Functional Bus Description Language

Revision 0.0

19 January 2026

Abstract

This document is the official specification of the Functional Bus Description Language. Its primary purpose is to define the syntax and semantics of the language. Functional Bus Description Language is a domain-specific language for bus and register management. Its main characteristic is the paradigm shift from the register-centric approach to the functionality-centric approach. In the register-centric approach, the user defines registers and then manually lays out the data into the registers. In the functionality-centric approach, the user defines the functionality of the data, and the registers, module hierarchy, and access codes are later automatically inferred. By defining the functionality of the data placed in the registers, it is possible to generate more code, increase code robustness, improve system design readability, and shorten the implementation process.

Contents

Participants

Michał Kruszewski, *Chair*, *Technical Editor*, mkru@protonmail.com

Glossary

Not all terms defined in the glossary list are used in the specification. Some of them are formally defined because they are helpful when discussing, for example, compiler implementation.

call register

The proc register with the associated call pulse signal. When the call register is written, the call pulse is generated.

compiler back-end

The part of a compiler responsible for taking registerification result and generating provider or requester firmware, gateware, or software.

compiler front-end

The part of a compiler responsible for parsing files with functional bus description and carrying out the registerification process.

data

The data term is used to refer to the content of the registers. Unless it is used in the context of internal data types of the language.

downstream

A stream from the requester to the provider.

dynamic generator

A generator capable of loading registerification results and producing access code during target system runtime.

exit register

The proc register with the associated exit pulse signal. When the exit register is read, the exit pulse is generated.

empty proc

A procedure without parameters and returns.

functionality

The functionality of given data. It can be seen as a type of the data. In case of functionalities encapsulating other functionalities, such as bus, block, proc or downstream, the functionality is used to denote a broader context of encapsulated data.

gap

Unused bits within a register.

gateware

The overall configuration of the logic placed in the FPGA to make it behave according to the desired description. The term is not formally defined anywhere, however it is used to unburden the firmware term. IEEE Std 610.12-1990 also mentions that the firmware term is too overloaded and confusing.

generator

A software, usually a part of a compiler, directly responsible for the access code generation based on the registerification results.

information

The metadata on the functionality data. The metadata describes where the data is located, for example bit masks and register addresses, and how to access the data.

means

The automatically generated method or data that shall be used by the requester to request particular functionality. A means in particular programming language is usually a function, method or procedure that shall be called or class, dictionary, map or structure containing information on how to access particular functionality.

mixed proc

A procedure with both, parameters and returns.

param proc

A procedure with parameters only.

provider

The system component containing the generated registers and providing described functionalities.

pure call register

The proc call register containing no proc returns.

pure exit register

The proc exit register containing no proc params.

register

A logical set of storage elements which content can be read or written by providing a single address. The set can be empty when the number of physical storage elements equals 0.

registerification

The process of placing data of functionalities into the registers. The process includes assigning data bit masks, register addresses as well as block addresses and masks. The term is new in the field and is coined in the specification.

requester

The system component accessing the generated registers and requesting described functionalities.

return proc

A procedure with returns only.

static generator

A generator producing access code that is afterward processed by a proper language toolchain or read by another software before being used.

strobe register

The stream register with the associated strobe pulse signal. When the strobe register is written (downstream), or read (upstream) the strobe pulse is generated.

target

The transpilation target. For example, a target can be a requester Python code allowing to access functionalities of the provider in an asynchronous fashion. A VHDL code providing description of the functionality registers and exposing AXI compliant interface is a valid provider target. A JSON file describing registerification results is for example a valid documentation target. The target depends on several factors, but the most important ones are programming/description language, synchronous or asynchronous access interface, bus type, dynamic or static address map reloading. Each target has its recipient. It is either provider, requester or documentation.

upstream

A stream from the provider to the requester.

1 Overview

1.1 Scope

This document specifies the syntax and semantics of the Functional Bus Description Language (FBDL).

1.2 Purpose

This document is intended for the implementers of tools supporting the language and for users of the language. The focus is on defining the valid language constructs, their meanings and implications for the hardware and software that is specified or configured, how compliant tools are required to behave, and how to use the language.

1.3 Motivation

Describing and managing registers can be a tedious and error-prone task. The information about registers is utilized by software, hardware, and verification engineers. Typically a specification of the registers is designed by the hardware designer or system architect. During the design and implementation phases, it changes multiple times due to different reasons such as bugs, requirement changes, technical limitations, or user feedback. A simple change in a single register may imply adjustments in both hardware and software. These adjustments cost money and time.

Several formal and informal tools exist to address issues related to register management. However, they all share the same concept of describing registers at a very low level. That is, the user has to implicitly define the layout of the registers. For example, in the case of a register containing multiple statuses, its the user responsibility to specify the bit position for every status.

The FBDL is different in this term. The user specifies the functionalities that must be provided by the data stored in the registers. The register layout is automatically generated based on the functional requirements. Such an approach increases the amount of automatically generated hardware description and software code and decreases the amount of code requiring manual implementation compared to the register-centric approach. Not only the register masks, addresses, and single read and write functions can be generated, but complete custom functions with optimized access methods. This, in turn, leads to shorter design iterations and fewer bugs.

1.4 Word usage

The terms "must", "must not", "required", "shall", "shall not", "should", "should not", "recommended", "may", and "optional" in this document are to be interpreted as described in the IETF Best Practices Document 14, RFC 2119.1.

1.5 Syntactic description

The formal syntax of the FBDL is described by means of context-free syntax using a simple variant of the Backus-Naur Form (BNF). In particular:

a) Lowercase words in `constant-width` font, some containing embedded underscores, are used to denote syntactic categories, for example:

    `single_import_statement`

Whenever the name of a syntactic category is used, apart from the syntax rules themselves, underscores are replaced with spaces thus, "single import statement" would appear in the narrative description when referring to the syntactic category.

b) Boldface words are used to denote keywords, for example:

    mask

Keywords shall be used only in those places indicated by the syntax.

c) A production consists of a left-hand side, the symbol ":::=" (which is read as can be replaced by), and a right-hand side. The left-hand side of a production is always a syntactic category, the right-hand side is a replacement rule. The meaning of a production is a textual-replacement rule. Any occurrence of the left-hand side may be replaced by an instance of the right-hand side.

d) A vertical bar (|) separates alternative items on the right-hand side of a production unless it occurs immediately after an opening brace, in which case it stands for itself, for example:

```
decimal_digit ::= zero_digit | non_zero_decimal_digit
choices ::= choice { | choice }
```

In the first instance, an occurrence of decimal digit can be replaced by either zero digit or non zero decimal digit. In the second case, "choices" can be replaced by a list of "choice", separated by vertical bars, see item f) for the meaning of braces.

e) Square brackets [ ] enclose optional items on the right-hand side of a production. Note, however, sometimes square brackets in the right-hand side of the production are part of the syntax. In such cases bold font is used.

f) Braces { } enclose a repeated item or items on the right-hand side of a production. The items may appear zero or more times.

g) The term *declared identifier* is used for any occurrence of an identifier that already denotes some declared item (declared by a user or by specification, for example, built-in function name).

2 References

The following referenced documents are indispensable for the application of this document (i.e., they must be understood and used, so each referenced document is cited in the text and its relationship to this document is explained). For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or corrigenda) applies.

- IETF Best Practices Document 14, RFC 2119,
- IETF UTF-8, a transformation format of ISO 10646, RFC 3629,
- IEEE Std 754™–2019, IEEE Standard for Floating-Point Arithmetic.

3 Concepts

The core concept behind the FBDL is based on the fact that if there is a system part with the registers that can be accessed, then there is at least one more system part accessing these registers. The part accessing the registers is called the *requester*. The part containing the registers is called the *provider*, as it provides functions via particular functionalities.

The code generated from the FBDL description can be conceptually divided into two parts, the requester part and the provider part. The requester code usually refers to the generated software or firmware implemented in typical programming languages such as: Ada, C, C++, Go, Java, Python, Rust etc. The provider code usually refers to the generated gateware or hardware implemented in hardware description languages or frameworks such as: VHDL, SystemVerilog, SystemC, Bluespec, PipelineC, MyHDL, Chisel etc. However, implementing the provider, for example as a firmware, using the C language and a microcontroller is practically doable and valid.

