High-Level Shader Language Specification

Working Draft

July 21, 2026

Contents

Introduction [Intro]

1 The High Level Shader Language (HLSL) is the GPU programming language originally provided in conjunction with the DirectX runtime. Over many years its use has expanded to cover every major rendering API across all major development platforms.

2 HLSL draws heavy inspiration originally from ISO/IEC 9899:2011 and later from ISO/IEC 14882:2011 with additions specific to graphics and parallel computation programming. The language is also influenced to a lesser degree by other popular graphics and parallel programming languages.

3 HLSL has two reference implementations which this specification draws heavily from. The original reference implementation Legacy DirectX Shader Compiler (FXC) has been in use since DirectX 9. The more recent reference implementation DirectX Shader Compiler (DXC) has been the primary shader compiler since DirectX 12.

4 In writing this specification bias is leaned toward the language behavior of DXC rather than the behavior of FXC, although that can vary by context.

5 In very rare instances this spec will be aspirational, and may diverge from both reference implementation behaviors. This will only be done in instances where there is an intent to alter implementation behavior in the future. Since this document and the implementations are living sources, one or the other may be ahead in different regards at any point in time. The authors will see to provide editorial notes in such cases to clarify the intent of the specification and the state of the implementations.

6 This document is a *working draft*, and has not been adopted by the Ecma General Assembly. As such, this document is subject to change as the specification process continues.

1.Scope [Scope]

1 This document specifies the requirements for implementations of the High Level Shader Language (HLSL). The primary requirement is that an implementation must implement the grammar and semantics described in this document. The secondary requirement is that an implementation provide the standard library of data types described in this document.

2 HLSL was originally developed as a domain-specific language for programming shaders programs. The language has since evolved into a general-purpose programming language for GPU programming. It is based on the C and C++ programming langauges as described in the ISO/IEC 9899:2011 and ISO/IEC 14882:2011 standards, with additions alterations specific to graphics and parallel computation programming.

2.Conformance [Conformance]

1 A conforming implementation of HLSL must provide and support all the types, values, objects, attributes, functions, and program syntax and semantics described in this specification.

2 A conforming implementation of HLSL may provide additional types, values, objects, attributes, and functions beyond those described in this specification, provided that they either do not modify the base namespace and namespaces defined in this specification. Additions must not be part of the base namespace or any namespaces defined by this specification unless they are implementation details with names that use the reserved *double underscore* (__) prefix.

3 A conforming implementation of HLSL must not redefine any behavior that is not defined as implementation-defined, undefined or unspecified in this specification.

3.Normative References [Refs]

1 The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

- ISO/IEC 9899:2011, *Programming languages - C*
- ISO/IEC 14882:2011, *Programming languages - C++*
- ISO/IEC 10646:2020, *Universal coded character set*
- UAX #31, *Unicode Standard Annex, UAX #31*
- UAX #44, *Unicode Standard Annex, UAX #44*

4.Terms and definitions [Terms]

1 This document uses terms consistent with their definitions in ISO/IEC 9899:2011 and ISO/IEC 14882:2011. In cases where the definitions are unclear, or where this document diverges from ISO/IEC 9899:2011 and ISO/IEC 14882:2011, the definitions in this clause, and the attached glossary (11.4) supersede other sources.

4.1.Common Definitions [Terms.Common]

1 The following definitions are consistent between HLSL and the ISO/IEC 9899:2011 and ISO/IEC 14882:2011 specifications, however they are included here for reader convenience.

4.1.1.Correct Data [Terms.CorrectData]

1 Data is correct if it represents values that have specified or unspecified but not undefined behavior for all the operations in which it is used. Data that is the result of undefined behavior is not correct, and may be treated as undefined.

4.1.2.Diagnostic Message [Terms.Diags]

1 An implementation defined message belonging to a subset of the implementation's output messages which communicates diagnostic information to the user.

4.1.3.Ill-formed Program [Terms.IllFormed]

1 A program that is not well-formed, for which the implementation is expected to return unsuccessfully and produce one or more diagnostic messages.

4.1.4.Implementation-defined Behavior [Terms.ImpDef]

1 Behavior of a well-formed program and correct data which may vary by the implementation, and the implementation is expected to document the behavior.

4.1.5.Implementation Limits [Terms.ImpLimits]

1 Restrictions imposed upon programs by the implementation of either the compiler or runtime environment. The compiler may seek to surface runtime-imposed limits to the user for improved user experience.

4.1.6.Undefined Behavior [Terms.Undefined]

1 Behavior of invalid program constructs or incorrect data for which this standard imposes no requirements, or does not sufficiently detail.

4.1.7.Unspecified Behavior [Terms.Unspecified]

1 Behavior of a well-formed program and correct data which may vary by the implementation, and the implementation is not expected to document the behavior.

