XC Specification

IN THIS DOCUMENT

The specification given in this document describes version 1.0 of XC.

> The layout of this manual and portions of its text are based upon the K&R definition of C (see X1805). Commentary material highlighting differences between XC and C is indented and written in smaller type.

1 Lexical Conventions

A program consists of one or more *translation units* stored in files. It is translated in several phases, which are described in §13. The first phases perform low-level lexical transformations, carry out directives introduced by lines beginning with the # character, and perform macro definition and expansion. When the preprocessing of §13 is complete, the program has been reduced to a sequence of tokens.

1.1 Tokens

There are six classes of tokens: identifiers, keywords, constants, string literals, operators, and other separators. Blank spaces, horizontal tabs, newlines, formfeeds, and comments as described below, collectively referred to as *white space*, are ignored except as they separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords and constants.

Publication Date: 2011/11/22 Document Number: X5965A

1.2 Comments

Two styles of commenting are supported: the characters `/*` introduce a comment, which terminates with the characters `*/`, and the characters `//` introduce a comment, which terminates with a newline. Comments may not be nested, and they may not occur within string or character literals.

1.3 Identifiers

An identifier is a sequence of letters, digits and underscore (_) characters of any length; the first character must not be a digit. Upper and lower case letters are different.

1.4 Keywords

The following identifiers are reserved for use as keywords and may not be used otherwise:

`auto`	`else`	`return`	`union`
`break`	`enum`	`short`	`unsigned`
`case`	`extern`	`signed`	`void`
`char`	`for`	`sizeof`	`volatile`
`const`	`if`	`static`	`while`
`continue`	`int`	`struct`	
`default`	`long`	`switch`	
`do`	`register`	`typedef`	

The following identifiers are also reserved for use as keywords and may not be used otherwise:

`buffered`	`inline`	`out`	`slave`
`chan`	`isnull`	`par`	`streaming`
`chanend`	`master`	`port`	`timer`
`core`	`null`	`select`	`transaction`
`in`	`on`	`service`	`when`

The construction `port:`*n* where *n* is a sequence of digits is also a valid identifier. The sequence of digits is taken to be decimal and is interpreted as an integer constant. The following identifiers are reserved for compatibility issues and for future use:

`accept`	`claim`	`float`	`restrict`
`asm`	`double`	`module`	

1.5 Constants

There are several kinds of constants. Each has a data type; §3.2 discusses the basic types.

constant	::=	*integer-constant*
	|	*character-constant*
	|	*enumeration-constant*
	|	`null`

Floating-point constants are unsupported.

1.5.1 Integer Constants

A sequence of digits is taken to be binary if preceded by `0b` or `0B`, octal if preceded by `0`, hexadecimal if preceded by `0x` or `0X`, and decimal otherwise. integer constant may be suffixed by the letter `u` or `U` (unsigned), the letter `l` or `L` (long), or both (unsigned long).

The type of an integer constant depends on its form, value and suffix. (See §3 for a discussion of types.) An unsuffixed decimal constant has the first of the following types in which its value can be represented: `int`, `long int`, `unsigned long int`; an unsuffixed octal or hexadecimal constant has the first possible of types: `int`, `unsigned int`, `long int`, `unsigned long int`. An unsigned constant has the first possible of types: `unsigned int`, `unsigned long int`; a long constant has the first possible of types: `long int`, `unsigned long int`.

1.5.2 Character Constants

A character constant is a sequence of one or more characters (excluding the single-quote and newline characters) enclosed in single quotes. The value of a character constant with a single character is the numeric value of the character in the machine's character set at execution time. The value of a multi-character constant is implementation-defined.

Wide character constants are unsupported.

The following escape sequences are supported.

newline	NL	`\n`	backslash	`\`	`\\`
horizontal tab	HT	`\t`	question mark	`?`	`\?`
vertical tab	VT	`\v`	single quote	`'`	`\'`
backspace	BS	`\b`	double quote	`"`	`\"`
carriage return	CR	`\r`	octal number	*ooo*	`\`*ooo*
formfeed	FF	`\f`	hex number	*hh*	`\x`*hh*
audible alert	BEL	`\a`			

The escape sequence `\`*ooo* requires one, two or three octal digits. The sequence `\x`*hh* requires one or more hexadecimal digits; its behaviour is undefined if the resulting character value exceeds that of the largest character. For either octal or hexadecimal escape characters, if the implementation treats the char type as signed, the value is sign-extended as if cast to char type. If any other character follows the \ then the behaviour is undefined.

1.5.3 Enumeration Constants

Identifiers declared as enumerators (see §7.5) are constants of type `int`.

1.5.4 Null Constants

The null constant has type `null`.

1.6 String Literals

A string literal is a sequence of zero or more characters (excluding the double-quote and newline characters) enclosed in double quotes. It has type "array of characters" and storage class static (see §3.1) initialised with the given characters. Whether identical string literals are distinct is implementation-defined, and the behaviour of a program that attempts to alter a string literal is undefined.

Adjacent string literals are concatenated into a single string. After any concatenation, a null byte `\0` is appended to the string. All of the character escape sequences are supported.

2 Syntax Notation

In the syntax notation used in this manual, syntactic categories are indicated by sans serif type, and literal words and characters by `typewriter` style. An optional terminal or nonterminal symbol carries the subscripted suffix "*opt*," so that, for example,

`{` $\textit{expression}_{\text{opt}}$ `}`

means an optional expression, enclosed in braces. The terms "zero or more" and "one or more" are represented using angled brackets along with the star (*) and plus (+) symbols respectively, so that, for example,

$\langle \textit{declaration} \rangle^*$

means a sequence of zero or more declarations, and

$\langle \textit{declaration} \rangle^+$

means a sequence of one or more declarations.

3 Meaning of Identifiers

Identifiers (or names) refer collectively to functions, tags of structures and unions, members of structures or unions, and objects. An object (or variable) is a location in storage, and its interpretation depends on its *storage class* and its *type*. The storage class determines the lifetime of the storage associated with the identifier; the type determines the meaning of the values found in the identified object. A name also has scope, which is the region of the program in which it is known, and a linkage, which determines whether the same name in another scope refers to the same object or function. Scope and linkage are discussed in §10.

3.1 Storage Class

An object has either *automatic* or *static* storage. Automatic objects are local to a block (§8.4) and are discarded on exit from the block. Declarations within a block create automatic objects if no storage class is mentioned, or if the `auto` or `register` specifier is used.

Static objects may be local to a block or external to all blocks, but in either case retain their values across exit from and reentry to functions and blocks. Within a block, static objects are declared with the keyword `static`. The objects declared outside all blocks, at the same level as function definitions, are always static. They may be made local to a particular translation unit by use of the `static` keyword; this gives them *file-scope* (or *internal linkage*). They become global to an entire program by omitting an explicit storage class, or by using the keyword `extern`; this gives them *program-scope* (or *external linkage*).

A function may be declared with the keyword `service`. This specifier has no effect on the behaviour of the function; the extent to which suggestions made by using this specifier are effective is implementation-defined.

3.2 Basic Types

Objects declared as char are large enough to store any member of the execution character set. If a genuine character from that set is stored in a `char` object, its value is equivalent to the integer code for the character, and is non-negative. Other quantities may be stored into char variables, but the available range of values, and especially whether the value is signed, is implementation-defined.

Objects declared `unsigned char` consume the same amount of space as plain characters, but always appear non-negative; explicitly signed characters declared `signed char` likewise take the same space as plain characters.

In addition to the char type, up to three sizes of integer are available, declared `short int`, `int` and `long int`. Plain `int` objects have the natural size suggested by the host machine architecture. Longer integers provide at least as much storage as shorter ones, but the implementation may make plain integers equivalent to either short or long integers. The `int` types all represent signed values unless specified otherwise.

Unsigned integers obey the laws of arithmetic modulo 2^n where n is the number of bits in the representation. The set of non-negative values that can be stored in a signed object is a subset of the values that can be stored in the corresponding unsigned object, and the representation for the overlapping values is the same.

All of the above types are collectively referred to as *arithmetic* types, because they can be interpreted as numbers, and as *integral* types, because they represent integer values.

The `void` type specifies an empty set of values; it is used as the type returned by functions that generate no value.

The C types `long long int`, `float` and `double` are unsupported.

The chan type specifies a logical communication channel over which values can be communicated between parallel statements (§ 8.8). The chanend type specifies one end of a communication channel.

The locations of at most two implied ends of chan (themselves chanends) are defined through the use of the channel in at most two parallel statements (§8.8).

Channel ends are used as operands of input and output statements (§ 8.3). Channels are bidirectional and synchronised: an outputter waits for a matching inputter to become ready before data is communicated. Whether a streaming channel is synchronised or unsynchronised is implementation-defined.

The port type specifies a p-bit register, which interfaces to a collection of p pins used for communicating with the environment where p is implementation-defined. The port:n type specifies an n-bit register, which interfaces to a collection of p pins used for communicating with the environment (where p need not equal n). A void port is a special type of port that may not be used for input or output. A port also has a notional *transfer type* and *counter* type (see §8.3); these types are implementation-defined.

Ports are used as operands of input and output statements (§8.3).

The timer type is a special type of input port that returns the current time when input from. A void timer is a timer that may not be used for input. A timer also has a notional *counter* type (see §8.3); this type is implementation-defined.

The core type specifies a processor core on which ports and parallel statements may be placed. Objects of core type do not reserve storage.

Channel ends, ports, timers and cores are collectively referred to as having *resource* types. Except for cores, which do not reserve storage, an object of resource type refers to a location in storage in which an identifier for the resource is recorded.

> chan, chanend, port, timer, core, buffered and streaming are new.

3.3 Derived Types

In addition to the basic types, the following derived types may be constructed in the following ways:

- Arrays of objects of a given type.
- Functions returning objects of a given type.
- References to objects of a given type.
- Structures containing a sequence of objects of various types.
- Unions capable of containing any one of several objects of various types.
- Lists of objects containing a sequence of objects of various types.

In general these methods of constructing objects can be applied recursively.

> Lists of types are used in multiple assignment statements (§8.2); pointers are replaced by references (see §6.1; §6.3.2; §7.7.2).

3.4 Type Qualifiers

An object's type may be qualified `const`, which announces that its value will not be changed; its range of values and arithmetic properties is unaffected.

A port may be qualified `in` or `out`, which announces that it will only be used for input or output operators (§8.3).

Qualifiers are discussed in §6.3.2 and §7.2.

`in` and `out` are new.

4 Objects and Lvalues

An *object* is a named region of storage; an *lvalue* is a reference to an object. For example, if `str` is an identifier of type "1-dimensional array of char" then `str[0]` is an lvalue referring to the character object indexed by the first element of the array `str`.

