are formed from a combination of letters, digits, and underscore (_) characters, beginning with a letter, containing no double underscores, and ending with a letter or digit. Identifiers are case sensitive.

A field within a Tuple subclass object is referred to by using the object's identifier, followed by a dot (.) and then the field's identifier. A field within a sub-structure of a structure is referred to by using the sub-structure's identifier, followed by a dot and then the field's identifier.

Identifiers have scopes, so an identifier might be re-used with local significance that overrides a use with a more global significance. Local scopes exist within class declarations, method declarations, and structure declarations.

The following PX reserved words cannot be used, with any case sensitivity, as identifiers:

```
class   const   else   if   map   method   struct
```

Various identifiers have predefined significance in relevant contexts, as described:

```
done                    Plain
EditingEngine           remove
eop                     rop
increment_offset        Section
insert                  set_offset
level                   set_virtual_offset
LookupEngine            sizeof
move_to_section         System
offset                  Tuple
Packet                  TupleEngine
packet_in               update
packet_out              UserEngine
ParsingEngine           virtual_offset
```

Constants

Constants express fixed non-negative integer values. Numbers can be expressed in decimal, hexadecimal (prefixed by 0x), or binary (prefixed by 0b), notation. If desired, more complex constant values can be formed using expressions involving numbers, unary and binary operators (as defined in the next section), and parentheses.

Expressions

Value Expressions

Value expressions are used to compute non-negative integer values. There are two special value expressions which can only be used atomically, without operators:

- The `eop()` expression denotes the offset of the end of a packet.
- The `rop()` expression denotes the number of remaining bits of a packet.

These expressions can only be used in the particular contexts described in Increment_offset, Set_offset, And Set_virtual_offset, Methods, page 14 and Additional Methods For Editing Engines, page 14.

Other expressions can involve operators, and the following as operands:

- Constants (Constants, page 19)
- Tuple subclass object field identifiers (Identifiers, page 18)
- Section subclass structure field identifiers (Identifiers, page 18)
- Map accesses: using a map containing non-negative integer constants as results

  mapname `(`*valueexpr*`)`
- Use of the function `level()` to return the current section traversal level (1, 2, ...) where the Section subclass that the function is used within is being used
- Use of the functions `offset()` or `virtual_offset()` to return the current offset, or virtual offset (relative to the original input packet), in bits within the packet, respectively
- Use of the function `sizeof()` to return the (constant) size in bits of a structure, object, class, or expression, given as the argument

Expressions can involve unary operators, which have right-to-left associativity, and have equal precedence and the highest precedence over other operators:

- - (negation), ! (Boolean not), ~ (bitwise complement)

Expressions can involve binary operators, which have left-to-right associativity, and are listed here in groups of equal precedence with groups having decreasing order of precedence:

- * (multiplication), / (division), % (modulus)
- + (addition), - (subtraction)
- << (left shift by constant distance), >> (right shift by constant distance)
- <, <=, >, >= (unsigned integer <, <=, >, and >=, comparisons); result: false=0, true=1
- == (equals), != (not equals); result: false=0, true=1
- & (bitwise and)
- ^ (bitwise exclusive or)
- | (bitwise or)
- && (Boolean and)
- || (Boolean or)

Parentheses can be used in expressions to override precedence rules, or for other cosmetic reasons.

A multi-armed conditional test can be used to select between top-level value expressions used in method definitions:

```
if (valueexpr) valueexpr  else if (valueexpr) valueexpr   …  else valueexpr
```

where the conditions are the expressions in parentheses.

Evaluation of value expressions, and multi-armed conditional tests selecting between top-level value expressions, is carried out using a “natural width” based on operand and result destination bit widths. This width is ascertained by the following rules.