4.1.8.Well-formed Program [Terms.WellFormed]

1 An HLSL program constructed according to the syntax rules, diagnosable semantic rules, and the One Definition Rule.

4.2.General Terms [Terms.General]

1 The following terms are specific to HLSL and are used throughout the specification.

4.2.1.Runtime Implementation [Terms.Runtime]

1 A runtime implementation refers to a full-stack implementation of a software runtime that can facilitate the execution of HLSL programs. This broad definition includes libraries and device driver implementations. The HLSL specification does not distinguish between the user-facing programming interfaces and the vendor-specific backing implementation.

4.2.2.Host and Device [Terms.HostDevice]

1 HLSL is a data-parallel programming language designed for programming auxiliary processors in a larger system. In this context the *host* refers to the primary processing unit that runs the application which in turn uses a runtime to execute HLSL programs on a supported *device*. There is no strict requirement that the host and device be different physical hardware, although they commonly are. The separation of host and device in this specification is useful for defining the execution and memory model as well as specific semantics of language constructs.

4.3.SPMD Terminology [Terms.SPMD]

1 HLSL is a Single Program Multiple Data (SPMD) programming language. The following terms are used to describe the execution model of SPMD programs and the semantics of language constructs that are specific to SPMD programming.

4.3.1.Lane [Terms.Lane]

1 A Lane represents a single computed element in an SPMD program. In a traditional programming model it would be analogous to a thread of execution, however it differs in one key way. In multi-threaded programming, threads advance independent of each other. In SPMD programs, a group of Lanes may execute instructions in lockstep because each instruction may be a SIMD instruction computing the results for multiple Lanes simultaneously, or synchronizing execution across multiple Lanes or Waves. A Lane has an associated *lane state* which denotes the execution status of the lane (4.3.6).

4.3.2.Wave [Terms.Wave]

1 A grouping of Lanes for execution is called a Wave. The size of a Wave is defined as the maximum number of *active* Lanes the Wave supports. Wave sizes vary by hardware architecture, and are required to be powers of two. The

number of *active* Lanes in a Wave can be any value between one and the Wave size.

2 Some hardware implementations support multiple Wave sizes. There is no overall minimum wave size requirement, although some language features do have minimum Lane size requirements.

3 HLSL is explicitly designed to run on hardware with arbitrary Wave sizes. Hardware architectures may implement Waves as Single Instruction Multiple Thread (SIMT) where each thread executes instructions in lockstep. This is not a requirement of the model. Some constructs in HLSL require synchronized execution. Such constructs will explicitly specify that requirement.

4.3.3.Quad [Terms.Quad]

1 A Quad is a subdivision of four Lanes in a Wave which are computing adjacent values. In pixel shaders a Quad may represent four adjacent pixels and Quad operations allow passing data between adjacent Lanes. In compute shaders quads may be one or two dimensional depending on the workload dimensionality. Quad operations require four active Lanes.

4.3.4.Thread Group [Terms.Group]

1 A grouping of Lanes executing the same shader to produce a combined result is called a Thread Group. Thread Groups are independent of SIMD hardware specifications. The dimensions of a Thread Group are defined in three dimensions. The maximum extent along each dimension of a Thread Group, and the total size of a Thread Group are implementation limits defined by the runtime and enforced by the compiler. If a Thread Group's size is not a whole multiple of the hardware Wave size, the unused hardware Lanes are implicitly inactive.

2 If a Thread Group size is smaller than the Wave size , or if the Thread Group size is not an even multiple of the Wave size, the remaining Lane are *inactive* Lanes.

4.3.5.Dispatch [Terms.Dispatch]

1 A grouping of Thread Groups which represents the full execution of a HLSL program and results in a completed result for all input data elements.

4.3.6.Lane States [Terms.LaneState]

1 Lanes may be in four primary states: *active*, *helper*, *inactive*, and *predicated off*.

2 An *active* Lane is enabled to perform computations and produce output results based on the initial launch conditions and program control flow.

3 A *helper* Lane is a lane which would not be executed by the initial launch conditions except that its computations are required for adjacent pixel operations in pixel fragment shaders. A *helper* Lane will execute all computations but will not perform writes to buffers, and any outputs it produces are discarded. *Helper* lanes may be required for Lane-cooperative operations to execute correctly.

4 An *inactive* Lane is a lane that is not executed by the initial launch conditions. This can occur if there are insufficient inputs to fill all Lanes in the Wave, or to reduce per-thread memory requirements or register pressure.

5 A *predicated off* Lane is a lane that is not being executed due to program control flow. A Lane may be *predicated off* when control flow for the Lanes in a Wave diverge and one or more lanes are temporarily not executing.