A *modifiable lvalue* is an lvalue which is modifiable: it must not be an array, and must not have a resource or incomplete type, or be a function. Also, its type must not be qualified with `const`; if it is a structure or union, it must not have any member or, recursively, submember qualified with `const`.

5 Conversions

Some operators, depending on their operands, cause conversion of the value of an operand from one type to another. This section explains the results to be expected from such conversions. It details the conversions demanded by most operators.

5.1 Integral Promotion

A character or a short integer, both either signed or not, may be used in an expression wherever an integer may be used. If an `int` can represent all the values of the original type then the value is converted to `int`, otherwise the value is converted to `unsigned int`.

5.2 Integral Conversions

Any integer is converted to a given unsigned type by finding the smallest non-negative value that is congruent to that integer, modulo one more than the largest value that can be represented in the unsigned type.

When any integer is converted to a signed type, its value is unchanged if it can be represented in the new type, and implementation-defined otherwise.

5.3 Arithmetic Conversions

Many operators cause conversions which bring their operands into a common type, which is also the type of the result. The rules for performing these *usual arithmetic conversions* are as follows:

- First, integral promotions are performed on both operands.

- If either operand is `unsigned long int` then the other is converted to `unsigned long int`.
- Otherwise, if one operand is `long int` and the other is `unsigned int` then: if a `long int` can represent all values of an `unsigned int` then the `unsigned int` operand is converted to `long int`, otherwise both operands are converted to `unsigned long int`.
- Otherwise, if one operand is `long int` then the other is converted to `long int`.
- Otherwise, if either operand is `unsigned int` then the other is converted to `unsigned int`.
- Otherwise both operands have type `int`.

5.4 Void

The (nonexistent) value of a `void` object may not be used in any way, and neither explicit nor implicit conversion to any non-void type may be applied. An object of type `void port` or `void timer` may not be used for input or output.

6 Expressions

The precedence of expression operators is the same as the order of the major subsections of this section, highest precedence first. Within each section, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein.

The precedence and associativity of operators is fully specified. The order of evaluation of expressions does not, with certain exceptions, affect the behaviour of the program, even if the subexpressions involve side effects. In particular, a variable which is changed in one part of an expression may not unless otherwise stated appear in any other part of the expression. This rule applies recursively to all variables which are changed in functions called in the expression.

The handling of overflow, divide check, and other exceptions in expression evaluation is implementation-defined.

6.1 Reference Generation

If the type of an expression is "array of *T*," for some type *T*, then the value of the expression is a reference to the array, and the type of the expression is altered to "reference to array of *T*."

6.2 Primary Expressions

Primary expressions are variable references, function calls, constants, strings, or expressions in parentheses:

primary-expression	::=	*variable-reference*
	|	*function-call*
	|	*constant*
	|	*string*
	|	(*expression*)

A variable reference is a primary expression, providing the identifier named (§6.3) has been suitably declared as discussed below; the type of the identifier is specified by its declaration; the type of the expression is that of the identifier.

A function call is a primary expression; the type of the expression depends on the return type of the function (§6.3.2).

A constant is a primary expression; the type of the expression is that of the constant (which depends on its form discussed in §1.5).

A string literal is a primary expression; the type of the expression is "array of `char`."

A parenthesised expression is a primary expression whose type and value are identical to those of the unadorned expression.

6.3 Postfix Expressions

The operators in postfix expressions group left to right.

postfix-expression	::=	*primary-expression*
	|	*variable-reference* ++
	|	*variable-reference* --
variable-reference	::=	*identifier*
	|	*variable-reference* [*expression*]
	|	*variable-reference* . *identifier*
	|	(*variable-reference* , *type-name*)
function-call	::=	*identifier* (*expression-list*$_{\text{opt}}$)
expression-list	::=	*expression*
	|	*expression* , *expression-list*

6.3.1 Array References

A variable reference followed by an expression in square brackets is a subscripted array reference. The variable reference must either have type "*n*-array of *T*" or "reference to an *n*-array of *T*," where *n* is the number of dimensions and *T* is some type, and the expression must have integral type; the type of the subscripted variable reference is *T*. If the value of the expression is less than zero or greater than or equal to *n* then the expression is invalid. See §7.7.1 for further discussion.

6.3.2 Function Calls

A function call is an identifier followed by parentheses containing an optional list of comma-separated expressions, which constitute the arguments to the function. If the identifier has type "transaction function returning `void`" then the call must be within the scope of a transaction statement (§ 8.9). Otherwise, the identifier must have type "function returning *T*," or "select function returning *T*," for some type *T*, in which case the value of the function call has type *T*.

> Function declarations are limited to file-scope only (§9). Implicit function declarations (see K&R §A7.3.2) are unsupported.

The term *argument* refers to an expression passed by a function call, and the term parameter refers to an input object (or its identifier) received by a function definition, or described in a function declaration.

If the type of a parameter is "reference to *T*," for some *T*, then its argument is passed by reference, otherwise the argument is passed by value. In preparing for the call to a function, a copy is made of each argument that is passed by value. A function may change the values of these parameter objects, which are copies of the argument expressions, but these changes cannot affect the values of the arguments. For objects that are passed by reference, a function may change the values that these objects dereference, thereby affecting the values of the arguments. For the purpose of disjointness checking, passing an object by reference counts as a write to that object unless the type of the parameter is qualified as const or an array of objects qualified as `const`.

For arguments passed by value, the argument and parameter are deemed to agree in type if the promoted type of the argument is that of the parameter itself, without promotion. For arguments passed by reference, the argument and parameter agree in type only if the types are equivalent (see §7.11) ignoring all qualifiers and array sizes, and obey the following rules:

- An object or an array of objects declared with the qualifier `const` may only be used as an argument to a function with parameter qualified `const`.
- An object declared with the qualifier `in` may only be used as an argument to a function with parameter qualified `in` or `void`.
- An object declared with the qualifier `out` may only be used as an argument to a function with parameter qualified `out` or `void`.
- An object declared with the specifier `void` may only be used as an argument to a function with parameter specified `void`. An object not qualified `in`, `out` or `void` may be used as an argument to a function with parameter qualified either `in`, `out` or `void`.
- An object declared with the qualifier `buffered` may only be used as an argument to a function parameter qualified `buffered`. An object not declared with the qualifier `buffered` may only be used as an argument to a function parameter not qualified `buffered`.
- An object declared with the qualifier `streaming` may only be used as an argument to a function parameter qualified `streaming`. An object not declared with the qualifier `streaming` may only be used as an argument to a function parameter not qualified `streaming`.
- An object declared with an array size of n may only be used as an argument to a function parameter that is an array of unknown size or of size m where $m <= n$.
- An object passed to a parameter declared without the qualifier `const` must be an lvalue.

A variable which is changed in one argument may not appear in any other argument. This rule applies recursively to all variables appearing in functions called by the arguments.

The arguments passed by value are converted, as if by assignment, to the types of the corresponding parameters of the function's declaration (or prototype). The number of arguments must be the same as the number of parameters, unless the declaration's parameter list ends with the ellipsis notation (`, ...`). In that case, the number of arguments must equal or exceed the number of parameters; trailing integral arguments beyond the explicitly typed parameters undergo integral promotion (§5.1).

The order of evaluation of arguments is unspecified, but the arguments are completely evaluated, including all side effects, before the function is entered. Recursive calls to any function are permitted.

The creation of more than one reference to the same object of basic type, a structure, a union or an array is invalid. The creation of a reference to a structure, union or array, and to a member or element recursively contained within is invalid. The creation of more than one reference to objects contained within distinct members of a union is invalid.

6.3.3 Structure References

A variable reference followed by a dot followed by an identifier is a member reference. The variable reference must be a structure or union, and the identifier must name a member of the structure or union. The value is the named member of the structure or union, and its type is the type of the member.

Structures and unions are discussed in §7.4.

6.3.4 Reinterpretation

A left parenthesis followed by a variable reference followed by a comma followed by a type name (§7.9) followed by a right parenthesis is a reinterpretation cast.

The variable reference must not specify a resource type; its type must be complete or it must be an incomplete array with the first dimension missing which, if provided, completes the type. The variable type name must not be a resource type; it must be complete.

If the size of the type of the variable reference is unknown because it references an array parameter with unknown size then the following two rules apply. First, if the size of the type name is a compile-time constant T then at run-time if the size of the variable reference is less than T then the reinterpret operation is invalid. Second, if the size of the type name is not known at compile-time because it is an array in which the largest dimension is unspecified then at run-time the reinterpret operation provides a value for the dimension d such that the size of the resulting type is not larger than the size of the type of the variable reference, but with a value of $d + 1$ it would be.

If the size of the type of the variable reference is a compile-time constant V then the following two rules apply. First, if the size of the type name is a compile-time constant T then T must not be greater than V. Second, if the size of the type name is unknown because it references an array in which the largest dimension is unspecified then a value for this dimension d is completed at compile-time such that the size of the resulting type is not larger than V, but with a value of $d + 1$ it would be.

No arithmetic conversions are performed: the effect of the reinterpretation is to treat the variable as if it had the specified type. An array of size zero is a valid reinterpretation; any attempted index into the array is invalid.

The use of a reinterpreted object may be invalid if it is not suitably aligned in storage. It is guaranteed only that an object may be reinterpreted to an object whose type requires less or equally strict storage alignment; the notion of "alignment" is implementation-defined, but objects of the `char` types have the least strict alignment requirements.

6.4 Unary Operators

Expressions with unary operators group right-to-left.

unary-expression	::=	*postfix-expression*
	|	`++` *variable-reference*
	|	`--` *variable-reference*
	|	*unary-operator cast-expression*
	|	`sizeof` *unary-expression*
	|	`sizeof` (*type-name*)
	|	`isnull` (*unary-expression*)
unary-operator	::=	*one of*
		`+ - ~ !`

6.4.1 Prefix Incrementation Operators

A unary expression preceded by a `++` or `--` operator is a unary expression. The operand is incremented (`++`) or decremented (`--`) by 1. The value of the expression is the value after the incrementation (decrementation). The operand must be a modifiable lvalue; see the discussion of additive operators (§6.7) and assignment (§6.17) for details of the operation. The result is not an lvalue.

6.4.2 Unary Plus Operator

The operand of the unary `+` operator must have arithmetic type, and the result is the value of its operand. An integral operand undergoes integral promotion; the type of the result is the type of the promoted operand.

6.4.3 Unary Minus Operator

The operand of the unary `-` operator must have arithmetic type, and the result is the negative of its operand. An integral operand undergoes integral promotion. The negative of an unsigned quantity is computed by subtracting the promoted value from the largest value of the promoted type and adding one; but negative zero is zero. The type of the result is the type of the promoted operand.