The description of functionalities shall be placed in files with the `.fbd` extension. By default, the bus named `"main"` is the entry point for the description used for the code generation. A compiler is free to support a parameter for changing the name of the main bus.

```
description ::=
  import_statement |
  constant_definition |
  type_definition |
  instantiation
```

3.1 Properties

Each data in the FBDL description has associated functionality and each functionality has associated properties. Properties allow the configuration of functionalities. Each property must have a concrete type. The default value of each property is specified in the round brackets ( ) in the functionality subsections. If the default value is `bus width`, then the default value equals the actual value of the bus `width` property. If the default value is `uninitialized`, then it shall be represented as the uninitialized meta value at the provider side. If the target language for the provider code does not have a concept of uninitialized value, then values such as 0, `Null`, `None`, `nil` etc. shall be used.

Each property either defines or declares functionality feature or behavior. Definitive properties specify the desired behavior of the automatically generated code. They specify elements directly managed by the FBDL. Examples of definitive properties include `atomic` or `width` properties. Declarative properties describe the behavior of external elements that automatically generated code only interacts with. Declarative properties are required to generate valid logic, and it is the user's responsibility to make sure their values match the behavior of external components. Examples of declarative properties include `access` or `in-trigger` properties.

```
property_assignment ::= property_identifier = expression
```

```
property_assignments ::=
  property_assignments
  { ; property_assignment }
  newline
```

```
semicolon_and_property_assignments ::= ; property_assignments
```

```
property_identifier ::=
  access | add-enable | atomic | byte-write-enable | clear | delay |
  enable-init-value | enable-reset-value | groups | init-value |
  in-trigger | masters | out-trigger | range | read-latency |
  read-value | reset | reset-value | size | width
```

3.2 Instantiation

A functionality can be instantiated in a single line or in multiple lines.

```
instantiation ::= single_line_instantiation | multi_line_instantiation
```

```
single_line_instantiation ::=
  identifier
  [ array_marker ]
  declared_identifier | qualified_identifier
  [ argument_list ]
  newline | semicolon_and_property_assignments
```

```
multi_line_instantiation ::=
  identifier
  [ array_marker ]
  declared_identifier | qualified_identifier
  [ argument_list ]
  functionality_body
```

```
array_marker ::= [ expression ]
```

```
functionality_body ::=
  newline
  indent
  { constant_definition | type_definition | property_assignments | instantiation }
  dedent
```

The following code shows examples of single line instantiations:

```
C config
C config; width = 8
M [8]mask; atomic = false; width = 128; init-value = 0
err error_t(48); atomic = false
```

The following code shows examples of multi line instantiations:

```
My_Config config
  width = 96
  atomic = false
My_Irq irq
  add-enable = true
  in-trigger = "Edge"
```

3.3 Addressing

The FBDL specification does not impose any addressing mode, for example, byte or word. There is also no property allowing to switch between different addressing modes. The handling of the addressing mode is left to the particular compiler implementation.

Depending on the use case and target bus, at least a few approaches are possible.

1. Suppose the compiler has a monolithic structure (no distinction between the compiler front-end and back-end). In that case, it is best to use the addressing mode required or preferred by the target bus (for example, byte addressing for AXI or word addressing for Wishbone).
2. Suppose the compiler is just a generic front-end:

- The compiler might provide command line flag or parameter to specify the addressing mode during the compiler call.
- The compiler might assume byte addressing. The drawback of this approach is that the data bus width must be a multiple of the number of bits in a byte. In such a case, word addressing is obtained by simply discarding a proper number of the least significant bits of the address bus. The address bus shift operation is carried out by the compiler back-end.
- The compiler might assume word addressing. The advantage of this approach is that there is no extra restriction on the data bus width when the back-end utilizes word addressing. If the back-end utilizes byte addressing, the address is obtained by adding a proper number of the least significant bits to the address bus. The address bus shift operation is carried out by the compiler back-end. The back-end must also ensure that the data bus width has the proper value.

3.4 Positive logic

The FBDL uses only positive logic. An active level in positive logic is a high level (binary 1), and an active edge is a rising edge (transition from the low level to the high level, from binary 0 to binary 1). It does not mean that FBDL cannot be used with external components using negative logic. To connect external negative logic components to the generated FBDL positive logic components, one shall negate the signals at the interface connection level. Supporting both positive and negative logic would unnecessarily complex the language and would create a second way for solving the same problem making the set of possible solutions non-orthogonal.

3.5 Domain-specific language

The FBDL is a domain-specific language with its own syntax. Some of the register-centric tools are built on top of standard file formats or markup languages such as JSON, TOML, XML or YAML. Such an approach allows for fast prototyping and has a lower entry threshold. However, it becomes a burden when more conceptually advanced features, for example parametrization, have to be supported. The description quickly begins to gain in volume, and the overall feeling is it is needlessly verbose. What is more, having its own adjusted language syntax allows for more informative compiler error messages.

4 Lexical elements

FBDL has following types of lexical tokens:

- comment,
- identifier,
- indent,
- keyword,
- literal,
- newline.

4.1 Comments

There is only a single type of comment, a *single-line comment*. A single-line comment starts with the '#' character and extends up to the end of the line. A single-line comment can appear on any line of an FBDL file and may contain any character, including glyphs and special characters. The presence or absence of comments has no influence on whether a description is legal or illegal. Their sole purpose is to enlighten the human reader.

4.1.1 Documentation comments

Documentation comments are comments that appear immediately before constant definitions, type definitions, and functionality instantiations with no intervening newlines. The following code shows examples of documentation comments:

```
# Number of receivers
const RECEIVERS_COUNT = 7
main bus
  # Data receivers
  receivers [RECEIVERS_COUNT]block
    # 0 disable receiver, 1 enable receiver
    enable config; width = 1
    # Number of frames in the buffer
    frame_count status
    # Documentation comments can consist of
    # multiple single-line comments.
    read_frame proc
      data [4]return; width = 8
```

4.2 Identifiers

Identifiers are used as names. An identifier shall start with a letter.

```
uppercase_letter ::=
  A | B | C | D | E | F | G | H | I | J | K | L | M |
  N | O | P | R | S | T | U | V | W | X | Y | Z

lowercase_letter ::=
  a | b | c | d | e | f | g | h | i | j | k | l | m |
  n | o | p | r | s | t | u | v | w | x | y | z

letter ::= uppercase_letter | lowercase_letter

letter_or_digit ::= letter | decimal_digit

identifier ::= letter { underscore | letter_or_digit }
```

The following code contains some valid and invalid identifiers:

```
const C_20 = 20 # Valid
const _C20 = 20 # Invalid
main bus
  cfg1 config # Valid
  1cfg config # Invalid
```

4.2.1 Declared identifier

Declared identifier is used for any occurrence of an identifier that already denotes some declared item.

```
declared_identifier ::= letter { underscore | letter_or_digit }
```

4.2.2 Qualified identifier

The qualified identifier is used to reference a symbol from foreign package.

```
qualified_identifier ::= declared_identifier.declared_identifier
```

The first declared identifier denotes the package, and the second one denotes the symbol from this package.

4.3 Indent

The indentation has semantics meaning in the FBDL. The indent sequence consists of two space characters (U+0020). It is hard to express the indent and dedent using BNF. Ident is the increase of the indentation level, and dedent is the decrease of the indentation level. In the following code the indent happens in the lines number 2, 5 and 7, and the dedent happens in the line number 4. What is more, double dedent happens at the EOF. The number of indents always equals the number of dedents in the syntactically and semantically correct file.

```
1: type cfg_t config
2:   atomic = false
3:   width = 64
4: main bus
5:   c cfg_t
6:   blk block
7:     c cfg_t
8:     s status
```

Not only the indent alignment is important, but also its level. In the following code the first type definition is correct, as the indent level for the definition body is increased by one. The second type definition is incorrect, even though the indent within the definition body is aligned, as the indent level is increased by two.

```
# Valid indent
type cfg1_t config
  atomic = false
  width = 8
# Invalid indent, indent increased by two
type cfg2_t config
    atomic = false
    width = 8
```

4.4 Keywords

FBDL has following keywords: **atomic**, **block**, **bus**, **clear**, **config**, **const**, **false**, **import**, **init-value**, **irq**, **mask**, **param**, **proc**, **range**, **reset**, **read-value**, **reset-value**, **return**, **static**, **stream**, **true**, **type**, **in-trigger**, **out-trigger**.

Keywords can be used as identifiers with one exception. Keywords denoting built-in types (function-alities) cannot be used as identifiers for custom types.