- Expression leaf operands have inherent bit widths, as follows:
  - (Non-negative) constant
    - Binary representation: the number of binary digits
    - Decimal representation: the minimum number of bits to represent the decimal value
    - Hexadecimal representation: four times the number of hexadecimal digits

- Tuple subclass object field value: the defined bit width of the field
- Section subclass structure field value: the defined bit width of the field
- map access result: the maximum bit width over all the constants stored in the map
- level() function result: the minimum number of bits needed to represent the value of the parsing or editing engine maxSectionDepth parameter
- offset() or virtual_offset() function result: the minimum number of bits needed to represent an offset in the range of the parsing or editing engine maxPacketRegion parameter
- sizeof() function result: the minimum number of bits needed to represent the size of the structure, class, object, or expression, given as its argument

- Expression computation and result width is then determined by the maximum bit width over all leaf operands within the expression context, which can be:
  - Right-hand side of a move_to_section, increment_offset, set_offset, remove, or insert, method definition
  - Update expression within the right-hand side of an update method
  - Constant value specification in any other context apart from method definitions
- It is then further maximized with the width of the destination, when the expression is assigned:
  - increment_offset or set_offset result: the minimum number of bits needed to represent an offset in the range of the parsing or editing engine maxPacketRegion parameter
  - Tuple subclass object field: the defined bit width of the field
- All shorter operands are zero-prefixed to extend them to the computation width.
- Computation is carried out as follows, where n is the computation width:
  - Arithmetic operations (+,-,*,/,%): are unsigned and computed modulo 2^n
  - Comparison operations (==,!=,<,<=,>,>=): use lexicographic ordering of n-bit strings
  - Logical operations (~,<<,>>,&,|,^): are performed on n-bit strings
  - Boolean operations (!,&&,||): are performed on n-bit strings (zero or non-zero values)

Class Expressions

Class expressions are used to compute Section subclass identifiers.

Expressions can involve the following as operands:

- Section subclass identifiers (Identifiers, page 18)
- The special done() class (Section Subclass Declarations, page 12)
- Map accesses: using a map containing either of the above two operand types as results

  mapname (*valueexpr*)

A multi-armed conditional test can be used to select between top-level class expressions used in method definitions:

```
if (valueexpr) classexpr  else if (valueexpr) classexpr   …  else classexpr
```

where the conditions are the expressions in parentheses.