6.4.4 One's (Bitwise) Complement Operator

The operand of the unary ~ operator must have integral type, and the result is the one's complement of its operand. The integral promotions are performed. If the operand is unsigned, the result is computed by subtracting the value from the largest value of the promoted type. If the operand is signed, the result is computed by converting the promoted operand to the corresponding unsigned type, applying ~, and converting back to the signed type. The type of the result is the type of the promoted operand.

6.4.5 Logical Negation Operator

The operand of the `!` operator must have arithmetic type, and the result is 1 if the value of its operand compares equal to 0, and 0 otherwise. The type of the result is `int`.

6.4.6 Sizeof Operator

The `sizeof` operator yields the number of bytes required to store an object of the type of its operand. The operand is either an expression, which is not evaluated, or a parenthesised type name. When `sizeof` is applied to a `char`, the result is 1; when applied to an array, the result is the total number of bytes in the array. When applied to a structure or union, the result is the number of bytes in the object, including any padding required to make the object tile an array: the size of an array of *n* elements is *n* times the size of one element. When applied to a reference, the result is the number of bytes in the object referred to. The operator may not be applied to an operand of function type, of resource type or of an incomplete type. The operator may not be applied to an operand of reference type where the reference is to an array of unknown size. The value of the result is implementation-defined. The result is an unsigned integral constant; the particular type is implementation-defined.

6.4.7 Isnull Operator

The operand of the `isnull` operator must be an lvalue. The result is 1 if its operand has value null, and 0 otherwise. The type of the result is `int`.

6.5 Casts

A unary expression preceded by the parenthesised name of a type causes conversion of the value of the expression to the named type.

cast-expression ::= *unary-expression*
| (*type-name*) *cast-expression*

This construction is called a *cast*. The cast must not specify a structure, a union, an array, or a resource type; neither must the expression. Type names are described in §7.9. The effects of arithmetic conversions are described in §5.3. An expression with a cast is not an lvalue.

6.6 Multiplicative Operators

The multiplicative operators *, / and % group left-to-right.

multiplicative-expression	::=	*cast-expression*
	|	*multiplicative-expression* * *cast-expression*
	|	*multiplicative-expression* / *cast-expression*
	|	*multiplicative-expression* % *cast-expression*

The operands of * and / must have arithmetic type; the operands of % must have integral type. The usual arithmetic conversions are performed on the operands, and determine the type of the result.

The binary * operator denotes multiplication.

The binary / operator produces the quotient, and the % operator the remainder, of the division of the first operand by the second; if the second operand is 0 then the result is implementation-defined. Otherwise, it is always true that `(a/b)*b + a%b` is equal to `a`. If both operands are non-negative, then the remainder is non-negative and smaller than the divisor; if not, it is guaranteed only that the absolute value of the remainder is smaller than the absolute value of the divisor.

6.7 Additive Operators

The additive operators + and - group left-to-right.

additive-expression	::=	*multiplicative-expression*
	|	*additive-expression* + *multiplicative-expression*
	|	*additive-expression* - *multiplicative-expression*

For both operators, each operand must have arithmetic type. The usual arithmetic conversions are performed on the operands, and determine the type of the result. The result of the + operator is the sum of the operands, and the result of the - operator is the difference of the operands.

6.8 Shift Operators

The shift operators << and >> group left-to-right. For both operators, each operand must be integral, and is subject to integral promotions. The type of the result is that of the promoted left operand. The result is undefined if the right operand is negative, or greater than or equal to the number of bits in the left expression's type.

shift-expression	::=	*additive-expression*
	|	*shift-expression* << *additive-expression*
	|	*shift-expression* >> *additive-expression*

The result of the << operator is the left operand left-shifted by the number of bits specified by the right operand. The value of the >> operator is the left operand right-shifted by the number of bits specified by the right operand.

6.9 Relational Operators

The relational operators < (less), > (greater), <= (less or equal) and >= (greater or equal) group left-to-right (but this fact is not useful).

relational-expression ::= *shift-expression*
| *relational-expression* < *shift-expression*
| *relational-expression* > *shift-expression*
| *relational-expression* <= *shift-expression*
| *relational-expression* >= *shift-expression*

For all of these operators, each operand must have arithmetic type. The usual arithmetic conversions are performed; the type of the result is `int`.

All of these operators produce a result of 0 if the specified relation is false and 1 if it is true.

6.10 Equality Operators

equality-expression ::= *relational-expression*
| *equality-expression* == *relational-expression*
| *equality-expression* != *relational-expression*

The equality operators == (equal to) and != (not equal to) are analogous to the relational operators except for their lower precedence.

6.11 Bitwise AND Operator

AND-expression ::= *equality-expression*
| *AND-expression* & *equality-expression*

The operands of the bitwise AND operator & must have integral type. The usual arithmetic conversions are performed; the result is the bitwise AND function of the operands.

6.12 Bitwise Exclusive OR Operator

exclusive-OR-expression ::= *AND-expression*
| *exclusive-OR-expression* ^ *AND-expression*

The operands of the bitwise exclusive OR operator ^ must have integral type. The usual arithmetic conversions are performed; the result is the bitwise exclusive OR function of its operands.

6.13 Bitwise Inclusive OR Operator

inclusive-OR-expression ::= *exclusive-OR-expression*
| *inclusive-OR-expression* | *exclusive-OR-expression*

The operands of the bitwise inclusive OR operator | must have integral type. The usual arithmetic conversions are performed; the result is the bitwise inclusive OR function of its operands.

6.14 Logical AND Operator

logical-AND-expression ::= *inclusive-OR-expression*
| *logical-AND-expression* `&&` *inclusive-OR-expression*

The logical AND operator `&&` groups left-to-right. It returns 1 if both its operands compare unequal to zero, 0 otherwise. It guarantees left-to-right *short-circuit* evaluation: the right operand is evaluated only if the left operand evaluates to 1. The operands must have arithmetic type, but need not be the same type; the type of the result is `int`. A variable which is changed by one operand may appear in the other operand.

6.15 Logical OR Operator

logical-OR-expression ::= *logical-AND-expression*
| *logical-OR-expression* `||` *logical-AND-expression*

The logical OR operator `||` groups left-to-right. It returns 1 if either of its operands compares unequal to zero, 0 otherwise. It guarantees left-to-right *short-circuit*: the right operand is evaluated only if the left operand evaluates to 0. The operands must have arithmetic type, but need not be the same type; the type of the result is `int`. A variable which is changed by one operand may appear in the other operand.

6.16 Conditional Operator

conditional-expression ::= *logical-OR-expression*
| *logical-OR-expression* `?` *expression* `:` *conditional-expression*

If the neither the second and third operands have `null` type they must have equivalent types (see §7.11) ignoring all qualifiers except for `buffered` and `streaming`, and any array sizes, or they must both have arithmetic type.

The first expression is evaluated including all side effects; if it compares unequal to 0, the result is the value of the second expression, otherwise the result is the value of the third expression. If either the second or third operand has type `null`, the result has the type of the other operand. Otherwise, if the second and third operands have equivalent types ignoring qualifiers and any array sizes, the result type has the common type with qualifiers determined by the following rules:

- If either operand is qualified `const`, the result is qualified `const`.
- If either operand is specified with `void`, the result is specified with `void`.
- If one operand is qualified in and the other operand is qualified `out`, the result is specified with `void`. Otherwise, if either operand is qualified `in` or `out`, the result is also qualified `in` or `out` respectively. If the operands are arrays of different sizes, the result has type "array of unknown size."

If the second and third operands have arithmetic type but are not equivalent, the usual arithmetic conversions are performed, and determine the type of the result.

The expression is a lvalue if no arithmetic conversions are performed and the second and third operands both have type `null` or are lvalues.

6.17 Assignment Expressions

There are several assignment operators; all group right-to-left.

assignment-expression	::=	*conditional-expression*
	|	*variable-reference assignment-operator assignment-expression*
assignment-operator	::=	*one of*
		`= *= /= %= += -= <<= >>= &= ^= |=`

All require a modifiable lvalue as the left operand. The identifier named by the *variable-reference* may appear in any other parts of the assignment, including recursively any functions called, as long as the variables named by the identifiers in these parts are not changed. The type of an assignment expression is that of its left operand, and the value is the value stored in the left operand after the assignment has taken place.

In the simple assignment with =, the value of the expression replaces that of the object referred to by the lvalue. One of the following must be true: both operands have arithmetic type, in which case the right operand is converted to the type of the left by the assignment; or both operands are structures or unions of the same type.

An expression of the form `V` *op*`= E` is equivalent to `V = V` *op* `(E)` except that `V` is evaluated only once.

6.18 Comma Operator

A restricted form of the comma operator is supported in `for` loops (see §8.6).

6.19 Constant Expressions

Syntactically, a constant expression is an expression restricted to a subset of operators.

constant-expression	::=	*conditional-expression*

Expressions that evaluate to a constant are required in several contexts: after case in labelled statements, as array bounds, and in certain preprocessor expressions.

Constant expressions may not contain assignments, increment or decrement operators or function calls, except in an operand of `sizeof`. If the constant expression is required to be integral, its operands must consist of integer, enumeration and character constants; casts must specify an integral type.

7 Declarations

Declarations specify the interpretation given to each identifier. Declarations that reserve storage are called *definitions*. The syntax of declarations is:

declaration	::=	*on-statement*$_{\text{opt}}$ *actual-declaration*
actual-declaration	::=	*var-declaration*
	|	*fnc-declaration* ;
	|	*trn-declaration* ;
	|	*sel-declaration* ;
on-statement	::=	`on` *variable-reference* :
var-declaration	::=	⟨*dec-specifierier*⟩* *init-var-declarator-list*$_{\text{opt}}$;
fnc-declaration	::=	⟨*dec-specifierier*⟩* *fnc-declarator*
	|	{ *dec-specifier-list* } *fnc-declarator*
trn-declaration	::=	⟨*dec-specifierier* ⟩* `transaction` *fnc-declarator*
sel-declaration	::=	⟨*dec-specifierier*⟩* `select` *fnc-declarator*
dec-specifier-list	::=	⟨*dec-specifier*⟩*
	|	*dec-specifier-list* , ⟨*dec-specifierier*⟩*

A variable declaration prefixed with `on` must declare an object of type `port` or `port:`n. The variable following `on` must refer to an object of type `core`.

`on` does not change the meaning of the declaration it prefixes.