4.5 Literals

4.5.1 Bool literals

```
bool_literal ::= false | true
```

4.5.2 Number literals

```
underscore ::= _
zero_digit ::= 0
non_zero_decimal_digit ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
decimal_digit ::= zero_digit | non_zero_decimal_digit
binary_base ::= 0B | 0b
binary_digit ::= 0 | 1
octal_base ::= 0O | 0o
octal_digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
hex_base ::= 0X | 0x
hex_digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  | A | a | B | b | C | c | D | d | E | e | F | f
```

4.5.3 Integer literals

```
integer_literal ::= binary_literal | octal_literal | decimal_literal | hex_literal
binary_literal ::= binary_base binary_digit { [ underscore ] binary_digit }
octal_literal ::= octal_base octal_digit { [ underscore ] octal_digit }
decimal_literal ::= non_zero_decimal_digit { [ underscore ] decimal_digit }
hex_literal ::= hex_base hex_digit { [ underscore ] hex_digit }
```

4.5.4 Real literals

The real literals shall be represented as described by IEEE Std 754, an IEEE standard for double-precision floating-point numbers.

Real numbers can be specified in either decimal notation (for example, 17.83) or in scientific notation (for example, 13e8, which indicates 13 multiplied by 10 to the eighth power). Real numbers expressed with a decimal point shall have at least one digit on each side of the decimal point.

4.5.5 String literals

A string literal is a sequence of zero or more UTF-8 characters enclosed by double quotes ("").

```
string_literal ::= "{UTF-8 character}"
```

4.5.6 Bit string literals

A bit string literal is a sequence of zero or more digit or meta value characters enclosed by double quotes ("") and preceded by a base specifier. The meta value characters are supported because of hardware description languages, that also have a concept of metalogical values.

```
meta_character ::= - | U | W | X | Z
```

The meta characters have following meaning:

- '-' - don't care,
- 'U' - uninitialized,
- 'W' - weak unknown,
- 'X' - unknown,
- 'Z' - high-impedance state.

```
binary_or_meta ::= binary_digit | meta_character
```

```
octal_or_meta ::= octal_digit | meta_character
```

```
hex_or_meta ::= hex_digit | meta_character
```

There are three types of bit string literals: binary bit string literal, octal bit string literal and hex bit stirng literal.

```
bit_string_literal ::=
  binary_bit_string_literal |
  octal_bit_string_literal |
  hex_bit_string_literal
```

```
binary_bit_string_base = B | b
```

```
binary_bit_string_literal = binary_bit_string_base "{binary_or_meta}"
```

```
octal_bit_string_base = O | o
```

```
octal_bit_string_literal = octal_bit_string_base "{octal_or_meta}"
```

```
hex_bit_string_base = X | x
```

```
hex_bit_string_literal = hex_bit_string_base "{hex_or_meta}"
```

If meta value is present in a bit string literal, then it is expanded to the proper width depending on the bit string base. For example, following equations are true:

```
  o"XW" = b"XXXWWW"
  x"U-" = b"UUUU----"
```

4.5.7 Time literals

A time literal is a sequence of integer literal and a time unit.

```
time_unit ::= ns | us | ms | s
```

```
time_literal ::= integer_literal time_unit
```

Time literals are used to create values of time data type, required for example by the delay property.

5 Data types

There are 7 data types in FBDL:

- bit string,
- bool,
- integer,
- range,
- real,
- string,
- time.

Types are implicit and are not declared. The type of the value evaluated from an expression must be checked before any assignment or comparison. If there is a type mismatch that can be resolved with implicit rules, then it shall be resolved. In case of a type mismatch that cannot be resolved, an error must be reported by the compiler.

Conversion from bool to integer in expressions is implicit. Conversion from integer to real in expressions is implicit. Conversion from integer to range can be implicit if the interger value is natural. Conversion from real to integer can be implicit if there is no fractional part. If fractional part is present, then conversion from real to integer must be explicit and must be done by calling any function returning integer type, for example `ceil()`, `floor()`.

The below picture presents a graph of possible implicit conversions between different data types.
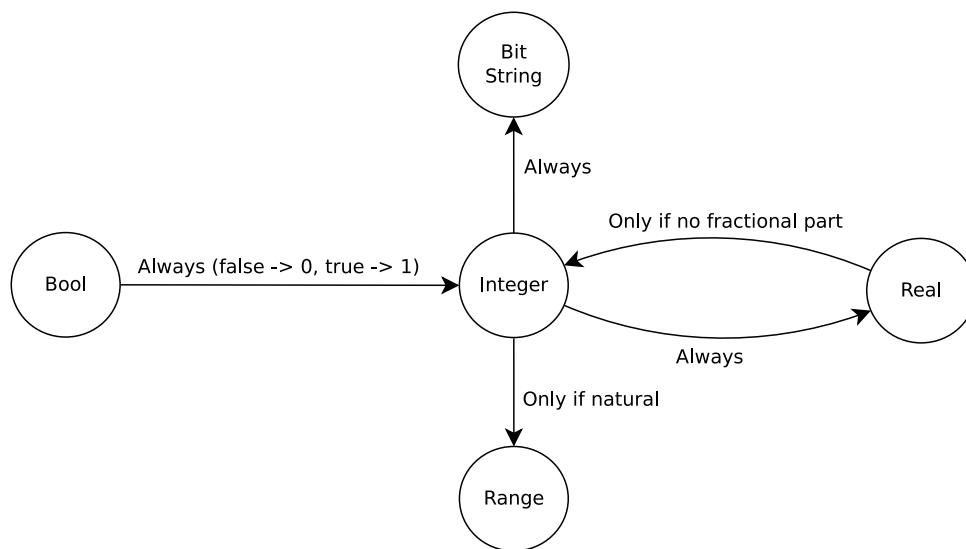
5.1 Bit string

The value of the bit string type is used for all **`*-value`** properties. It might be created explicitly using the bit string literal or it might be converted implicitly from the value of integer type. The only way to create a bit string value containing meta values is to explicitly use the bit string literal.

The below table presents unary negation operation results applied to possible bit string data type values.

Bit string unary bitwise negation

In Value	Out Value
0	1
1	0
-	-
U	U
W	W
X	X

Below tables present binary operation results applied to possible bit string data type values.

Bit string binary bitwise and (&) resolution

Operands	0	1	-	U	W	X	Z
0	0	0	0	U	0	X	0
1	0	1	1	U	1	X	1
-	0	1	-	U	W	X	Z
U	U	U	U	U	U	U	U
W	0	1	X	U	W	X	W
X	X	X	X	U	X	X	X
Z	0	1	X	U	W	X	Z

Bit string binary bitwise or (|) resolution

Operands	0	1	-	U	W	X	Z
0	0	1	0	U	0	X	0
1	1	1	1	U	1	X	1
-	0	1	-	U	W	X	Z
U	U	U	U	U	U	U	U
W	0	1	X	U	W	X	W
X	X	X	X	U	X	X	X
Z	0	1	X	U	W	X	Z

Bit string binary bitwise xor (^) resolution

Operands	0	1	-	U	W	X	Z
0	0	1	0	U	0	X	0
1	1	0	1	U	1	X	1
-	0	1	-	U	W	X	Z
U	U	U	U	U	U	U	U
W	0	1	X	U	W	X	W
X	X	X	X	U	X	X	X
Z	0	1	X	U	W	X	Z

5.2 Bool

The value of the bool type can be created explicitly using `true` or `false` literals. The value of the bool type shall be implicitly converted to the value of the integer type in places where the value of the integer type is required. The boolean `false` value shall be converted to the integer value 0. The boolean `true` value shall be converted to the integer value 1. In the following example, the value of `I1` evaluates to 1, and the value of `I2` evaluates to 2.

```
const B0 = false
const B1 = true
const I1 = B0 + B1
const I2 = B1 + B1
```

The bool - integer conversion is asymmetric. Implicit conversion of a value of the integer type to a value of the bool type is forbidden. This is becuase values of the bool type are often used to count the number of elements or to arbitrarily enable/disable an element generation. However, a value of the integer type appearing in a place where a value of the bool type is required is usually a sign of a mistake. To convert a value of the integer type to a value of the bool type the built-in `bool()` function must be called.

5.3 Integer

The integer data type is always signed integer and must be at least 64 bits wide.

5.4 Range

The range data type is used to assign value to the `range` property. It consists of left and right bounds, both expressed as expressions, separated by the colon character. The range data type can be created solely using the range expression. There is no range literal.

```
range_expression ::= expression : expression
```

5.5 Real

The real data type is 64 bits IEEE 754 double precision floating-point type.

5.6 String

The string data type can only be created explicitly using a string literal. The string data type is only used for setting values of some properties, for example `access`.