Example PX Program

Note: This is an illustrative example showing the range of PX language constructs in use, and should not be taken as a program that is fit for any particular practical purpose.

```
// *********************************************


class Packet_input  :: Packet(in) {}
class Packet_output :: Packet(out) {}


class DecryptSystem :: System {

  Packet_input          inP;
  Packet_output         outP;

  DecryptParser         parser;
  SAD_CAM               sad_table;
  KEY_RAM               key_table;
  PadEditor             pad_editor;
  AES_CTR               aes_ctr;
  DecryptSeqNum         seq_num;
  DecryptFormatEditor   format_editor;

  method connect = {
    parser.packet_in = inP,

    sad_table.request = parser.key_tuple,

    seq_num.sad_tuple  = sad_table.response,
    seq_num.seq_tuple  = parser.seq_tuple,

    pad_editor.packet_in = parser.packet_out,
    pad_editor.esp_tuple = parser.esp_tuple,
    pad_editor.sad_tuple = sad_table.response,
    pad_editor.check_tuple = seq_num.check_tuple,

    key_table.request    = seq_num.index_tuple,

    aes_ctr.packet_in = pad_editor.packet_out,
    aes_ctr.key_tuple = key_table.response,
    aes_ctr.pad_tuple = pad_editor.pad_tuple,
    aes_ctr.iv_tuple  = parser.iv_tuple,

    format_editor.packet_in = aes_ctr.packet_out,
    format_editor.esp_tuple = parser.esp_tuple,
    format_editor.pad_tuple = pad_editor.pad_tuple,
    format_editor.sad_tuple = sad_table.response,
    format_editor.aes_tuple = aes_ctr.aes_tuple,
```

```
    outP = format_editor.packet_out
  }
}

// ********************************************

// ********************************************

struct espFormat {
        ip_version  :     1,  // 0=IPv4, 1=IPv6
        offset      :    16,  // start of the ESP (bytes)
        hdr_len     :     8,  // IP header length (bytes)
        pld_len     :    16,  // IP payload length (bytes)
        hdr_crc     :    16   // IPv4 checksum
}

class DecryptEspTupleOut :: Tuple(out) {
    struct espFormat;
}

class DecryptEspTupleIn :: Tuple(in) {
    struct espFormat;
}


// ********************************************

struct keyFormat {
      //dst_addr    :  128, // destination IP address
        spi         :   32  // security payload identifier
}

class DecryptKeyTupleOut :: Tuple(out) {
    struct keyFormat;
}

class DecryptKeyTupleIn :: Tuple(in) {
    struct keyFormat;
}


// ********************************************

const NUM_ENTRIES = 64;
const INDEX_WIDTH = 6;

struct spdSadFormat {
        match       :   1,      // matching record found in SPD/SAD
        mode        :   1,      // 0=transport, 1=tunnel
        bozo2       :   8-INDEX_WIDTH,  // alignment
        index       :   INDEX_WIDTH,    // address of the entry
        bozo        :   1,      // alignment
        iv_present  :   1,      // ESP header contains Initialization Vector
        action      :   2,      // 0=bypass, 1=drop, 2=encrypt, 3=decrypt
        spi         :   32      // Security Parameters Index
}

class SpdSadTupleOut :: Tuple(out) {
    atruct spdSadformat;
}
```

```
class SpdSadTupleIn :: Tuple(in) {
    struct spdSadformat;
}


// ********************************************

struct cryptoFormat {
        keylen      :     2,    // encryption key length: 0=128, 1=192, 2=256
        key         :   256,    // encryption key (msb-aligned)
        nonce       :    32     // nonce for AES_CTR mode
}

class CryptoTupleOut :: Tuple(out) {
    struct cryptoFormat;
}

class CryptoTupleIn :: Tuple(in) {
    struct cryptoFormat;
}



// ********************************************

struct tunnelFormat {
        version    : 1, // 0=IPv4, 1=IPv6
        id         : 20, // only 16 LSBs are used for IPv4
        ttl        : 8,
       checksum   : 16,  // precomputed over IPv4-formatted data, only used for IPv4,
augmented with inner DS/ECN
        src_addr   : 32, // only 32 LSBs are used for IPv4
        dst_addr   : 32  // only 32 LSBs are used for IPv4
}

class TunnelTupleOut :: Tuple(out) {
    struct tunnelFormat;
}

class TunnelTupleIn :: Tuple(in) {
    struct tunnelFormat;
}



// ********************************************

struct padFormat {
        invmod    :   2,  // number of invalid 4-byte chunks in last 16-byte block
        padding   :   4,  // amount of padding needed to align the start of payload
to 16-byte boundary
        begin     :   8,  // index of the first encrypted 16-byte block
        end       :   8   // index of the last encrypted 16-byte block + 1
}

class PadTupleOut :: Tuple(out) {
    struct padFormat;
}
```

```
class PadTupleIn :: Tuple(in) {
    struct padFormat;
}



// ***********************************************
struct indexFormat {
        index       :   INDEX_WIDTH // address of the entry
}

class IndexTupleOut :: Tuple(out) {
    struct indexFormat;
}

class IndexTupleIn :: Tuple(in) {
    struct indexFormat;
}



// ***********************************************

struct seqNumFormat {
        num         : 32
}

class SeqNumTupleIn :: Tuple(in) {
    struct seqNumFormat;
}

class SeqNumTupleOut :: Tuple(out) {
    struct seqNumFormat;
}



// ***********************************************

struct seqCheckFormat {
        ok          : 1
}

class SeqCheckTupleIn :: Tuple(in) {
    struct seqCheckFormat;
}


class SeqCheckTupleOut :: Tuple(out) {