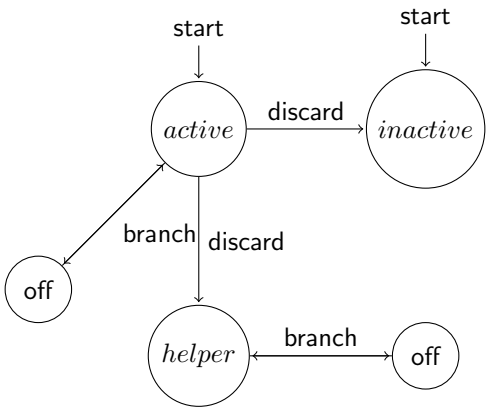
6 The diagram blow illustrates the state transitions between Lane states:

4.3.7.Uniformity [Terms.Uniformity]

1 Uniformity is a property of either a value or a control flow construct. Uniformity can be discussed in terms of any grouping of Lanes (e.g. Quad, Thread Group, Wave, Dispatch), if a grouping of Lanes is not explicitly mentioned, it is assumed to be a Wave.

2 When referring to a value, uniformity means that the value is the same across all Lanes in the grouping. When referring to a control flow construct, uniformity means that all Lanes in the grouping are executing the same path through the program's control flow.

Figure 4.1 – State transitions for Lane states



3 Uniformity can be a static property that is determined at compile time, or a dynamic property that is determined at runtime.

4.3.8.Divergence [Terms.Divergence]

1 Divergence is a dynamic property of program execution where one or more Lanes in a Wave are executing different paths through the program's control flow. Control flow that causes divergence is said to be *divergent*.

4.3.9.Convergence [Terms.Convergence]

1 Convergence is a property of an operation that requires the Lanes executing the operation to synchronize in order to produce correct results. Operations requiring convergence are said to be *convergent*.

4.4.Memory Spaces [Terms.MemorySpaces]

1 Memory spaces refer to the different types of memory that programs can access. Each memory space has different properties and semantics for how it can be accessed and used.

5.Lexical Conventions [Lex]

5.1.Unit of Translation [Lex.Translation]

1 The text of HLSL programs is collected in *source* and *header* files. The distinction between source and header files is social and not technical. An implementation will construct a *translation unit* from a single source file and any included source or header files referenced via the `#include` preprocessing directive conforming to the ISO/IEC 9899:2011 preprocessor specification.

2 An implementation may implicitly include additional sources as required to expose the HLSL library functionality as defined in (**??**).

5.2.Phases of Translation [Lex.Phases]

1 HLSL inherits the phases of translation from ISO/IEC 14882:2011, with minor alterations, specifically the removal of support for trigraph and digraph sequences. Below is a description of the phases.

1. Source files are characters that are mapped to the basic source character set in an implementation-defined manner.
2. Any sequence of backslash (\) immediately followed by a new line is deleted, resulting in splicing lines together.
3. Tokenization occurs and comments are isolated. If a source file ends in a partial comment or preprocessor token the program is ill-formed and a diagnostic shall be issued. Each comment block shall be treated as a single white-space character.
4. Preprocessing directives are executed, macros are expanded, `pragma` and other unary operator expressions are executed. Processing of `#include` directives results in all preceding steps being executed on the resolved file, and can continue recursively. Finally all preprocessing directives are removed from the source.

5. Character and string literal specifiers are converted into the appropriate character set for the execution environment.
6. Adjacent string literal tokens are concatenated.
7. White-space is no longer significant. Syntactic and semantic analysis occurs translating the whole translation unit into an implementation-defined representation.
8. The translation unit is processed to determine required instantiations, the definitions of the required instantiations are located, and the translation and instantiation units are merged. The program is ill-formed if any required instantiation cannot be located or fails during instantiation.
9. External references are resolved, library references linked, and all translation output is collected into a single output.

5.3.Character Sets [Lex.CharSet]

1 The *basic source character set* is a subset of the ASCII character set. The table below lists the valid characters and their ASCII values:

Hex ASCII Value	Character Name	Glyph or C Escape Sequence
0x09	Horizontal Tab	\t
0x0A	Line Feed	\n
0x0D	Carriage Return	\r
0x20	Space	
0x21	Exclamation Mark	!
0x22	Quotation Mark	"
0x23	Number Sign	#
0x25	Percent Sign	%
0x26	Ampersand	&
0x27	Apostrophe	’
0x28	Left Parenthesis	(
0x29	Right Parenthesis	)
0x2A	Asterisk	*
0x2B	Plus Sign	+
0x2C	Comma	,
0x2D	Hyphen-Minus	-
0x2E	Full Stop	.
0x2F	Solidus	/
0x30 .. 0x39	Digit Zero .. Nine	0 1 2 3 4 5 6 7 8 9
0x3A	Colon	:
0x3B	Semicolon	;
0x3C	Less-than Sign	<
0x3D	Equals Sign	=
0x3E	Greater-than Sign	>
0x3F	Question Mark	?
0x41 .. 0x5A	Latin Capital Letter A .. Z	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0x5B	Left Square Bracket	[
0x5C	Reverse Solidus	\
0x5D	Right Square Bracket	[
0x5E	Circumflex Accent	^
0x5F	Underscore	_
0x61 .. 0x7A	Latin Small Letter a .. z	a b c d e f g h i j k l m n o p q r s t u v w x y z
0x7B	Left Curly Bracket	{
0x7C	Vertical Line	|
0x7D	Right Curly Bracket	}