5.7 Time

The time data type is only used for assigning value to the properties expressed in time. The value of time type can be created explicitly using the time literal. Values of time type can be added regardless of their time units. Values of the time type can also be multiplied by values of the integer type. All of the below property assignments are valid.

```
delay = 1 s + 1 ms + 1 us + 1 ns
delay = 5 * 60 s # Sleep for 5 minutes.
delay = 10 ms * 4 + 7 * 8 us
```

6 Expressions

An expression is a formula that defines the computation of a value by applying operators and functions to operands.

```
expression ::=
  bool_literal |
  integer_literal |
  real_literal |
  string_literal |
  bit_string_literal |
  time_literal |
  declared_identifier |
  qualified_identifier |
  unary_operation |
  binary_operation |
  function_call |
  subscript |
  parenthesized_expression |
  range_expression |
  expression_list
```

The function call is used to call one of built-in functions.

```
function_call ::= declared_identifier( [ expression { , expression } ] )
```

The subscript is used to refer to a particular element from the expression list.

```
subscript ::= declared_identifier[ expression ]
```

The parenthesized expression may be used to explicitly set order of operations.

```
parenthesized_expression ::= ( expression )
```

The expression list may be used to create a list of expressions.

```
expression_list ::= [ [ expression { , expression } ] ]
```

6.1 Operators

6.1.1 Unary Operators

```
unary_operation ::= unary_operator expression
```

```
unary_operator ::= unary_arithmetic_operator | unary_bitwise_operator
```

```
unary_arithmetic_operator ::= -
```

```
unary_bitwise_operator ::= !
```

FBDL unary operators

Token	Operation	Operand Type	Result Type
-	Opposite	Integer Real	Integer Real
!	Negation	Bit string Integer	Bit string Integer

6.1.2 Binary Operators

```
binary_operation ::= expression binary_operator expression
```

```
binary_operator ::=
  binary_arithmetic_operator |
  binary_bitwise_operator |
  binary_comparison_operator |
  binary_logical_operator |
  binary_range_operator

binary_arithmetic_operator ::= + | - | * | / | % | **

binary_bitwise_operator ::= << | >>

binary_comparison_operator ::= == | != | < | <= | > | >=

binary_logical_operator ::= && | ||

binary_range_operator ::= :
```

FBDL binary arithmetic operators

Token	Operation	Left Operand Type	Right Operand Type	Result Type
+	Addition	Integer	Integer	Integer
		Integer	Real	Real
		Real	Integer	Real
		Real	Real	Real
		Time	Time	Time
-	Subtraction	Integer	Integer	Integer
		Integer	Real	Real
		Real	Integer	Real
		Real	Real	Real
*	Multiplication	Integer	Integer	Integer
		Integer	Real	Real
		Real	Integer	Real
		Real	Real	Real
		Integer	Time	Time
		Time	Integer	Time
-	Division	Integer	Integer	Real
		Integer	Real	Real
		Real	Integer	Real
		Real	Real	Real
%	Remainder	Integer	Integer	Integer
**	Exponentiation	Integer	Integer	Integer
		Integer	Real	Real
		Real	Integer	Real
		Real	Real	Real

FBDL binary bitwise operators

Token	Operation	Left Operand Type	Right Operand Type	Result Type
<<	Left Shift	Integer	Integer	Integer
>>	Right Shift	Integer	Integer	Integer
&	And	Bit String Integer	Bit String Integer	Bit String Integer
|	Or	Bit String Integer	Bit String Integer	Bit String Integer
^	Xor	Bit String Integer	Bit String Integer	Bit String Integer

FBDL binary comparison operators

Token	Operation	Left Operand Type	Right Operand Type	Result Type
==	Equality	Integer Integer Real Real	Integer Real Integer Real	Bool Bool Bool Bool
!=	Nonequality	Integer Integer Real Real	Integer Real Integer Real	Bool Bool Bool Bool
<	Less Than	Integer Integer Real Real	Integer Real Integer Real	Bool Bool Bool Bool
<=	Less Than or Equal	Integer Integer Real Real	Integer Real Integer Real	Bool Bool Bool Bool
>	Greater Than	Integer Integer Real Real	Integer Real Integer Real	Bool Bool Bool Bool
>=	Greater Than or Equal	Integer Integer Real Real	Integer Real Integer Real	Bool Bool Bool Bool

FBDL binary logical operators

Token	Operation	Left Operand Type	Right Operand Type	Result Type
&&	Short-circuiting logical AND	Bool	Bool	Bool
||	Short-circuiting logical OR	Bool	Bool	Bool

FBDL binary range operator

Token	Operation	Left Operand Type	Right Operand Type	Result Type
:	Range	Integer	Integer	Range

The bool data type is not valid operand type for the most of the binary operations. However, as there is the rule for implicit conversion from the bool data type to the integer data type, all operations accepting the integer operands work also for the bool operands.

6.2 Functions

The FBDL does not allow defining custom functions for value computations. However, FBDL has following built-in functions:

abs(x integer|real) integer|real

The abs function returns the absolute value of x.

bool(x integer) bool

The bool function returns a value of the bool type converted from a value x of the integer type. If x equals 0, then the false is returned. In all other cases the true is returned.

ceil(x real) integer

The ceil function returns the least integer value greater than or equal to x.

floor(x real) integer

The floor function returns the greatest integer value less than or equal to x.

log2(x real) integer|real

The log2 function returns the binary logarithm of x.

log10(x real) integer|real

The log10 function returns the decimal logarithm of x.

log(x, b real) integer|real

The log function returns the logarithm of x to the base b.

u2(x, w integer) integer

The u2 function returns two's complement representation of x as an integer assuming width w. For example, u2(-1, 8) returns 255.

7 Functionalities

Functionalities are the core part of the FBDL. They define the capabilities of the provider. Each functionality is distinct and unambiguously defines the provider behavior and the interface that must be generated for the requester. There are following 13 functionalities:

1. blackbox,
2. block,
3. bus,
4. config,
5. group,
6. irq,
7. mask,
8. param,
9. proc,
10. return,
11. static,
12. status,
13. stream.

Sections of this chapter describe all available functionalities. Functionality properties are defined in the following notation.

property-name expected value type (default value) {kind}

Due to implicit data type conversion rules, some properties accept more than one data type. For example, the range property expects a value of the range data type. However, as there is a rule for implicit conversion from the integer type to the range type, the "range = 5" assignment is also valid.

The property kind indicates whether the property is a definitive or a declarative property as described in Section 3.1.

7.1 Blackbox

The blackbox functionality is used to incorporate blocks implemented manually or generated by external tools. For example, a user may want to explicitly manage some particular registers' layouts. In such a case, a register-centric tool might be used, and the generated block can be incorporated into the wrapping functionality-centric description using the blackbox functionality.

The blackbox functionality has following properties:

access string ("*Read Write*") {declarative}

> The access property declares the valid access permissions to the blackbox for the requester. Valid values of the access property are: "*Read Write*", "*Read Only*", and "*Write Only*".

size integer (obligatory) [definitive]

> The size property defines size of the blackbox in the number of words with width equal to the width property value of the block in which blackbox is defined.

The code generated for the requester must provide means for single read/write and block read/write transactions. Whether access means for cyclic or vectored (scatter-gather) transactions are automatically generated is up to the compiler. If blackbox is read-only or write-only, then an unsupported write or read access code is recommended not to be generated. The code generated for the provider must provide a means to connect the blackbox to the remaining part of the bus generated by an FBDL compiler.

7.2 Block

The block functionality is used to logically group or encapsulate functionalities. The block is usually used to separate functionalities related to particular peripherals such as UART, I2C transceivers, timers, ADCs, DACs etc. The block might also be used to limit the access for particular provider to only a subset of functionalities.

The block functionality has following properties:

align integer (inherited from bus align) {definitive}

  The align property defines the alignment of the block's address space. The default value is inherited from the bus align property value.

masters integer (1) {definitive}

  The masters property defines the number of block master ports. This property defines how many master ports shall be generated. However, it is up to the user how many master ports are connected in the design.

reset string (None) {definitive}

  The reset property defines the block reset type. By default the block has no reset. Valid values of the reset property are "*Sync*" for synchronous reset and "*Async*" for asynchronous reset.