The *var-declarator*s in the *init-var-declarator-list* and the *fnc-declarator* (see §7.6) contain the identifiers being declared.

dec-specifier	::=	*storage-class-specifier*
	|	*type-specifier*
	|	*type-qualifier*
	|	`inline`
init-var-declarator-list	::=	*init-var-declarator*
	|	*init-var-declarator* , *init-var-declarator-list*
init-var-declarator	::=	*var-declarator* ⟨= *initialiser* ⟩$_{\text{opt}}$

Declarators are discussed later (§7.6); they contain the names being declared. A declaration must have at least one declarator, or its type specifier must declare a structure tag or a union tag; empty declarations are not permitted.

7.1 Storage Class Specifiers

The storage class specifiers are:

```
storage-class-specifier ::= auto
                          | register
                          | static
                          | extern
                          | typedef
                          | service
```

The meanings of the storage classes were discussed in §3.

The `auto` and `register` specifiers give the declared objects automatic storage class, and may be used only within functions. Such declarations also serve as definitions and cause storage to be reserved.

The `static` specifier gives the declared objects static storage class, and may be used either inside or outside functions. Inside a function, this specifier causes storage to be allocated, and serves as a definition; for its effect outside a function, see §10.2.

The `extern` specifier, used inside a function, specifies that the storage for the declared objects is defined elsewhere; for its its effects outside a function see §10.2.

The `typedef` specifier does not reserve storage and is called a storage class specifier for syntactic convenience; it is discussed in §7.10.

At most one of each of the storage class specifiers may be given in a declaration. If none is given, these rules are used: objects declared inside a function are taken to be `auto`; objects and functions declared outside a function, at file-scope, are taken to be static, with external linkage. The specifier `service` may only be given with external function declarations.

7.2 Type Specifiers

The type specifiers are:

type-specifier ::= `void`
| `char`
| `short`
| `int`
| `long`
| `signed`
| `unsigned`
| `chan`
| `chanend`
| `port`
| `port:`*n*
| `timer`
| `core`
| *struct-or-union-specifier*
| *enum-specifier*
| *typedef-name*

At most one of `long` or `short` may be specified together with `int`; the meaning is the same if `int` is not specified. At most one of signed or unsigned may be specified together with `int`, `short`, `long` or `char`; either may appear alone, in which case `int` is understood. The `signed` specifier is useful for forcing `char` objects to carry a sign; it is permissible but redundant with other integral types. `void` may be specified together with `port` or `port:`*n* to declare a `void port`; it may be specified together with `timer` to specify a `void timer`.

Otherwise, at most one type specifier may be given in a declaration; if omitted then it is taken to be `int`.

Types may also be qualified, to indicate special properties of the objects being declared.

type-qualifier ::= `const`
| `volatile`
| `in`
| `out`
| `buffered`
| `streaming`

`const` may appear with any type specifier. A `const` object may be initialised, but not thereafter assigned or input to. No object may be qualified `volatile`. A compiler is required to recognise this qualifier and issue an appropriate error message.

`in` and `out` may appear with the `port` and `port:`*n* type specifiers but not with `void`. An object qualified `in` may appear in input operations only, and an object qualified `out` may appear in output operations only (§8.3). `buffered` may appear with the `port` and `port:`*n* type specifiers. `streaming` may appear with the `chan` and `chanend` type specifiers.

Automatic variables may not be declared with type `port`, `port:`n, `chanend` or `core`. Static variables may not be declared with types `chan` or `chanend`. Ports specified with `void` may not be used in input or output operations.

7.3 `inline` specifier

Types may be specified `inline`, to suggest that calls to the function be as fast as possible. A definition is an *inline definition* if all the file-scope declarations for a function in the translation unit include the `inline` specifier without `extern`. An inline definition of a function with external linkage must not contain a definition of a modifiable object with static storage, and must not contain a reference to an identifier with external linkage.

The extent to which suggestions are effective is implementation-defined.

7.4 Structure and Union Declarations

A structure is an object consisting of a sequence of named members of various types. A union is an object that contains, at different times, any one of several members of various types. Structures and unions have the same form.

struct-or-union-specifier	::=	*struct-or-union identifier*$_{\text{opt}}$ `{` ⟨*member*⟩$^+$ `}`
	|	*struct-or-union identifier*
struct-or-union	::=	`struct`
	|	`union`

A *member* is a declaration for a member of the structure or union.

member	::=	⟨*specifier-or-qualifier*⟩$^+$ *struct-var-declarator-list* `;`
specifier-or-qualifier	::=	*type-specifier*
	|	*type-qualifier*
struct-var-declarator-list	::=	*var-declarator*
	|	*var-declarator* `,` *struct-var-declarator-list*

A type specifier of the form

struct-or-union identifier `{` ⟨*member*⟩$^+$ `}`

declares the identifier to be the *tag* of the structure or union specified by the member list. A subsequent declaration may refer to the same type by using the tag in a specifier without the member list:

struct-or-union identifier

If a specifier with a tag but without a list appears when the tag is not declared, an *incomplete type* is specified. Objects with an incomplete structure or union type may be used in contexts where their size is not needed. The type becomes complete on occurrence of a subsequent specifier with that tag, and containing a declaration list. Even in specifiers with a list, the structure or union type being declared is incomplete within the list, and becomes complete only at the } terminating the specifier.

A structure or union may not contain a member of incomplete or resource type, except that a structure may contain a member of type port or timer. If a structure is declared to have a member with one of these types then variables of the structure may be declared only as external declarations (see §9).

A structure or union specifier with a list but no tag creates a unique type; it can be referred to directly only in the declaration of which it is a part.

The names of members and tags do not conflict with each other or with ordinary variables. A member name may not appear twice in the same structure or union, but the same member name may be used in different structures or unions.

The members of a structure have addresses increasing in the order of their declarations. A member of a structure is aligned at an addressing boundary depending on its type.

A union may be thought of as a structure all of whose members begin at offset 0 and whose size is sufficient to contain any of its members. At most one of the members can be stored in a union at any time.

If a union contains several structures that share a common initial sequence, and if the union currently contains one of these structures, it is permitted to refer to the common initial part of any of the contained structures.

Bit-fields are unsupported.

7.5 Enumerations

Enumerations are unique types with values ranging over a set of named constants called enumerators. The form of an enumeration specifier borrows from that of structures and unions.

enum-specifier	::=	`enum` *identifier*$_{\text{opt}}$ { *enumerator-list* }
	|	`enum` *identifier*$_{\text{opt}}$ { *enumerator-list* , }
	|	`enum` *identifier*
enumerator-list	::=	*enumerator*
	|	*enumerator-list* , *enumerator*
enumerator	::=	*identifier*
	|	*identifier* = *constant-expression*

The enumerator type is compatible with `int`; identifiers in an enumerator list are declared as constants of type `int`, and may appear wherever constants are required. If no enumerators with = appear, then the values of the corresponding constants begin at 0 and increase by 1 as the declaration is read from left to right. An enumerator with = gives the associated identifier the value specified; subsequent identifiers continue the progression from the assigned value.

Enumerator names in the same scope must all be distinct from each other and from ordinary variable names, but the values need not be distinct.

The identifier in the enum-specifier names a particular enumeration. The rules for enum specifiers with and without tags and lists are the same as those for structure or union specifiers, except that incomplete enumeration types do not exist; the tag of an enum-specifier without an enumerator-list must refer to an in-scope specifier with a list.

7.6 Declarators

Declarators have the syntax:

var-declarator	::=	*identifier* ⟨*dimension-size* ⟩*
	|	`&` *identifier*
	|	`?` *identifier* ⟨*dimension-size*⟩*
	|	`&` `?` *identifier*
fnc-declarator	::=	*identifier* `(` *parameter-type-list*$_{\text{opt}}$ `)`
dimension-size	::=	`[` *constant-expression*$_{\text{opt}}$ `]`

The structure of declarators resembles that of function and array expressions; the grouping is the same.

7.7 Meaning of Declarators

One or more declarators appear after a sequence of storage class and type specifiers. The declarators may be prefixed by either `select` or `transaction`, in which case only storage class specifiers are permitted as the declaration specifiers; the return type is implicitly `void`. Each declarator declares a unique main identifier. The storage class specifiers apply directly to this identifier, but its type depends on its form.

Considering only the type parts of the declaration specifiers (§7.2), the optional `transaction` and `select`, and a particular declarator, a declaration has the form "*opt-transaction-or-select* `T D`" where `T` is a type and `D` is a declarator. The type attributed to the identifier in the various forms of declarator is described using this notation.

In a declaration `T D` where `D` is an unadorned identifier, the type of the identifier is `T`.

A port may be declared as an external declaration (see §9) or as a parameter only. A channel may be declared as a local variable only and a channel end may be declared as a parameter only. A structure containing a member or, recursively, any submember of resource type may be declared as an external declaration only.

7.7.1 Array Declarators

In a non-parameter declaration `T D` where `D` has the form

identifier `[` *constant-expression* `]`

and the type of the identifier in the declaration `T` identifier is "*type-modifier* `T`," the type of the identifier of `D` is "*type-modifier* *n*-array of `T`," where *n* is the result of evaluating the constant expression and specifies the number of elements in the array. The constant expression must have integral type, and value greater than 0.

In a parameter declaration `T D` where `D` has the form

identifier `[` *constant-expression* `]`

and the type of the identifier in the declaration `T` identifier is "*type-modifier* `T`," the type of the identifier of `D` is "reference to *type-modifier n*-array of `T`," where *n* is the result of evaluating the constant expression and specifies the number of elements in the array. The constant expression must have integral type, and value greater than 0.

In a declaration `T D` where `D` has the form

identifier `[]`

and the type of the identifier in the declaration `T` *identifier* is "*type-modifier* `T`," the type of the identifier of `D` is "*type-modifier incomplete*-array of `T`."

An array may be constructed from an arithmetic type, from a structure or union, from a port, timer, channel or channel end, or from another array (to generate a multi-dimensional array). Any type from which an array is constructed must be complete; it must not be an array or structure of incomplete type. This implies that for a multi-dimensional array, only the first dimension may be missing. The type of an object of incomplete array type is completed either by another, complete, declaration for the object (§9.2), or by initialising it (§7.8) or, for a function parameter in which the first dimension is missing, at run-time on entry to the function by the caller.

If `E1` is an array and `E2` an integer, then `E1[E2]` refers to the `E2`th member of `E1`. Arrays are stored by rows (last subscript varies faster) so that the first subscript in the declaration helps determine the amount of storage consumed by an array, but plays no other part in subscript calculations.

7.7.2 Reference Declarators

In a declaration `T D` where `D` has the form

`&` *identifier*

and the type of the identifier in the declaration `T` *identifier* is "*type-modifier* `T`," the type of the identifier of `D` is "reference to *type-modifier* `T`."

A reference declared with `&` may have an arithmetic, structure or union type, and may only be declared as a function parameter.