    struct seqCheckFormat;
}
```

```
// ************************************************

struct initVectorFormat {
        iv          : 64
}

class InitVectorTupleIn :: Tuple(in) {
    struct initVectorFormat;
}

class InitVectorTupleOut :: Tuple(out) {
    struct initVectorFormat;
}



// ************************************************

struct aesFormat {
       pad     : 8,  // amount of padding needed to align the end of payload to 4-byte
boundary
        next    : 8   // ESP inner protocol
}

class AesTupleOut :: Tuple(out) {
    struct aesFormat;
}

class AesTupleIn :: Tuple(in) {
    struct aesFormat;
}



// ************************************************

class DecryptParser :: ParsingEngine (
    16383, // 2 KBytes
    9,
    Ethernet
)

{

    // ************************************************
    DecryptEspTupleOut esp_tuple;
    DecryptKeyTupleOut key_tuple;

    SeqNumTupleOut seq_tuple;

    InitVectorTupleOut iv_tuple;


    // ************************************************

    // EtherType encodings
    const QINQ_TYPE     = 0x88a8;
    const VLAN_TYPE     = 0x8100;
    const MPLS_UC_TYPE  = 0x8847;
```

```
const MPLS_MC_TYPE  = 0x8848;
const IPv4_TYPE     = 0x0800;
const IPv6_TYPE     = 0x86dd;

// parser error codes
const ERR_ETHERNET = 1;
const ERR_QINQ     = 2;
const ERR_VLAN     = 3;
const ERR_MPLS     = 4;
const ERR_IPv4     = 5;
const ERR_IPv6     = 6;

// IP protocol codes
const ESP_PROTOCOL = 50;


// ************************************************

class Ethernet :: Section(1) {
    struct {
        dmac   : 48,
        smac   : 48,
        tpid   : 16
    }
    map types {
        (QINQ_TYPE,     QINQ),
        (VLAN_TYPE,     VLAN),
        (IPv4_TYPE,     IPv4),
        (IPv6_TYPE,     IPv6),
        (MPLS_UC_TYPE,  MPLS),
        (MPLS_MC_TYPE,  MPLS),
                        done(ERR_ETHERNET)
    }
    method move_to_section = types(tpid);
}

// ************************************************

class QINQ :: Section(2) {
    struct {
        pcp  : 3,
        cfi  : 1,
        vid  : 12,
        tpid : 16
    }
    map types {
        (VLAN_TYPE,     VLAN),
        (QINQ_TYPE,     QINQ),
                        done(ERR_QINQ)
    }
    method move_to_section = types(tpid);
}
```

```
// ************************************************

class VLAN :: Section(3)  {
    struct {
        pcp  : 3,
        cfi  : 1,
        vid  : 12,
        tpid : 16
    }
    map types {
        (IPv4_TYPE,     IPv4),
        (IPv6_TYPE,     IPv6),
        (MPLS_UC_TYPE,  MPLS),
        (MPLS_MC_TYPE,  MPLS),
                        done(ERR_VLAN)
    }
    method move_to_section = types(tpid);
}

// ************************************************

class MPLS :: Section(4:6) {
    struct {
        label   : 20,
        tc      : 3,
        bottom  : 1,
        ttl     : 8
    }
    method move_to_section = if (bottom) IP else MPLS;
}

// ************************************************

class IP :: Section(7) {
    struct {
        version  : 4
    }
    map ip_versions {
        (4,  IPv4),
        (6,  IPv6),
        done(ERR_MPLS)
    }
    method increment_offset = 0;
    method move_to_section = ip_versions(version);
}

// ************************************************

class IPv4 :: Section(8) {
    struct {
        version  : 4,
        hdr_len  : 4,
        dscp     : 6,
        cu       : 2,
        length   : 16,
        id       : 16,
        flags    : 3,
        offset   : 13,
        ttl      : 8,
```

```
            protocol : 8,
            hdr_crc  : 16,
            src_addr : 32,
            dst_addr : 32
        }
        method update = {
            //key_tuple.dst_addr = 0xFFFF00000000 | dst_addr

            esp_tuple.offset = offset()/8 + hdr_len*4,
            esp_tuple.hdr_len = hdr_len*4,
            esp_tuple.pld_len = length - hdr_len*4,
            esp_tuple.hdr_crc = hdr_crc
        }
        // calculate header length
        method increment_offset = hdr_len*32;
        // do not process non-ESP packets
        method move_to_section = if (protocol == ESP_PROTOCOL) ESP else done(1);
    }

    // ************************************************

    class IPv6 :: Section(8) {
        // …
        method move_to_section = done(1);
    }

    // ************************************************

    class ESP :: Section(9) {
        struct {
            spi : 32,
            seq : 32,
            iv  : 64 // may not be valid, need SAD lookup to validate
        }
        method update = {
            key_tuple.spi = spi,
            seq_tuple.num = seq,
            iv_tuple.iv = iv
        }
        method move_to_section = done(0);
    }


}

// *********************************************

class DecryptFormatEditor :: EditingEngine (
    16383, // 2 KBytes
    4,
    DECODE
) {

    const NUM_ENTRIES = 256;
    const INDEX_WIDTH = 8;

    const ESP_PROTOCOL = 50;
```

```
// ************************************************
DecryptEspTupleIn esp_tuple;

// ************************************************
const ACTION_BYPASS  = 0;
const ACTION_DROP    = 1;
const ACTION_DECRYPT = 2;

const MODE_TRANSPORT = 0;
const MODE_TUNNEL    = 1;


SpdSadTupleIn sad_tuple;

// ************************************************
PadTupleIn pad_tuple;

// ************************************************