The following example presents how to limit the scope of access for particular requester.

```
main bus
  c config
  blk block
    masters = 2
    s status
```

The logical connection of the system components may look as follows:
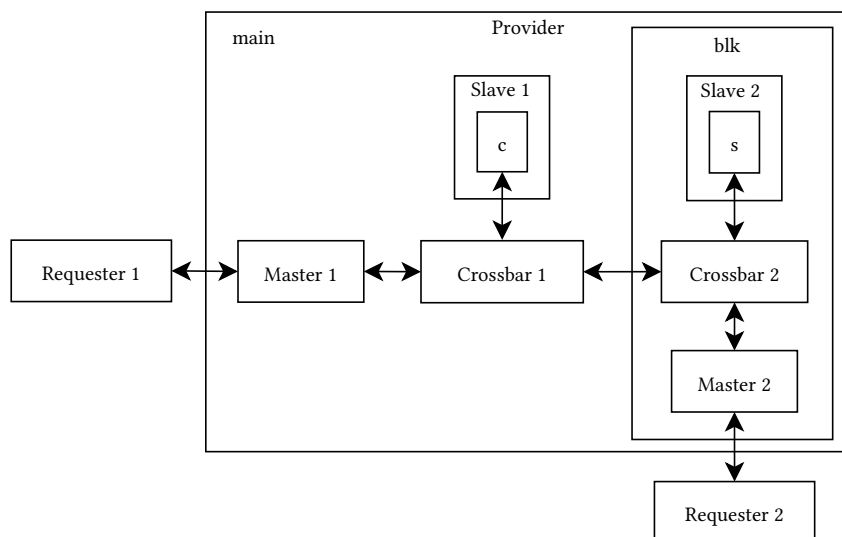
The requester number 1 can acces both config C and status S. However, the requester number 2 can access only the status S.

7.3 Bus

The bus functionality represents the bus structure. Every valid description must have at least one bus instantiated, as the bus is the entry point for the description used for the code generation.

The bus functionality has following properties:

align integer (0) {definitive}

The align property defines the alignment of blocks address space. The default value (0) means that no specific alignment is imposed.

By default, all blocks within the bus inherit the align property value from the bus align property. However, a specific block alignment can be modified by changing the align property value within the block.

A proper value of the align property might be required for some buses. For example, the AMBA AXI protocol specification requires that the address space of blocks (Subordinates) is aligned to 4 KB. In the case of 32-bit wide data bus, the align property should be set to 1024 (4096 / 32 * 8). However, any value being a multiple of 1024 is correct. A compiler back-end supporting the AMBA AXI generator should assert that the align property value is correct.

masters integer (1) {definitive}

The masters property defines the number of bus master ports. This property defines how many master ports shall be generated. However, it is up to the user how many master ports are connected in the design.

reset string (None) {definitive}

The reset property defines the bus reset type. By default the bus has no reset. Valid values of the reset property are "*Sync*" for synchronous reset and "*Async*" for asynchronous reset.

width integer (32) {definitive | declarative}

The width property defines the bus data width. Whether the width property is definitive or declarative depends on the compiler, more precisely on the compiler back-end. For example, if the compiler back-end supports only 16 bits data width, then the property is declarative, as it is up to the user to provide the correct width value. If the compiler back-end can handle multiple data bus widths, then the property is definitive.

The bus address width is not explicitly set, as it implies from the address space size required to pack all functionalities included in the bus description.

7.4 Config

The config functionality represents configuration data. The configuration data is data that is automatically read by the provider from its registers. As the config is automatically read by the provider, there is no need for an additional signal associated with the config, indicating the config write by the requester. By default, a config can be written and read by the requester.

The config functionality has following properties:

atomic bool (true) {definitive}

The atomic property defines whether an access to the config must be atomic. If atomic is true, then the provider must guarantee that any change of the config value, triggered by the requester write, is seen as an atomic change by the other modules of the provider. This is especially important when the config spans more than single register, as in case of single register write the change is always atomic.

init-value bit string (uninitialized) {definitive}

The init-value property defines the initial value of the config.

range range | [range] (None) {definitive}

The range property defines the range of valid values. Range bound values shall not be negative. This is because the FBDL makes no assumptions on the negative values encoding. To accomplish negative range checks functions such as u2 must be explicitly called. For example, following assignment limits the possible range from −16 to −8: `range = u2(-8, 8):u2(-16, 8)`. The range property shall not be explicitly set if the width property is already set. If the range property is not set, then the actual range implies from the width property. The code generated for the provider is not required to check or report if the value provided for the config write is within the valid range. The recommended way is to implement compiler parameter allowing enabling/disabling range check generation.

read-value `bit string (uninitialized) {definitive}`

The read-value property defines the value returned by the provider on the config read. If the read-value is not set, then the provider must return the actual value of the config.

reset-value `bit string (uninitialized) {definitive}`

The reset-value property defines the value of the config after the reset. If the reset-value is set, but a bus or block containing the config is not resettable (`reset = None`), then the compiler shall report an error.

width `integer (bus width) {definitive}`

The width property defines the bit width of the config. The width property shall not be explicitly set if the range property is already set.

The code generated for the requester must provide means for writing and reading the config.

7.5 Group

The group functionality represents a group of functionalities. It used to inform a compiler that particular functionalities might be accessed together, and their register location must meet additional constraints.

The group functionality has following properties:
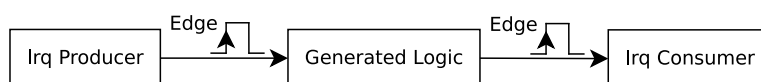
virtual `bool (false) {definitive}`

The virtual property defines whether the group is virtual. Virtual groups are used to group functionalities without generating the group interface for the requester code.

Not all functionalities can be grouped. Functionality grouping is described in detail in Section 10.
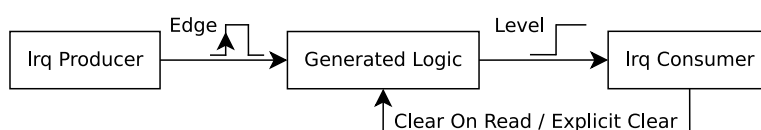
7.6 Irq

The irq functionality represents an interrupt handling. The irq functionality allows for automatic connection of the following interrupt producers (in-trigger) and consumers (out-trigger):
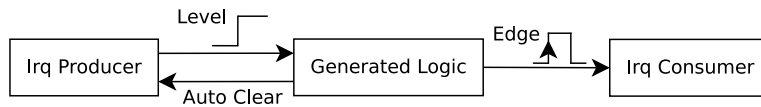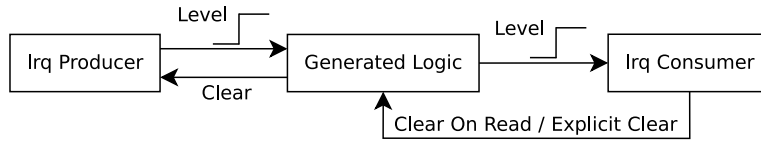
1) edge producer and edge sensitive consumer,

2) edge producer and level sensitive consumer,

3) level producer and edge sensitive consumer,

4) level producer and level sensitive consumer,

The irq functionality has following properties:

add-enable `bool (false) {definitive}`

The add-enable property defines whether an interrupt has associated enable bit in the interrupt enable register. The enable can be used to mask the interrupt.

clear `string ("Explicit") {definitive}`

The clear property defines how particular interrupt flag is cleared. The clear property is valid only in case of level-triggered interrupt consumer. If clear property is set for edge-triggered interrupt consumer a compiler shall report an error. Valid values are `"Explicit"` and `"On Read"`. The `"Explicit"` clear requires compiler to generate a means that must be explicitly used to clear the interrupt flag. The `"On Read"` clear requires the provider to clear the interrupt flag on each interrupt flag read.

enable-init-value `bit string (uninitialized) {definitive}`

The enable-init-value property defines the initial value of the enable bit in the interrupt enable register. The value must not exceed one bit. If add-enable is false and enable-init-value is set, then a compiler must report an error.

enable-reset-value `bit string (uninitialized) {definitive}`

The enable-reset-value property defines the value of the enable bit in the interrupt enable register after the reset. The value must not exceed one bit. If add-enable is false and enable-reset-value is set, then a compiler must report an error. If the enable-reset-value is set, but a bus or block containing the irq is not resettable (`reset = None`), then the compiler shall report an error.

in-trigger `string ("Level") {declarative}`

The in-trigger property declares the interrupt producer type of trigger. Valid values are `"Edge"` and `"Level"`. It is up to the user to make sure declared trigger is coherent with the actual producer behavior. A mismatch may lead to incorrect behavior.

out-trigger `string ("Level") {declarative}`

The out-trigger property declares the interrupt consumer type of trigger. Valid values are `"Edge"` and `"Level"`. It is up to the user to make sure declared trigger is coherent with the actual consumer requirement. A mismatch may lead to incorrect behavior.

7.7 Mask

The mask functionality represents a bit mask. The mask is data that is automatically read by the provider from its registers. By default, a mask can be written and read by the requester. The mask is very similar to the config. The difference is that the config is value-oriented, whereas the mask is bit-oriented. From the provider's perspective the mask and the config are the same. From the requester's perspective the code generated for interacting with the mask and the config is different.