7.7.3 Nullable Declarator

In a declaration `T D` where `D` has the form

`?` *identifier*

and the type of the identifier in the declaration `T` *identifier* is "*type-modifier* `T`," the type of the identifier of `D` is "nullable *type-modifier* `T`."

A nullable object declared with `?` may have a resource type or a reference type, and may only be declared as a function parameter.

If an object contains a reference to `null`, it is invalid to reference the object except as the argument to a function taking a nullable parameter, as the operand of the `isnull` operator, or as the operand of the `sizeof` operator.

7.7.4 Function Declarators

In a function declaration `T D` where `D` has the form

`D1(` *parameter-type-list* `)`

and the type of the identifier in the declaration `T D1` is "*type-modifier* `T`," the type of the identifier of `D` is "*type-modifier* function with arguments *parameter-type-list* returning `T`." If `T` has the form $\{T_1, \ldots, T_n\}$ then the return type is modified to read "list of types $T_1, \ldots, T_n$." In a function declaration `transaction void D` where `D` has the form

`D1(` *parameter-type-list* `)`

and the type of the identifier in the declaration `T D1` is "*type-modifier* `T`," the type of the identifier of `D` is "*type-modifier* transaction function with arguments *parameter-type-list* returning `void`." In a function declaration `select void D` where `D` has the form

`D1(` *parameter-type-list* `)`

and the type of the identifier in the declaration `T D1` is "*type-modifier* `T`," the type of the identifier of `D` is "*type-modifier* select function with arguments *parameter-type-list* returning `void`."

The syntax of the parameters is:

parameter-type-list	::=	*parameter-list*
	|	*parameter-list* , *parameter-declaration*
parameter-list	::=	*parameter-declaration*
	|	*parameter-list* , *parameter-declaration*
parameter-declaration	::=	$\langle$*dec-specifier*$\rangle^+$ *abstract-or-void-dec*
abstract-or-void-dec	::=	*var-declarator*
	|	*abstract-var-declarator*

The parameter list specifies the types of the parameters. As a special case, the declarator for a function with no parameters has a parameter list consisting solely of the keyword `void`. This is also signified by an empty parameter list. If the parameter type list ends with an ellipsis "`, ...`", then the function may accept more than the number of parameters explicitly described; see §6.3.2.

The only storage class specifier permitted in a parameter's declaration specifier is `register`, and this specifier is ignored unless the function declarator heads a function definition. This specifier has no effect on the behaviour of the function; the extent to which suggestions made by using this specifier are effective is implementation-defined.

Similarly, if the declarators contain identifiers and the function declarator does not prefix a function definition, the identifiers go out of scope immediately. Abstract declarators, which do not mention the identifiers, are discussed in §7.9.

A function declared with the storage class specifier `service` may declare only variables of type `chanend`.

> Old-style C function declarations (see K&R §A8.6.3) are unsupported.

7.8 Initialisation

When an object is declared, its init-var-declarator may specify an initial value for the identifier being declared. The initialiser is preceded by =, and is either an expression, or a list of initialisers nested in braces.

initialiser	::=	*on-statement*$_{\text{opt}}$ *expression*
	|	{ *initialiser-list* }
	|	{ *initialiser-list* , }
initialiser-list	::=	*initialiser*
	|	*initialiser-list*

On statements are discussed in §8.8. If more than one `on`-statement is used with the same variable declaration, then all of these statements must refer to the same core.

All the expressions in the initialiser for a static object or array must be constant expressions as described in §6.19. The expressions in the initialiser for an `auto` or `register` object must likewise be constant expressions if the initialiser is a brace-enclosed list. However, if the initialiser for an automatic object is a single expression, it need not be a constant expression, but must have appropriate type for assignment to the object.

Timers, channels and cores must not be explicitly initialised. Timers not declared `extern` are initialised, at run-time, to refer to an unaliased hardware timer. Channels not declared `extern` are initialised, at run-time, to refer to two unaliased hardware channel ends that are connected together to create a point-to-point communication link. Ports not declared `extern`, and not explicitly initialised, are initialised with an implementation-defined value.

A static object that is not a timer, channel or port, and is not explicitly initialised, is initialised as if its expression (or its members) were assigned the constant 0. The initial value of an automatic object with arithmetic type not explicitly initialised is undefined.

The initialiser for an object of arithmetic type is a single expression, possibly in braces. The expression is assigned to the object. The initialiser for a port is a single constant expression, possibly in braces. The expression is assigned to the object; its interpretation and validity is implementation-defined.

The initialiser for a structure is either an expression of the same type, or a brace-enclosed list of initialisers for its members in order. If there are fewer initialisers in the list than members of the structure, the trailing members are initialised with 0. There may not be more initialisers than members.

The initialiser for an array is a brace-enclosed list of initialisers for its members. If the array has unknown size, the number of initialisers determines the size of the array, and its type becomes complete. If the array has fixed size, the number of initialisers may not exceed the number of members of the array; if there are fewer, the trailing members are initialised with 0.

As a special case, a character array may be initialised by a string literal (braces are optional); successive characters of the string initialise successive members of the array. If the array has unknown size, the number of characters in the string, including the terminating null character, determines its size; if its size is fixed, the number of characters in the string, not counting the terminating null character, must not exceed the size of the array.

The initialiser for a union is either a single expression of the same type, or a brace-enclosed initialiser for the first member of the union.

An *aggregate* is a structure or array. If an aggregate contains members of aggregate type, the initialisation rules apply recursively. Braces may be elided in the initialisation as follows: if the initialiser for an aggregate's member that is itself an aggregate begins with a left brace, then the succeeding comma-separated list of initialisers initialises the members of the subaggregate; it is erroneous for there to be more initialisers than members. If, however, the initialiser for a subaggregate does not begin with a left brace, then only enough elements from the list are taken to account for the members of the subaggregate; any remaining members are left to initialise the next member of the aggregate of which the subaggregate is a part.

7.9 Type Names

In several contexts (to specify type conversions explicitly with a cast, in a reinterpretation, and to declare parameter types in function declarators) it is necessary to supply the name of a data type. This is accomplished using a *type name*, which is syntactically a declaration for an object of that type omitting the name of the object.

type-name	::=	$\langle$*specifier-or-qualifier*$\rangle^{+}$ *abstract-var-declarator*
abstract-var-declarator	::=	$\langle$*dimension-size*$\rangle$*

7.10 Typedef

Declarations whose storage class specifier is `typedef` do not declare objects; instead they define identifiers that name types (called typedef names).

typedef-name	::=	*identifier*

A `typedef` declaration attributes a type to each name among its declarators in the usual way (see §7.7). Thereafter, each such typedef name is syntactically equivalent to a type specifier keyword for the associated type. `typedef` does not introduce new types, only synonyms for types that could be specified in another way. Typedef names may be redeclared in an inner scope, but a non-empty set of type specifiers must be given.

7.11 Type Equivalence

Two type specifier lists are equivalent if they contain the same set of type specifiers, taking into account that some specifiers can be implied by others (for example, `long` alone implies `long int`, `register` in formals is ignored). Structures and unions with different tags are distinct, and a tagless structure or union specifies a unique type.

Two types are the same if their abstract declarators (§7.9), after deleting any function parameter identifiers, are the same up to equivalence of type specifier lists. Array sizes (including the size of array parameters) are significant. For each parameter qualified `const` that is not a reference type, its type for this comparison is the unqualified version of its type.

8 Statements

Except as described, statements are executed in sequence. Statements are executed for their effect, and do not have values. They fall into several groups.

statement	::=	*simple-statement*$_{\text{opt}}$;
	|	*compound-statement*
	|	*selection-statement*
	|	*iteration-statement*
	|	*jump-statement*
	|	*parallel-statement*
	|	*transaction-statement*
simple-statement	::=	*expression-statement*
	|	*multiple-assignment*
	|	*input*
	|	*output*

A semicolon by itself is called a null statement; it is often used to supply an empty body to an iteration statement.

8.1 Expression Statement

The syntax of an expression statement is:

expression-statement	::=	*expression*
expression	::=	*assignment-expression*

Most expression statements are assignments or function calls. An expression statement must not have resource type. All side effects from the expression are completed before the next statement is executed.

8.2 Multiple Assignment Statement

The syntax of a multiple assignment statement is:

multiple-assignment	::=	`{` *return-list* `}` *arithmetic-operator function-call*
return-list	::=	*optional-variable*
	|	*optional-variable* `,` *return-list*
optional-variable	::=	*variable-reference*
	|	`void`

The function must have return type "list of types $T_1,\ldots,T_n$" where n is the same as the number of optional variables in the return list.

The rules for assignment (see §6.17) apply to each of the variables in the return list: the ith value returned by the function replaces that of the object referred to by the ith optional variable reference. If the optional variable reference is `void` then the value is discarded.

A variable which is changed in the subscript of an optional variable may not appear in any other optional variable or in the function call, including the arguments to the function. A variable which is changed in the function call, including arguments to function may not appear in any optional variable. These rules apply recursively to variables which are changed or appear in functions called in the optional variables or the function call.

A variable which is changed by the assignment may not appear in any other optional variable or recursively appear in functions called in any other optional variable.

If any of the objects assigned to are the same as one another then the assignment is invalid.

8.3 Input and Output Statements

An input statement receives a value from a channel end, port or timer, and assigns the received value to an object.

input	::=	*resource* $time_{opt}$ $predicate_{opt}$ *input-operator dest* $input\text{-}timestamp_{opt}$
resource	::=	*variable-reference*
time	::=	`@` *expression*
input-operator	::=	`:>`
	|	`:> >>`
dest	::=	*declared-var-reference*

input-timestamp	::=	`@` *declared-var-reference*
declared-var-reference	::=	⟨*declaration-specifier*⟩$^+$ *identifier*$_{\text{opt}}$
	|	*variable-reference*
predicate	::=	`when` *function-call*

The resource must name either a channel end, port or timer. If the resource names a channel end or timer then neither a *time* nor an *input-timestamp* is allowed. If the resource names a channel end then a *predicate* is not allowed. If the resource names a port then the port must not be qualified out and the destination variable must have arithmetic type.

If a *time* is provided then the time expression must have arithmetic type. The input is said to be *timed*.

If an *input-timestamp* is provided then the *declared-var-reference* must name a modifiable lvalue with arithmetic type. The input is said to be *timestamped*.

If a *predicate* is provided then the named function must have been declared to return `void` and from its parameter list there must be precisely one port or timer declaration, which must be qualified `void`. The input is said to be *predicated*. The supported predicates are implementation-defined. The function call is shortcutted: the resource variable must not be passed as an argument; it is passed implicitly as the port or timer argument.