AesTupleIn aes_tuple;

// ************************************************

class DecryptFormatEditorIntTuple :: Tuple {
    struct {
        checksum : 20,  // IPv4 transport checksum terms
        esp_size : 8    // ESP header + IV + trailing padding + trailer (bytes)
    }
}
DecryptFormatEditorIntTuple int_tuple;

// ************************************************
// go to the start of the data to be encrypted
class DECODE :: Section(1) {
    method update = {
        int_tuple.checksum = esp_tuple.hdr_crc + ESP_PROTOCOL - aes_tuple.next,
        int_tuple.esp_size = sad_tuple.iv_present * 8 + (aes_tuple.pad + 10)
    }
    method move_to_section = if (!sad_tuple.match)                  done(1)
                        else if (sad_tuple.action == ACTION_BYPASS) done(2)
                      else if (sad_tuple.action == ACTION_DROP)   done(3) // DROP
                        else if (sad_tuple.mode   == MODE_TUNNEL)   UNPAD_FRONT
                       else                                         UPDATE_TRANSPORT;
}

// ************************************************
// go to the start of the data to be encrypted
class UPDATE_TRANSPORT :: Section(2) {
    struct {
        version  : 4,
        hdr_len  : 4,
        dscp     : 6,
        ecn      : 2,
        length   : 16,
        id       : 16,
        flags    : 3,
        offset   : 13,
        ttl      : 8,
        protocol : 8,
        hdr_crc  : 16,
```

```
            src_addr : 32,
            dst_addr : 32
        }
        method set_virtual_offset = (esp_tuple.offset - esp_tuple.hdr_len) * 8;
        method update = {
            // decrement packet length by ESP header, trailer and trailing padding
            length = esp_tuple.hdr_len + esp_tuple.pld_len - int_tuple.esp_size,
            // change protocol to decrypted info
            protocol = aes_tuple.next,
            // update checksum folding-in msb carry
            hdr_crc  =  (int_tuple.checksum + int_tuple.esp_size) +
                        ((int_tuple.checksum + int_tuple.esp_size) >> 16)
        }
        method move_to_section = UNPAD_FRONT;
    }

    // ************************************************
    // remove front padding
    class UNPAD_FRONT :: Section(3) {
        method set_virtual_offset = pad_tuple.begin * 128 - pad_tuple.padding * 8;
        method remove = pad_tuple.padding * 8;
        method move_to_section = REMOVE_TRAILER;
    }

    // ************************************************
    // remove trailing padding and trailer
    class REMOVE_TRAILER :: Section(4) {
        method increment_offset = (esp_tuple.pld_len - int_tuple.esp_size) * 8;
        method remove = (aes_tuple.pad + 2) * 8;
        method move_to_section = done(0);
    }

}


// *********************************************


class SAD_CAM :: LookupEngine
(
    EM,
    NUM_ENTRIES, // entries
    sizeof(DecryptKeyTupleIn), // key_width
    sizeof(SpdSadTupleOut), // value_width
    0,  // format 0 == {value}
    0   // internal
) {
   DecryptKeyTupleIn request;
   SpdSadTupleOut response;

   method send_request     = { key      = request }
   method receive_response = { response = value }
}
```

```
// ************************************************


class KEY_RAM :: LookupEngine
(
    DIRECT,
    NUM_ENTRIES, // entries
    INDEX_WIDTH, // key_width
    sizeof(CryptoTupleOut), // value_width
    0,  // format 0 == {value}
    0   // internal
) {
   IndexTupleIn request;
   CryptoTupleOut response;

   method send_request     = { key      = request }
   method receive_response = { response = value }
}


// ************************************************


class DecryptSeqNum :: UserEngine(1, 0) {
   SpdSadTupleIn    sad_tuple;
   SeqNumTupleIn    seq_tuple;
   IndexTupleOut    index_tuple;
   SeqCheckTupleOut check_tuple;
}


// ************************************************

class AES_CTR :: UserEngine(20, 0) {
    Packet_input            packet_in;
    Packet_output           packet_out;

    PadTupleIn              pad_tuple;
    CryptoTupleIn           key_tuple;
    InitVectorTupleIn       iv_tuple;

    AesTupleOut             aes_tuple;
}


// ************************************************


class checksum :: TupleEngine(3, state1) {
  class internal :: Tuple {
    struct{
      hc_inv : 17,
      m_p    : 17,
      m_inv  : 17,
      result : 17
    }
  }
  class checksum_ipv4_tup_in :: Tuple(in) {
    struct{
```

```
        ttl : 8,
        protocol : 8,
        checksum : 16
      }
    }
    class checksum_ipv4_tup_out :: Tuple(out) {
      struct{
        header : 32
      }
    }
    checksum_ipv4_tup_in ipv4_tup_in;
    checksum_ipv4_tup_out ipv4_tup_out;
    internal vars;

    class state1 :: Section(1) {
        method update = {
      vars.hc_inv = ~ipv4_tup_in.checksum,
      vars.m_p = (ipv4_tup_in.ttl << 8) | ipv4_tup_in.protocol
        }
        method move_to_section = state2;
    }
    class state2 :: Section(2) {
        method update = {
      vars.m_inv = ~(vars.m_p + 0x0100),
      vars.result = vars.hc_inv + vars.m_inv + vars.m_p
        }
        method move_to_section = state3;
    }
    class state3 :: Section(3) {
        method update = {
            ipv4_tup_out.header = ((vars.result & 0xffff) << 16) | ((vars.result &
0x10000) >> 16)
        }
        method move_to_section = done(0);
     }
}
```