The mask functionality has following properties:

atomic bool (true) {definitive}

> The atomic property defines whether an access to the mask must be atomic. If atomic is true, then the provider must guarantee that any change of the mask value, triggered by the requester write, is seen as an atomic change by the other modules of the provider. This is especially important when the mask spans more than single register, as in case of single register write the change is always atomic.

init-value bit string (uninitialized) {definitive}

> The init-value property defines the initial value of the mask.

read-value bit string (None) {definitive}

> The read-value property defines the value returned by the provider on the mask read. If the read-value is not set, then the provider must return the actual value of the mask.

reset-value bit string (None) {definitive}

> The reset-value property defines the value of the mask after the reset. If the reset-value is set, but a bus or block containing the mask is not resettable (reset = None), then the compiler shall report an error.

width integer (bus width) {definitive}

> The width property defines the bit width of the mask.

The code generated for the requester must provide means for setting, clearing and updating particular bits of the mask. The updating includes setting, clearing and toggling. The set differs from the update set. The set sets particular bits and simultaneously clears all remaining bits. The update set sets particular bits and keeps the value of the remaining bits. The clear differs from the update clear in an analogous way. The toggle always works on provided bits leaving the remaining bits untouched.

7.8 Param

The param functionality is an inner functionality of the proc and stream functionalities. It represents a data fed to a procedure or streamed by a downstream.

The param functionality has following properties:

range range | [range] (None) {definitive}

> The range property defines the range of valid values. The range property on param behaves exactly the same as the range property on config.

width integer (bus width) {definitive}

> The width property defines the bit width of the param.

The following example presents the definition of a downstream with three parameters.

```
Sum_Reduce stream
  type param_t param; width = 16
  a param_t
  b param_t
  c param_t
```

7.9 Proc

The proc functionality represents a procedure called by the requester and carried out by the provider. The proc functionality might contain param and return functionalities. Params are procedure parameters and returns represent data returned from the procedure.

The proc has associated signals at the provider side, the call signal and the exit signal. The call signal must be driven active for one clock cycle after all registers storing the parameters have been written. The exit signal must be driven active for one clock cycle after all registers storing the returns have been read. An empty proc (proc without params and returns) by default has only the call signal. However, if an empty proc has the delay property set, then it has both the call signal and the exit signal. A proc having only parameters has by default only the call signal. However, if a proc having only parameters has the delay property set, then it also has the exit signal. A proc having only returns has by default only the exit signal. However, if a proc having only returns has the delay property set, then it also has the call signal. The existence or absence of call and exit signals is summarized in the below table.

Proc call and exit signals occurrence

Delay Set	Empty	Only Params	Only Returns	Params & Returns
No	call	call	exit	call & exit
Yes	call & exit	call & exit	call & exit	call & exit

The proc functionality has following properties:

delay time (None) {definitive}

    The delay property defines the time delay between parameters write end and returns read start.

The code generated for the requester must provide a mean for calling the procedure.

7.10 Return

The return functionality is an inner functionality of the proc and stream functionalities. It represents data returned by a procedure or streamed by an upstream.

The return functionality has following properties:

width integer (bus width) {definitive}

    The width property defines the bit width of the return.

The following example presents the definition of a procedure returning four element byte array, and a single bit flag indicating whether the data is valid.

```
Read_Data proc
  data [4]return; width = 8
  valid return; width = 1
```

7.11 Static

The static functionality represents data, placed at the provider side, that shall never change.

The static functionality has following properties:

init-value bit string (obligatory) {definitive}

    The init-value property defines the initial value of the static.

read-value bit string (None) {definitive}

The read-value property defines the value that must be returned by the provider on the static read after the first read. If the read-value property is set, then the actual value of the static can be read only once.

reset-value bit string (None) {definitive}

The reset-value property defines the value of the static after the reset. If the reset-value is set, but a bus or block containing the static is not resettable (reset = None), then the compiler shall report an error. If both read-value and reset-value properties are set, then the static can be read one more time after the reset.

width integer (bus width) {definitive}

The width property defines the bit width of the static.

The static functionality may be used for example for versioning, bus id, bus generation timestamp or for storing secrets, that shall be read only once. Example:

```
Secret static
  width = 8
  init-value = 0x13
  read-value = 0xFF
```

7.12 Status

The status represents data that is produced by the provider and is only read by the requester.

The status functionality has following properties:

atomic bool (true) {definitive}

The atomic property defines whether an access to the status must be atomic. If atomic is true, then the provider must guarantee that any change of the status value is seen as an atomic change by the requester. This is especially important when a status spans more than single register, as in case of single register read the change is always atomic.

read-value bit string (None) {definitive}

The read-value property defines the value that must be returned by the provider on the status read after the first read. If the read-value property is set, then the actual value of the status can be read only once.

width integer (bus width) {definitive}

The width property defines the bit width of the status.

The code generated for the requester must provide a mean for reading the status.

7.13 Stream

The stream functionality represents a stream of data to a provider (downstream), or a stream of data from a provider (upstream). An empty stream (stream without any param or return) is always a downstream. It is useful for triggering cyclic action with constant time interval. A downstream must not have any return. An upstream shall not have any param, and must have at least one return.

The stream functionality is very similar to the proc functionality, but they are not the same. There are two main differences. The first one is that the stream must not contain both param and return. The second one is that the code for the stream, generated for the requester, shall take into account the fact that access to the stream is multiple and access to the proc is single. For example, lets consider the following bus description:

```
main bus
  p proc
    par param
  s stream
    par param
```

The code generated for the requester, implemented in the C language, might include following function prototypes:

```
int main_p(const uint32_t par);
int main_s(const uint32_t * par, size_t count);
```

The stream has associated strobe signal at the provider side. The strobe signal must be driven active for one clock cycle after all registers storing the parameters of a downstream have been written. It also must be driven active for one clock cycle after all registers storing the returns of an upstream have been read.

The stream functionality has following properties.

delay time (None) {definitive}

The delay property defines the time delay between writing/reading consecutive datasets for a downstream/upstream.

8 Parametrization

The FBDL provides the following three ways for description parametrization:

- constants,
- type definitions,
- type extending.

8.1 Constant

The constant represents a constant value. The value might be used in expression evaluations. The following code presents a bus description with three functionalities, all having the same array dimensions and width.

```
main bus
    const ELEMENT_COUNT = 4
    const WIDTH = 8
    c [ELEMENT_COUNT]config; width = WIDTH
    m [ELEMENT_COUNT]mask; width = WIDTH
    s [ELEMENT_COUNT]status; width = WIDTH
```

Constants defined inside bus or block functionalities must be included in the generated code, both for the provider and for the requester. This allows for having a single source of the constant value.

A constant can be defined in a single line in the single-line constant definition or as a part of the multi-constant definition.

```
single_constant_definition ::= const identifier = expression newline
```

Examples of single constant definition:

```
const WIDTH = 16
const FOO = 8 * BAR
const LIST = [1, 2, 3, 4, 5]
```

```
multi_constant_definition ::=
    const newline
    indent
    identifier = expression newline
    { identifier = expression newline }
    dedent
```

Examples of multi-constant definition:

```
const
    WIDTH = 16
    FOO = 8 * BAR
    LIST = [1, 2, 3, 4, 5]
const
    ONE = 1
    TWO = ONE + 1
    THREE = TWO + 1
```

8.2 Type definition

The type definition allows for defining custom functionalities. Any custom functionality resolves to one of the built-in functionalities. However, by defining custom functionality types it is possible to preset property values or to create easily parametrizable functionalities. The former leads to shorter descriptions and helps to avoid duplication.

```
type_definition ::= single_line_type_definition | multi_line_type_definition
```

```
single_line_type_definition ::=
  type
  identifier
  [ parameter_list ]
  [ array_marker ]
  declared_identifier | qualified identifier
  [ argument_list ]
  semicolon_and_property_assignments | newline

multi_line_type_definition ::=
  type
  identifier
  [ parameter_list ]
  [ array_marker ]
  declared_identifier | qualified identifier
  [ argument_list ]
  functionality_body

parameter_list ::= ( parameters )

parameters ::= parameter { , parameter }

parameter ::= identifier [ = expression ]
```

Parameters in the parameter list might have default values, but parameters with the default values must prepend parameters without default values in the parameter list.

```
argument_list ::= ( arguments )

arguments ::= argument { , argument }

argument ::= [ declared_identifier = ] expression
```

Arguments in the argument list may be prepended with the parameter name. However, arguments with parameter names must prepend arguments without parameter names in the argument list.