A *declared-var-reference* must be a modifiable lvalue if an identifier is named. It must not define a new type. If the resource names a port or timer then the lvalue must not reference a structure or union; if no identifier is given then the type must not specify a structure or union, but it may specify `void`. If no declaration specifiers are provided then the type of the variable must not be qualified with `const`; if it is a structure or union, it must not have any member or, recursively, submember qualified with `const`. If any declaration specifiers are provided then the variable reference is also a declaration; the specifiers must not contain `typedef` but may contain `const`.

A variable which is changed by any part of the input may not, except as described below, appear in any other part of the input. If the *declared-var-reference* is a *variable-reference* then the identifier named may appear in any other parts of the input, as long as the variables named by the identifiers in these parts are not changed. Additionally, the variable which is written by the input-timestamp may not appear in the dest, and the variable which is written by the dest may not appear in the *input-timestamp*. These rules apply recursively to all variables which are changed in functions called by the input.

The first variable declared in an input begins an inner scope which is understood to begin immediately preceding the declaration and which persists to the end of the input. If the input appears in the case of a select then this scope continues to persist to the end of the statement list after the colon.

If the resource names a channel end or timer, or the destination identifier is omitted, then the `:>` >> operator is not allowed.

An output statement transmits the value of an expression to a channel end or port.

output ::= *resource* $time_{\text{opt}}$ *output-operator expression* $output\text{-}timestamp_{\text{opt}}$

output-operator ::= `<:`
| `<: >>`

output-timestamp ::= `@` *variable-reference*

The resource must name a channel end or port. If the resource names a channel end then neither a *time* nor an *output-timestamp* is allowed. If the resource names a port then the port must not be qualified `in` and the output expression must have arithmetic type; otherwise the output expression must either have arithmetic type, or must be a structure or union.

If the resource names a channel end then the `<: >>` operator is not allowed. If the `<: >>` is specified then the output expression must be a modifiable lvalue.

If a *time* is provided then an *output-timestamp* is not allowed. The time expression must have arithmetic type. The output is said to be *timed*.

If an *output-timestamp* is provided then the variable reference must be a modifiable lvalue with arithmetic type. The output is said to be *timestamped*.

A variable which is changed by any part of the output may not, except as described below, appear in any other part of the output. The identifier named by the *output-timestamp* may appear in any other parts of the output as long as the variables named by the identifiers in these parts are not changed. These rules apply recursively to all variables which are changed in functions called by the output.

> Input and output statements are new; I/O operations are conventially performed using interrupts and system calls (via library routines in C).

8.3.1 Channel Input and Output

An input on a channel end causes the processor to wait until a matching outputter is ready in a parallel statement (see §8.8) before receiving a value. If the type of an input variable is specified but the identifier is missing then the received value is ignored. See §11 for the meaning of an input in a channel communication.

An output on a channel causes the processor to wait until a matching inputter is ready in a parallel statement before sending the value. See §11 for the meaning of an output in a channel communication.

8.3.2 Port Input and Output

An input from a port causes the specified port to provide the processor a value. If the port transfer width is *w* bits, these *w* bits are assigned to the least significant bits of a variable with the port's notional transfer type (see §3.2) with any remaining bits being set to zero. If the type of an input variable is specified but an identifier is missing, or if a `void` type is specified, then this input variable is ignored. If the input is used with the `:>` `>>` operator, the destination variable is right-shifted by *w* bits and the bitwise inclusive-or of this value and the input variable is then assigned to the destination variable; otherwise the input variable is assigned to the destination variable.

If a `when` condition is provided, the function and its arguments are provided to the port before performing the input.

An output to a port causes the output expression to be first cast to the port's notional transfer type and then provided to the port. If the output is used with the `<:` `>>` operator, the output variable is then right-shifted by *w* bits.

If the input or output is timed, the value specified by time is cast to the port's notional counter type and provided to the port before performing the input or output.

If the input or output is timestamped, *t* bits are assigned to the least significant bits of a variable with the port's notional counter type (see §3.2) with any remaining bits being set to zero; this variable is then assigned to the timestamp variable.

The meaning of inputs and outputs on ports is implementation-defined.

8.3.3 Timer Input

An input from a timer causes the timer to provide the current value of its counter. This value is assigned to the least significant bits of a variable with the timer's notional counter type (see §3.2) with any remaining bits being set to zero. If the type of an input variable is specified but an identifier is missing, or if a `void` type is specified, then this input variable is ignored; otherwise the input variable is assigned to the destination variable.

8.4 Compound Statement

So that several statements can be used where one is expected, the compound statement (or "block") is provided. The body of a function definition is a compound statement.

compound-statement ::= `{` ⟨*var-declaration*⟩* ⟨*statement*⟩* `}`

If an identifier in the *var-declaration-list* was in scope outside the block, the outer declaration is suspended within the block (see §10.1). An identifier may be declared only once in the same block. These rules apply to identifiers in the same name space (§10); identifiers in different name spaces are treated as distinct.

Initialisation of automatic objects is performed each time the block is entered at the top, and proceeds in the order of the declarators. Initialisation of `static` objects is performed only once, before the program begins execution.

8.5 Selection Statements

Selection statements choose one of several flows of control.

selection-statement	::=	`if` (*expression*) *statement*
	|	`if` (*expression*) *statement* `else` *statement*
	|	`switch` (*expression*) { ⟨*labelled-statement*⟩$^+$ }
	|	`select` { ⟨*guarded-statement*⟩$^+$ }
labelled-statement	::=	`case` *constant-expression* : ⟨*statement*⟩*
	|	`default` : ⟨*statement*⟩*
guarded-statement	::=	`case` *replicator*$_{\text{opt}}$ *enable-exp*$_{\text{opt}}$ *input* : ⟨*statement*⟩*
	|	`case` *replicator*$_{\text{opt}}$ *enable-exp*$_{\text{opt}}$ *function-call* : ⟨*statement*⟩*
	|	`case` *replicator*$_{\text{opt}}$ *enable-exp*$_{\text{opt}}$ *slave-statement* : ⟨*statement*⟩*
	|	`default` : ⟨*statement*⟩*
	|	`case` *function-call* ;
replicator	::=	(`int` *variable* = *expression* ; *expression* ; *expression*)
enable-exp	::=	*expression* =>

In both forms of the `if` statement, the expression, which must have arithmetic type, is evaluated, including all side effects, and if it compares unequal to 0, the first substatement is executed. In the second form, the second substatement is executed if the expression is 0. The `else` ambiguity is resolved by connecting an `else` with the last encountered `else`-less `if` at the same block nesting level.

The `switch` statement causes control to be transferred to one of several case statements depending on the value of the expression, which must have integral type. The controlling expression undergoes integral promotion (§5.1), and the case constants are converted to the promoted type. No two of the case constants in the same switch may have the same value after conversion. There may also be at most one `default` label associated with a switch.

When the `switch` statement is executed, its expression is evaluated, including all side effects, and compared with each case constant. If one of these case constants is equal to the value of the expression, control passes to the statement of the matched `case` label. If no case constant matches the expression, and if there is a `default` label, control passes to the default-labelled statement. If no case matches, and if there is no `default`, then none of the substatements of the switch is executed.

The `select` statement causes control to be transferred to one of several guarded case statements. A guarded statement may consist of an optional replicator and an optional expression followed by an input (§8.3), a slave transaction statement (§8.9) or a function call, followed by a colon and a list of zero or more statements.

In a replicator, the third expression must either add or subtract a constant expression to the variable declared by the replicator. A replicator is short-hand for multiple cases, and has the same meaning as if the code was expanded as with a `for` loop. In addition, if the initialiser is a constant expression and the second expression is a relational expression that compares the variable declared by the replicator to a constant expression, the variable declared by the replicator is treated as a constant expression in the replicator body. The declared variable may not be modified outside of the replicator.

If the statement before the colon is a call to a transaction function (§9.1.1) then this is considered shorthand for a slave transaction statement that performs the call. The enable expression must have arithmetic type, and it must not modify a local variable, static variable or reference parameter; any functions called within the expression, recursively, must not modify a static variable, reference parameter, or perform an input or output. The modification rules that apply to the enable expression also apply to the arguments of a call to a select function; the rules also apply to an input statement that appears before the colon, except that the input lvalue is (by definition) modified. An input guard that causes any observable behaviour on a port prior to being selected is invalid. There may be at most one `default` label associated with a select.

A guarded statement may also consist of a call to a select function (see §9.1.2) followed by a semicolon. The rules that apply to the enable expression also apply to the arguments of a call to a select function. The ports, timers and channel ends named before each colon, and as arguments to a select function, must be distinct.

When the `select` statement is executed, each guard that contains no enable expression is enabled. For each guard containing an enable expression, the expression is evaluated and, if it compares unequal to 0, the case is enabled. The behaviour of a call to a select function is the same as if the cases of the select function were included inline in the select.

Following the enabling sequence, if no cases are enabled then either the `default` case is executed, if provided, or none of the substatements of the select is executed and the select never completes (it deadlocks). Otherwise, the select waits until an input or transaction in one of the enabled cases is ready and performs the corresponding input or transaction. If more than one of these inputs or transactions is ready then the choice of which is executed is made nondeterministically.

After performing an input or transaction, the statements following the colon of the selected case are executed.

> The statements after the colon in each `select` case statement must terminate with a `break` or `return`, so that control never flows from one case statement to the next.

8.6 Iteration Statements

Iteration statements specify looping.

iteration-statement	::=	`while (` *expression* `)` *statement*
	|	`do` *statement* `while (` *expression* `) ;`
	|	`for (` *for-init*$_{\text{opt}}$ `;` *expression*$_{\text{opt}}$ `;` *simple-list*$_{\text{opt}}$ `)` *statement*
for-init	::=	*var-declaration*
	|	*simple-list*
simple-list	::=	*simple-statement*
	|	*simple-statement* `,` *simple-list*

In the while and do statements, the substatement is executed repeatedly so long as the value of the expression remains unequal to 0; the expression must have arithmetic type. With while, the test, including all side effects from the expression, occurs before each execution of the statement; with do, the test follows each iteration.

A for statement may declare a variable (see § 3), whose scope begins immediately after the declaration and persists to the end of the statement; if present, the variable initialiser is evaluated once. Alternatively, if a list of simple statements is provided, the statements are executed once. The expression must have arithmetic type; it is evaluated before each iteration, and if it is equal to 0, the for is terminated. The optional list of simple statements following the second semicolon is evaluated after each iteration. Any of these three components may be dropped; a missing test expression makes the implied test equivalent to testing a non-zero constant.