Current Implementation Restrictions

There are a small number of implementation limitations in the PX compiler included in SDNet 2017.4 which restrict the full breadth of the PX language. Use of the restricted features is flagged as an error by the compiler.

These restrictions are:

- Offset increments and absolute values, and remove and insert sizes, must be multiples of eight bits.
- In a value expression, multiplication (*) is only allowed when (a) both operands are constants or (b) one operand is constant zero or (c) one operand is a constant power of two.
- In a value expression, division (/) is only allowed when (a) both operands are constants or (b) the dividend is constant zero or (c) the divisor is a constant power of two.
- In a value expression, modulus (%) is only allowed when (a) both operands are constants or (b) the dividend is constant zero or (c) the divisor is a constant power of two.
- Verilog reserved words cannot be used as PX program identifiers.
- Constant expressions (or const identifiers defined using an expression) cannot be used in the parameter lists for engine subclass declarations.

There is one area where the semantics of PX might not be completely respected by the PX compiler:

- The scope over which expression widths are maximized (see Value Expressions, page 19 for more information).

The compiler can only generate synthesizable RTL code for the architecture of a PPP instance, and cannot change firmware for an existing PPP instance.'

Appendix C

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open the Xilinx Documentation Navigator (DocNav):

- From the Vivado® IDE, select **Help > Documentation and Tutorials**.
- On Windows, select **Start > All Programs > Xilinx Design Tools > DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In the Xilinx Documentation Navigator, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

Note: For more information on Documentation Navigator, see the Documentation Navigator page on the Xilinx website.

References

SDNet Packet Processor User Guide (UG1012)

Please Read: Important Legal Notices