The below snippet presents examples of type definitions.

```
  # Single line type definition
  type cfg_t(w = 10) config; width = w; reset-value = 1

  # Multi line type definition
  type blk_t(with_status = true, mask_count) block
    S [with_status]status
    M [mask_count]mask

  main bus
    type irq_t irq; clear = "On Read"
    i1 irq_t
    i2 irq_t

    c1 cfg_t
    c2 cfg_t(6)
    c3 cfg_t(width = 8)

    blk1 blk_t(7)
    blk2 blk_t(with_status = false, mask_count = 11)
```

8.3 Type extending

The type extending allows extending any custom defined type, either by instantiation or by defining a new type. This is mainly, but not only, useful when there are similar blocks with only slightly different set of functionalities.

Example:

```
type blk_common_t block
  c1 config
  m1 mask
  s1 status
main bus
  blk_c blk_common_t
    c2 config
  blk_m blk_common_t
    m2 mask
  blk_s blk_common_t
    s2 status
```

This description is equivalent to the following description:

```
type blk_common_t block
  c1 config
  m1 mask
  s1 status
type blk_c_t blk_common_t
  c2 config
type blk_m_t blk_common_t
  m2 mask
type blk_s_t blk_common_t
  s2 status
main bus
  blk_c blk_c_t
  blk_m blk_m_t
  blk_s blk_s_t
```

The type nesting has no depth limit. However, no property already set in one of the ancestor types can be overwritten. Also no symbol identifier defined in one of the ancestor types can be redefined.

9 Scope and visibility

9.1 Import and package system

The FBDL has a concept of packages and allows importing packages into the file scope using the import statements. A package consists of files with `.fbd` extension placed in the same directory. A package directory name must start with the `"fbd-"` prefix. A package should have at least one file and shall not be placed in more than a single directory. A package is uniquely identified by its path. The name of a package is equivalent to the package directory name without the `"fbd-"` prefix. The package name `"main"` is reserved for the package consisting of a single file provided for the compiler as the compilation entry point.

Two packages may have the same name, but not the same path. For example, packages with following paths `foo/bar/fbd-uart` and `baz/zaz/fbd-uart` have the same name `"uart"`.

A package can be imported in a single line using the single-line import statement or as a part of the multi-import statement.

```
single_import_statement ::= import [ identifier ] string_literal
```

Examples of single import statement:

```
import "uart"
import spi "custom_spi"
```

```
multi_import_statement ::=
  import newline
  indent
  [ identifier ] string_literal
  { [ identifier ] string_literal }
```

Examples of multi import statement:

```
import
  "uart"
  spi "custom_spi"
```

The string literal is the path of the package. The path might not be complete, but shall be unambiguous. For example, if two paths are visible by the import statement ("*foo/bar/fbd-uart*" and "*baz/zaz/ fbd-uart*"), and both ends with "*fbd-uart*", then "*uart*" path is ambiguous, but "*bar/fbd-uart*" and "*zaz/fbd-uart*" are not.

The optional identifier is an identifier that shall denote the imported package within the importing file. If the identifier is omitted, then the implicit identifier for the package is the package name.

9.1.1 Package discovery

Each FBDL compiler is required to carry out the package auto-discovery procedure. The procedure consists of recursive checking of subdirectories of the following directories:
1) compiler working directory,
2) all directories defined in the `FBDPATH` environment variable,
3) additional directories provided to the compiler via custom parameters.

The below snippet presents a tree of an example working directory.

```
|-- externals
|    `-- timer
|        |-- fbd-timer
|        |    `-- timer.fbd
|        `-- gw
|             `-- timer.vhd
|-- fbd
|    |-- main.fbd
|    |-- fbd-i2c
|    |    `-- i2c.fbd
|    |-- tb
|    |    |-- i2c.fbd
|    |    `-- spi.fbd
|    `-- fbd-spi
|         `-- spi.fbd
|-- gw
|    |-- modules
|    |    |-- i2c.vhd
|    |    `-- spi.vhd
|    `-- top.vhd
`-- sw
     `-- run.py
```

In this case each FBDL compilant compiler must automatically discover following three packages:

- `i2c` - path `"./fbd/fbd-i2c"`,
- `spi` - path `"./fbd/fbd-spi"`,
- `timer` - path `"./externals/timer/fbd-timer"`.

9.2 Scope rules

The following elements define a new scope in the FBDL:

- package,
- type definition,
- functionality instantiation.

The following example presents all scopes.

```
const WIDTH = 16
main bus
  width = WIDTH
  const C20 = 20
  blk block
    const C30 = 30
    type cfg_t(WIDTH = WIDTH) config
      atomic = false
      width = WIDTH
    cfg16 cfg_t
    cfg20 cfg_t(C20)
    cfg30 cfg_t(C30)
```

The `WIDTH` constant has package scope, and it is visible at the package level, in the `main` bus instantiation and in the `Blk` block instantiation. It would also be visible in the `cfg_t` type definition. However, the `cfg_t` type has the parameter with the same name `WIDTH`. As a result, only the `WIDTH` parameter is visible within the type definition. The `WIDTH` parameter has a default value that equals `16`. This is because at this point the name `WIDTH` denotes the package level `WIDTH` constant. Type parameters are visible inside the type definition, but not in the type parameter list. The `cfg16` is thus a non-atomic

config of width 16, the `cfg20` is a non-atomic config of width 20 and the `cfg30` is a non-atomic config of width 30.

10 Grouping

Grouping is a feature of the FBDL used to inform a compiler that particular functionalities might be accessed together, and their register location must meet additional constraints. This is achieved using the group functionality described in Section 7.5. The following functionalities can be grouped:

- config,
- group,
- irq,
- mask,
- param,
- return,
- static,
- satus.

However, not all groupable functionalities can belong to the same group. For example, a config shall not be grouped with a param, as param is an inner functionality of a proc, and config instantiation within a proc is invalid.

The following snippet presents three grouped configs.

```
main bus
  config_group group
    type cfg_t; width = 8;
    a cfg_t
    b cfg_t
    c cfg_t
```

Any FBDL compliant compiler must place all three configs (a, b, c) in the same register.

10.1 Single register group

A single register group is a group of functionalities that fit a single register. The overall width of all functionalities is not greater than the single register width. In such a case, all functionalities must be placed in the same register. The specification does not impose any specific order of the functionalities within the register, and it is left to the compiler implementation. The following listing presents an example bus description with three single register groups.

```
main bus
  read_write group
    c0 config; width = 16
    m0 mask;   width = 15
  mixed group
    c1 config;  width = 16
    s11 static; width = 8
    s12 status; width = 8
  read_only group
    s21 status; width = 4
    s22 status; width = 7
```

All functionalities of the read_write group can be both read and written. The code generated by a compiler for the requester must provide means for reading/writing the whole group as well as for reading/writing particular functionalities of the group.

The mixed group contains functionality that can be read and written (c1), as well as functionalities that can only be read (s11, s12). The code generated by a compiler for the requester must provide a means for reading all readable functionalities and writing all writable functionalities. It is valid even if the group has single readable or single writable functionality. The compiler must also generate means

for reading/writing particular functionalities of the group. In the case of `mixed` group this will result in two means doing exactly the same (writing the `c1` config). However, it is up to the user to decide which of the means should be used. If it makes sense, it is perfectly valid to use both of them in different contexts.

All functionalities of the `read_only` group are read-only. In this case, the compiler must generate a mean only for reading the group. It must also generate means for reading particular functionalities.

10.2 Multi register group

A multi register group is a group with functionalities that overall width is greater than the width of a single register. The specification does not impose any order of functionalities or registers in such cases, and it is left to the compiler implementation. However, the compiler must not split functionalities narrower or equal to the register width into multiple registers. This implies that any functionality with a width not greater than the register width is always read or written using single read or write access. The following snippet presents a bus description with one multi register group.

```
main bus
  multireg group
    c config;  width = 10
    m mask;    width = 10
    sc static; width = 10
    ss status; width = 10
```

The compiler must generate code for the requester allowing to write all writable functionalities of the group as well as the code allowing reading all readable functionalities of the group. It must also generate means for reading or writing particular functionalities.