8.7 Jump Statements

Jump statements transfer control unconditionally.

jump-statement	::=	*continue ;*
	|	break ;
	|	return *expression*$_{\text{opt}}$;
	|	return { *expression-list* } ;

A continue statement may appear only within an iteration statement, and may not appear in a parallel, master or slave statement, unless that statement contains an iteration statement in which it is enclosed. It causes control to pass to the loop-continuation portion of the smallest enclosing such statement.

A break statement may appear only in an iteration statement, a switch statement or a select statement, and may not appear in a parallel, master or slave statement, unless that statement contains an iteration, switch or select statement in which it is enclosed. It terminates execution of the smallest enclosing such statement; control passes to the statement following the terminated statement.

A function returns to its caller by the return statement. A return statement may not appear in a parallel, master or slave statement. When return is followed by an expression, the value is returned to the caller of the function. The expression is converted, as if by assignment, to the type returned by the function in which it appears.

When return is followed by an list of expressions in braces, the list of values is returned to the caller of the function. For a return with n expressions, the return type of the function must be "list of types $T_1,\ldots,T_n$." For all expressions (i=1..n), the ith expression is converted, as if by assignment, to the ith type returned by the function in which it appears.

Flowing off the end of a function is equivalent to a return with no expression. In either case, the returned value is undefined.

goto is unsupported.

8.8 Concurrency Statement

So that several statements can be executed concurrently, the parallel statement is provided.

parallel-statement ::= `par` *replicator*$_{\text{opt}}$ `{` ⟨*thread*⟩* `}`

thread ::= *on-statement*$_{\text{opt}}$ *statement*

Replicators are discussed in §8.5. In addition, the initialiser must assign a constant expression, and the second expression must be a relational expression that compares the variable declared by the replicator to a constant expression. The relation operator may not be equality or inequality and the condition must be satisfiable for some value of the declared variable (for example, `x > MAX_INT` is disallowed).

An *on-statement* is only permitted if it appears in a *parallel-statement* that is either the only statement in the enclosing function, or if it is one of two statements of a function *compound-statement*, the second being a return statement that returns a constant expression that evaluates to 0.

`on` does not change the behaviour of the statement it prefixes.

Values may be passed between concurrent statements by communication on channels (§3.2) using input and output statements (§8.3).

Variables and channels used in parallel statements are subject to usage rules which prevent them from being accidentally shared between statements in potentially dangerous ways, as described below.

A variable which is changed by assignment or input in one of the statements of a `par` may not appear in any other statement of the `par`. This rule applies recursively to all variables which are changed by assignment or input in a function that is called by a statement of a `par`.

(By implication, a variable may appear in expressions in any number of statements of a `par` so long as it is not assigned or input in any of these statements.)

A channel may not be used in more than two statements of a `par`. Channel ends, ports and timers may not be used in more than one statement of a `par`.

If a statement contains of a number of sub-statements, such as a *compound-statement* (§8.4), then all of the sub-statements are considered together as a single statement for the purpose of this rule.

8.9 Transaction Statement

So that several communications over a channel can be logically grouped together, the transaction statement is provided.

transaction-statement	::=	*slave-statement*
	|	*master-statement*
slave-statement	::=	`slave` *statement*
master-statement	::=	`master` *statement*

All inputs and outputs within `master` or `slave` are logically part of the same transaction; the extent to which the underlying communication protocols are optimised for transaction communications is implementation-defined.

The statements must reference precisely one channel end, which is said to be the *transactor*. If the variable reference designating the transactor contains any array indices then the indices must be constant expressions. The transactor must not name a streaming channel.

Within a transaction statement, inputs and outputs on any channel end other than the transactor is prohibited; using a channel end other than the transactor as an argument to a function is prohibited; using the transactor as an argument to a function that is not a transaction function is prohibited; introducing a nested transaction statement is prohibited; and declaring a channel (in the statement or, recursively, in any function called within the transaction) is prohibited.

9 External Declarations

The unit of input provided to the XC compiler is called a translation unit; it consists of a sequence of external declarations, which are either declarations or function definitions.

translation-unit	::=	⟨*external-declaration*⟩$^{+}$
external-declaration	::=	*declaration*
	|	*function-definition*

The scope of external declarations persists to the end of the translation unit in which they are declared.

9.1 Function Definitions

Function definitions have the form:

function-definition	::=	*fnc-declaration compound-statement*
	|	*trn-declaration compound-statement*
	|	*sel-declaration* `{` ⟨*guarded-statement*⟩$^{+}$ `}`

The only *storage-class* specifiers allowed among the declaration specifiers are `extern`, `static` or `inline`; see §10.2 for the effect. The ellipses "`, ...`" operator is not allowed in function definitions.

A function may return an arithmetic type, a structure, a union or `void`, but not a resource type, a function or an array. Alternatively it may return a list of any combination of arithmetic types, structures and unions. A function may not return a structure containing a member or, recursively, any submember of resource type.

Unless the parameters consist solely of `void`, indicating that the function takes no parameters, each declarator in the parameter list must contain an identifier. The parameters are understood to be declared just after the beginning of the compound statement constituting the function's body, and thus the same identifiers must not be redeclared there (although they may be redeclared in inner blocks). During the call to a function, the arguments are converted as necessary and assigned to the parameters; see §6.3.2.

9.1.1 Transaction Functions

A function declaration modified by the keyword `transaction` is a transaction function (see §7.7.4). The function body consists of a list of statements, which is by definition a transaction statement (see §8.9). The function must declare precisely one channel end in its parameter list, which is by definition the transactor.

9.1.2 Select Functions

A function declaration modified by the keyword `select` is a select function (see §7.7.4). The function body consists of a list of guarded statements, which is by definition a select statement (see §8.5). The guards of a select function may not contain replicators or transactors.

9.2 External Declarations

External declarations specify the characteristics of objects, functions and other identifiers. The term "external" refers to their location outside functions, and is not directly connected with the `extern` keyword; the storage class for an externally-declared object may be left empty, or it may be specified as `extern` or `static`.

Several external declarations for the same identifier may exist within the same translation unit if they agree in type and linkage, and if there is at most one definition for the identifier.

Two declarations for an object or function are deemed to agree in type under the rules discussed in §7.11. In addition, if the declarations differ because one type is an incomplete structure or union and the other is the corresponding completed type with the same tag, the types are taken to agree. If one type is an incomplete array type (§7.7.1) and the other is a completed array type, the types, if otherwise identical, are also taken to agree.

If the first external declaration for a function or object includes the `static` specifier, the identifier has *file-scope* (*internal linkage*); otherwise it has *program-scope* (*external linkage*). Linkage is discussed in §10.2.

An external declaration for an object is a definition if it has an initialiser. An external object declaration that does not have an initialiser, and does not contain the `extern` specifier, is a *tentative definition*. If a definition for an object appears in a translation unit, any tentative definitions are treated as redundant declarations. If no definition for the object appears in the translation unit, all its tentative definitions become a single definition with initialiser 0.

Each object must have exactly one definition. For objects with internal linkage, the rules apply separately to each translation unit. For objects with external linkage, it applies to the entire program.

10 Scope and Linkage

There are two kinds of scope to consider: first, the *lexical scope* of an identifier, which is the region of the program text within which the identifier's characteristics are understood; and second, the scope associated with objects with external linkage, which determines the connections between identifiers in separately compiled translation units.

10.1 Lexical Scope

Identifiers fall into several name spaces that do not interfere with one another; the same identifier may be used for different purposes, even in the same scope, if the uses are in different name spaces. These classes are: objects and functions; tags of structures and unions; and members of each structure or union individually.

The lexical scope of an object or function identifier in an external declaration begins at the end of its declarator and persists to the end of the translation unit in which it appears. The scope of a parameter of a function definition begins at the start of the block defining the function, and persists through the function; the scope of a parameter in a function declaration ends at the end of the declarator. The scope of an identifier declared at the head of a block begins at the end of its declarator, and persists to the end of the block. The scope of a structure or union begins at its appearance in a type specifier, and persists to the end of the translation unit (for declarations at the external level) or to the end of the block (for declarations within a function).

If an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of the identifier outside the block is suspended until the end of the block.

10.2 Linkage

Within a translation unit, all declarations of the same object or function identifier with internal linkage refer to the same thing, and the object or function is unique to that translation unit. All declarations for the same object or function identifier with external linkage refer to the same thing, and the object or function is shared by the entire program.

The first external declaration for an identifier gives the identifier internal linkage if the `static` specifier is used, external linkage otherwise.

An inline definition (§7.3) does not provide an external definition for the function and does not forbid an external definition. An inline definition provides an alternative to an external definition which may be used it implement any call to the function in the same translation unit. It is unspecified whether a call to the function uses the inline definition or the external definition.

11 Channel Communication

A channel communication occurs when, on the same channel,

- an output is executed in parallel with an input, or
- a master transaction is executed in parallel with a slave transaction.

An output executed in parallel with a slave transaction is invalid; a master transaction executed in parallel with an input is invalid.

Outside a transaction, an output-input communication in which the number of bytes output is unequal to the number of bytes input is invalid. Inside a transaction, if all communications are valid individually then the transaction is also valid. Additionally, if a communication occurs in which the number of bytes output is unequal to the number of bytes input then whether or not the transaction is invalid, and the value communicated, is implementation-defined.

An invalid communication within a transaction need not cause the transaction to become invalid until slave transaction statement goes out of scope.

The meaning of an output-input communication in which the type of the output expression *e* is the same as the type of the input variable *v* is the same as the assignment *v* = *e*. If the types are different and the communication is not invalid then the meaning is the assignment *v* = (*e*, *type*(*v*)) (see §6.3.4).

12 Invalid Operations

An operation that is syntactically legal but for some reason or under some circumstances is semantically invalid may be treated in one of three ways:

- It may be reported as a compiler error.
- It may have implementation-defined behaviour, for example the processor could issue a trap, and a trap handler could terminate the program.
- It may result in undefined behaviour.

If at time *t* a program is guaranteed to execute some sequence of events that cause it to become invalid at some time in the future *t*+*n* then it is permitted to become invalid any time during [*t*..*t*+*n*]. This allows an implementation to improve code efficiency, for example by relocating safety checks outside of loops.

13 Preprocessing

The preprocessor specification is defined to be the same as with C99 §6.10 (see X1805), with the following exceptions:

- The macro `__XC__` is defined as 1.
- The macros `__STDC__`, `__STDC_HOSTED__` and `__STD_VERSION__` are not defined.

14 Grammar

Below is a summary of the grammar given throughout this document. The grammar has undefined terminal symbols integer-constant, character-constant, identifier, string and enumeration-constant; words and symbols wrtten in `typewriter` are terminals given literally.

translation-unit ::= ⟨*external-declaration*⟩$^+$

external-declaration ::= *declaration*
| *function-definition*

function-definition ::= *fnc-declaration compound-statement*
| *trn-declaration compound-statement*
| *sel-declaration* `{` ⟨*guarded-statement*⟩$^+$ `}`

declaration ::= *on-statement*$_{\mathrm{opt}}$ *actual-declaration*

actual-declaration ::= *var-declaration*
| *fnc-declaration* `;`
| *trn-declaration* `;`
| *sel-declaration* `;`

on-statement ::= `on` *variable-reference* `:`

var-declaration ::= ⟨*dec-specifierier*⟩* *init-var-declarator-list*$_{\mathrm{opt}}$ `;`

fnc-declaration ::= ⟨*dec-specifierier*⟩* *fnc-declarator*
| `{` *dec-specifier-list* `}` *fnc-declarator*

trn-declaration ::= ⟨*dec-specifierier* ⟩* `transaction` *fnc-declarator*

sel-declaration ::= ⟨*dec-specifierier*⟩* `select` *fnc-declarator*

dec-specifier-list ::= ⟨*dec-specifier*⟩*
| *dec-specifier-list* `,` ⟨*dec-specifierier*⟩*

dec-specifier ::= *storage-class-specifier*
| *type-specifier*
| *type-qualifier*
| `inline`

storage-class-specifier	::=	`auto`
	|	`register`
	|	`static`
	|	`extern`
	|	`typedef`
	|	`service`
type-specifier	::=	`void`
	|	`char`
	|	`short`
	|	`int`
	|	`long`
	|	`signed`
	|	`unsigned`
	|	`chan`
	|	`chanend`
	|	`port`
	|	`port:`*n*
	|	`timer`
	|	`core`
	|	*struct-or-union-specifier*
	|	*enum-specifier*
	|	*typedef-name*
type-qualifier	::=	`const`
	|	`volatile`
	|	`in`
	|	`out`
	|	`buffered`
	|	`streaming`
struct-or-union-specifier	::=	*struct-or-union* $identifier_{opt}$ `{` $\langle member \rangle^{+}$ `}`
	|	*struct-or-union identifier*
struct-or-union	::=	`struct`
	|	`union`
init-var-declarator-list	::=	*init-var-declarator*
	|	*init-var-declarator* `,` *init-var-declarator-list*
init-var-declarator	::=	*var-declarator* $\langle$= *initialiser* $\rangle_{opt}$
member	::=	$\langle specifier\text{-}or\text{-}qualifier \rangle^{+}$ *struct-var-declarator-list* `;`

specifier-or-qualifier	::=	*type-specifier*
	|	*type-qualifier*
struct-var-declarator-list	::=	*var-declarator*
	|	*var-declarator* , *struct-var-declarator-list*
enum-specifier	::=	`enum` $\textit{identifier}_{\text{opt}}$ { *enumerator-list* }
	|	`enum` $\textit{identifier}_{\text{opt}}$ { *enumerator-list* , }
	|	`enum` *identifier*
enumerator-list	::=	*enumerator*
	|	*enumerator-list* , *enumerator*
enumerator	::=	*identifier*
	|	*identifier* = *constant-expression*
var-declarator	::=	*identifier* ⟨*dimension-size* ⟩*
	|	& *identifier*
	|	? *identifier* ⟨*dimension-size*⟩*
	|	& ? *identifier*
fnc-declarator	::=	*identifier* ($\textit{parameter-type-list}_{\text{opt}}$)
dimension-size	::=	[$\textit{constant-expression}_{\text{opt}}$]
parameter-type-list	::=	*parameter-list*
	|	*parameter-list* , *parameter-declaration*
parameter-list	::=	*parameter-declaration*
	|	*parameter-list* , *parameter-declaration*
parameter-declaration	::=	⟨*dec-specifier*⟩+ *abstract-or-void-dec*
abstract-or-void-dec	::=	*var-declarator*
	|	*abstract-var-declarator*
initialiser	::=	$\textit{on-statement}_{\text{opt}}$ *expression*
	|	{ *initialiser-list* }
	|	{ *initialiser-list* , }
initialiser-list	::=	*initialiser*
	|	*initialiser-list*
type-name	::=	⟨*specifier-or-qualifier*⟩+ *abstract-var-declarator*
abstract-var-declarator	::=	⟨*dimension-size*⟩*

typedef-name ::= *identifier*

statement ::= *simple-statement*$_{\text{opt}}$ `;`
| *compound-statement*
| *selection-statement*
| *iteration-statement*
| *jump-statement*
| *parallel-statement*
| *transaction-statement*

simple-statement ::= *expression-statement*
| *multiple-assignment*
| *input*
| *output*

compound-statement ::= `{` ⟨*var-declaration*⟩* ⟨*statement*⟩* `}`

selection-statement ::= `if` `(` *expression* `)` *statement*
| `if` `(` *expression* `)` *statement* `else` *statement*
| `switch` `(` *expression* `)` `{` ⟨*labelled-statement*⟩+ `}`
| `select` `{` ⟨*guarded-statement*⟩+ `}`

labelled-statement ::= `case` *constant-expression* `:` ⟨*statement*⟩*
| `default` `:` ⟨*statement*⟩*

guarded-statement ::= `case` *replicator*$_{\text{opt}}$ *enable-exp*$_{\text{opt}}$ *input* `:` ⟨*statement*⟩*
| `case` *replicator*$_{\text{opt}}$ *enable-exp*$_{\text{opt}}$ *function-call* `:` ⟨*statement*⟩*
| `case` *replicator*$_{\text{opt}}$ *enable-exp*$_{\text{opt}}$ *slave-statement* `:` ⟨*statement*⟩*
| `default` `:` ⟨*statement*⟩*
| `case` *function-call* `;`

replicator ::= `(` `int` *variable* `=` *expression* `;` *expression* `;` *expression* `)`

enable-exp ::= *expression* `=>`

iteration-statement ::= `while` `(` *expression* `)` *statement*
| `do` *statement* `while` `(` *expression* `)` `;`
| `for` `(` *for-init*$_{\text{opt}}$ `;` *expression*$_{\text{opt}}$ `;` *simple-list*$_{\text{opt}}$ `)` *statement*

for-init ::= *var-declaration*
| *simple-list*

simple-list ::= *simple-statement*
| *simple-statement* `,` *simple-list*

jump-statement	::=	*continue ;*
	|	`break ;`
	|	`return` *expression*$_{opt}$ `;`
	|	`return {` *expression-list* `} ;`
parallel-statement	::=	`par` *replicator*$_{opt}$ `{` ⟨*thread*⟩* `}`
thread	::=	*on-statement*$_{opt}$ *statement*
transaction-statement	::=	*slave-statement*
	|	*master-statement*
slave-statement	::=	`slave` *statement*
master-statement	::=	`master` *statement*
expression-statement	::=	*expression*
expression	::=	*assignment-expression*
assignment-expression	::=	*conditional-expression*
	|	*variable-reference assignment-operator assignment-expression*
assignment-operator	::=	*one of*
		`= *= /= %= += -= <<= >>= &= ^= |=`
conditional-expression	::=	*logical-OR-expression*
	|	*logical-OR-expression* `?` *expression* `:` *conditional-expression*
constant-expression	::=	*conditional-expression*
logical-OR-expression	::=	*logical-AND-expression*
	|	*logical-OR-expression* `||` *logical-AND-expression*
logical-AND-expression	::=	*inclusive-OR-expression*
	|	*logical-AND-expression* `&&` *inclusive-OR-expression*
inclusive-OR-expression	::=	*exclusive-OR-expression*
	|	*inclusive-OR-expression* `|` *exclusive-OR-expression*
exclusive-OR-expression	::=	*AND-expression*
	|	*exclusive-OR-expression* `^` *AND-expression*
AND-expression	::=	*equality-expression*
	|	*AND-expression* `&` *equality-expression*

equality-expression ::= *relational-expression*
| *equality-expression* == *relational-expression*
| *equality-expression* != *relational-expression*

relational-expression ::= *shift-expression*
| *relational-expression* < *shift-expression*
| *relational-expression* > *shift-expression*
| *relational-expression* <= *shift-expression*
| *relational-expression* >= *shift-expression*

shift-expression ::= *additive-expression*
| *shift-expression* << *additive-expression*
| *shift-expression* >> *additive-expression*

additive-expression ::= *multiplicative-expression*
| *additive-expression* + *multiplicative-expression*
| *additive-expression* - *multiplicative-expression*

multiplicative-expression ::= *cast-expression*
| *multiplicative-expression* * *cast-expression*
| *multiplicative-expression* / *cast-expression*
| *multiplicative-expression* % *cast-expression*

cast-expression ::= *unary-expression*
| (*type-name*) *cast-expression*

unary-expression ::= *postfix-expression*
| ++ *variable-reference*
| -- *variable-reference*
| *unary-operator cast-expression*
| `sizeof` *unary-expression*
| `sizeof` (*type-name*)
| `isnull` (*unary-expression*)

unary-operator ::= *one of*
`+ - ~ !`

postfix-expression ::= *primary-expression*
| *variable-reference* ++
| *variable-reference* --

primary-expression	::=	*variable-reference*
	|	*function-call*
	|	*constant*
	|	*string*
	|	(*expression*)
variable-reference	::=	*identifier*
	|	*variable-reference* [*expression*]
	|	*variable-reference* . *identifier*
	|	(*variable-reference* , *type-name*)
function-call	::=	*identifier* (*expression-list*$_{\text{opt}}$)
expression-list	::=	*expression*
	|	*expression* , *expression-list*
multiple-assignment	::=	*{ return-list } assignment-operator function-call*
return-list	::=	*optional-variable*
	|	*return-list , optional-variable*
optional-variable	::=	*variable-reference*
	|	`void`
input	::=	*resource time*$_{\text{opt}}$ *predicate*$_{\text{opt}}$ *input-operator dest input-timestamp*$_{\text{opt}}$
resource	::=	*variable-reference*
time	::=	`@` *expression*
input-operator	::=	`:>`
	|	`:> >>`
dest	::=	*declared-var-reference*
input-timestamp	::=	`@` *declared-var-reference*
declared-var-reference	::=	$\langle$*declaration-specifier*$\rangle^{+}$ *identifier*$_{\text{opt}}$
	|	*variable-reference*
predicate	::=	`when` *function-call*
output	::=	*resource time*$_{\text{opt}}$ *output-operator expression output-timestamp*$_{\text{opt}}$
output-operator	::=	`<:`
	|	`<: >>`

output-timestamp	::=	`@` *variable-reference*
constant	::=	*integer-constant*
	|	*character-constant*
	|	*enumeration-constant*
	|	`null`