There are multiple ways to place functionalities from the above example into registers. The following snippet presents one possible way.

```
        Nth register                Nth + 1 register
-----------------------------    ----------------------
|| c | m | sc | 2 bits gap ||    || ss | 22 bits gap ||
-----------------------------    ----------------------
```

However, the above arrangement might not be optimal if there is a need to read both `sc` and `ss` at the same time as it would require reading two registers not a single one. The below listing presents how to group elements within the group using subgroups.

```
main bus
  user group
    rw group
      c config; width = 10
      m mask;   width = 10
    read_only group
      sc static; width = 10
      ss status; width = 10
```

The set of possible functionalities placements within the registers is now limited as the `rw` and `read_only` subgruops are registerified before the `user` group. The below snippet shows a possible arrangement.

```
        Nth register                 Nth + 1 register
-------------------------     ---------------------------
|| c | m | 12 bits gap ||     || sc | ss | 12 bits gap ||
-------------------------     ---------------------------
```

This time reading both `sc` and `s` requires reading only one register, while reading the whole `user` group still requires reading two registers.

10.3 Array groups

An array group is a group with all functionalities being arrays. The functionalities do not necessarily have the same number of elements.

The code generated by a compiler, for an array group, for the requester must provide a means for writing an arbitrary number of elements for all writable functionalities starting from an arbitrary index. It must also provide a mean for reading an arbitrary number of elements for all readable functionalities starting from an arbitrary index.

The specification does not define what happens on access to the elements with an index greater than the length of some arrays. This is because some of the target languages support special data types indicating that the value is absent (for example, `None` - Python, `Option` - Rust), while others use for this purpose completely valid values (`0` - C, Go).

10.3.1 Single register array group

The single register array group is an array group with overall single elements width not greater than the width of a single register. The below listing presents an example bus description with a single register array group.

```
main bus
  single_reg_array group
    type cfg_t config; width = 8;
    a [1]cfg_t
    b [2]cfg_t
    c [3]cfg_t
    d [3]status; width = 8
```

In the case of a single register array group all elements with corresponding indices must be placed in the same register. Elements with consecutive indexes must be placed in consecutive registers. The below snippet presents a possible arrangement of elements for the example bus.

```
        Nth register
-------------------------------
|| d[0] | c[0] | b[0] | a[0] ||
-------------------------------
        Nth + 1 register
------------------------------------
|| d[1] | c[1] | b[1] | 8 bits gap ||
------------------------------------
        Nth + 2 register
-------------------------------
|| d[2] | c[2] | 16 bits gap ||
-------------------------------
```

10.3.2 Mutli register array group

The multi register array group ia an array group with overall single elements width greater than the width of a single register. The below listing presents an example bus description with a multi register array group.

```
main bus
  multi_reg_array group
    a [1]config; width = 16
    b [2]config; width = 12
    c [2]config; width = 12
```

In the case of multi register array group all elements with corresponding indices must be placed in consecutive registers. Also all elements with consecutive indexes must be placed in consecutive registers. Such a requirement guarantees that block access can always be used. The below snippet presents possible arrangement of elements for the example bus.

```
        Nth register                   Nth + 1 register
-------------------------------    -----------------------
|| c[0] | b[0] | 8 bits gap ||     || a[0] | 16 bits gap ||
-------------------------------    -----------------------
        Nth + 2 register                  Nth + 3 register
-------------------------------    -------------------------------
|| c[1] | b[1] | 8 bits gap ||     || c[2] | b[2] | 8 bits gap ||
-------------------------------    -------------------------------
```

10.3.3 Mixed group

A mixed group is a group with both single functionalities and array functionalities. The below listing presents an example bus description with a mixed group.

```
main bus
  mixed group
    c config; width = 10
    m mask;   width = 7
    s status; width = 8
    ca [3]config; width = 10
    sa [3]config; width = 12
```

In case of a mixed group, array functionalities shall be registerified as the first ones assuming a pure array group. Single functionalities shall be later placed in the gaps created during array registerification. If there are no gaps, or gaps are not wide enough, then all reamining single functionalities shall be registerified as single register group or multi register group. If the gaps are wide enough to place single functionalities there, but for some reason it is not desired, then subgroup can be defined to group single functionalities of the mixed group as the first ones. The below snippet presents a possible arrangement of elements for the example bus.

```
     Nth register                     Nth + 1 register
----------------------    ------------------------------------
|| ca[0] | sa[0] | c ||    || ca[1] | sa[1] | m | 3 bits gap ||
----------------------    ------------------------------------
          Nth + 2 register
------------------------------------
|| ca[2] | sa[2] | s | 2 bits gap ||
------------------------------------
```

10.4 Irq group

The irq group is used for interrupt grouping. Grouped irqs have a common interrupt consumer signal. Irqs belonging to the same group might have different values of the producer trigger (in-trigger property), but all of them must have the same value for the consumer trigger (out-trigger property). In the case of level-triggered interrupt consumer the information on the interrupt source can be read from the interrupt group flag register.

The below snippet shows an example of an irq group for level-sensitive interrupt consumer.

```
main bus
  irq group
    type irq_t irq; add-enable = true
    irq0 irq_t
    irq1 irq_t; clear = "On Read"
    irq2 irq_t; in-trigger = "Edge"
    irq3 irq_t; in-trigger = "Edge"; clear = "On Read"
```

The picture below presents a possible logical block diagram of the `irq` group with level trigger for the interrupt consumer and enable register. The `"Clear On Read"` signal is driven on every Flag Register read. The `"Explicit Clear"` signal must be driven when the requester calls a means for clearing given interrupt flags. Probably the easiest form of the `"Explicit Clear"` implementation is clear on Flag Register write, where the clear bit-mask is the value of the data bus. The Flag Register is to some extent a virtual register, as it has an address, but it does not have any storage elements. The flag is stored in the interrupt producer in case of a level-triggered producer or in the Edge Detector in case of an edge-triggered producer.
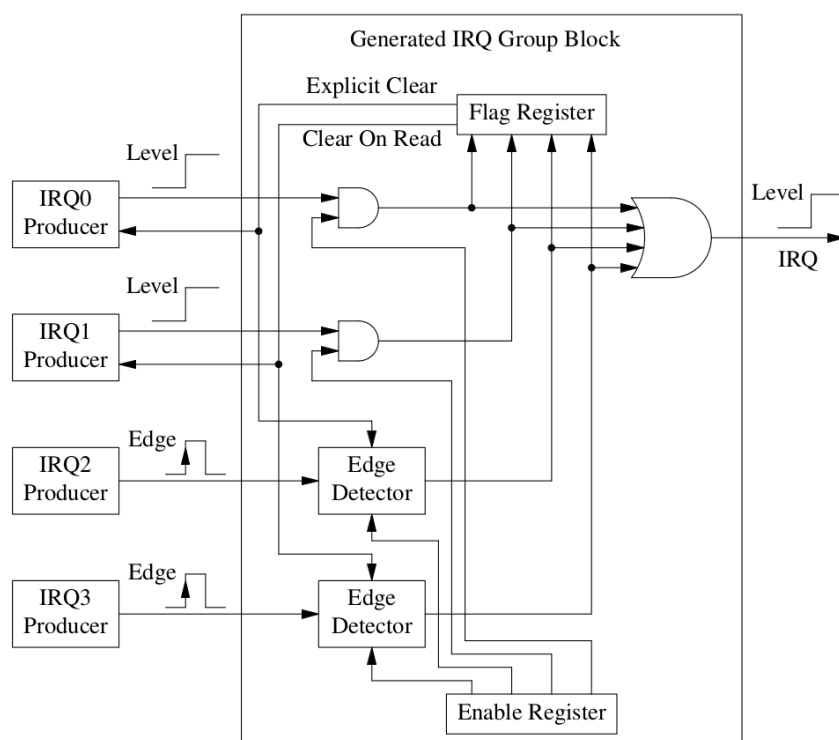
10.5 Param and return groups

Param and return groups are used to group proc or stream parameters or returns. Such a kind of grouping may be necessary for performance optimizations, as the requester may store parameters or returns in a single list or in multiple distinct lists. Param and return groups help to avoid data reshuffling before or after the access. Param and return groups are independent. The below snippet presents a valid description with a single proc with one param and one return group.

```
main bus
  p proc
    params group
      p1 param
      p2 param
    returns group
      r1 return
      r2 return
```

Param and return groups may contain subgroups. The below snippet presents examples of two invalid
and two valid params and returns grouping.

```
main bus
  # Param and return groups must have distinct names.
  invalid1 proc
    g group
      p1 param
      p2 param
    g group
      r1 return
      r2 return
  # Param and return must not be placed in the same group.
  invalid2 proc
    g group
      p param
      r return
  # Param and return groups might have subgroups.
  valid1
    params group
      p1 param
      subgroup group
        p2 param
        p3 param
  # Params and returns in different groups might have the same names.
  valid2
    params group
      x param
      y param
    returns group
      x return
      y return
```