2 An implementation may allow source files to be written in alternate *extended character sets* as long as that set is a superset of the *basic character set*. The *translation character set* is an *extended character set* or the *basic character set* as chosen by the implementation.

5.4. Preprocessing Tokens [Lex.PPTokens]

preprocessing-token:
- *header-name*
- *identifier*
- *pp-number*
- *character-literal*
- *string-literal*
- *preprocessing-op-or-punc*
- each non-whitespace character from the *translation character set* that cannot be one of the above

1 Each preprocessing token that is converted to a token shall have the lexical form of a keyword, an identifier, a constant, a string literal or an operator or punctuator.

2 Preprocessing tokens are the minimal lexical elements of the language during translation phases 3 through 6 (5.2). Preprocessing tokens can be separated by whitespace in the form of comments, white space characters, or both. White space may appear within a preprocessing token only as part of a header name or between the quotation characters in a character constant or string literal.

3 Header name preprocessing tokens are only recognized within `#include` preprocessing directives, `__has_include` expressions, and implementation-defined locations within `#pragma` directives. In those contexts, a sequence of characters that could be either a header name or a string literal is recognized as a header name.

5.5. Tokens [Lex.Tokens]

token:
- *identifier*
- *keyword*
- *literal*
- *operator-or-punctuator*

1 There are five kinds of tokens: identifiers, keywords, literals, and operators or punctuators. All whitespace characters and comments are ignored except as they separate tokens.

5.6. Comments [Lex.Comments]

1 The characters `/*` start a comment which terminates with the characters `*/`. The characters `//` start a comment which terminates at the next new line.

5.7. Header Names [Lex.Headers]

header-name:
- < *h-char-sequence* >
- " *q-char-sequence* "

h-char-sequence:
- *h-char*
- *h-char-sequence h-char*

h-char:
- any character in the *translation character set* except newline or >

q-char-sequence:
- *q-char*
- *q-char-sequence q-char*

q-char:
- any character in the *translation character set* except newline or "

1 Character sequences in header names are mapped to header files or external source file names in an implementation defined way.

5.8. Preprocessing numbers [Lex.PPNumber]

pp-number:
- *digit*
- . *digit*
- *pp-number* ' *digit*
- *pp-number* ' *non-digit*
- *pp-number* `e` *sign*
- *pp-number* `E` *sign*
- *pp-number* `p` *sign*
- *pp-number* `P` *sign*
- *pp-number* .

1 Preprocessing numbers begin with a digit or period (`.`), and may be followed by valid identifier characters and floating point literal suffixes (`e+`, `e-`, `E+`, `E-`, `p+`, `p-`, `P+`, and `P-`). Preprocessing number tokens lexically include all *integer-literal* and *floating-literal* tokens.

2 Preprocessing numbers do not have types or values. Types and values are assigned to *integer-literal*, *floating-literal*, and *vector-literal* tokens on successful conversion from preprocessing numbers.

3 A preprocessing number cannot end in a period (`.`) if the immediate next token is a *scalar-element-sequence* (**??**). In this situation the *pp-number* token is truncated to end before the period.

5.9. Identifiers [Lex.Ident]

identifier:
- *identifier-start*
- *identifier identifier-continue*

identifier-start:
- *nondigit*
- an element of the translation character set of class XID_Start

identifier-continue:
- *digit*
- *nondigit*
- an element of the translation character set of class XID_Continue

nondigit: one of

```
a b c d e f g h i j k l m
n o p q r s t u v w z y z
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z _
```

digit: one of

```
0 1 2 3 4 5 6 7 8 9
```

1 The XID_Start and XID_Continue properties are Derived core properties defined in UAX #44. The characters which have the property are defined in UAX #31.

2 An XID_Start character is a character of the translation character set whose corresponding code point in ISO/IEC 10646:2020 has the XID_Start property. An XID_Continue character is a character of the translation character set whose corresponding code point in ISO/IEC 10646:2020 has the XID_Continue property. An identifier shall conform to Normalization Form C as specified in ISO/IEC 10646:2020.

3 Some identifiers are reserved for use by implementations of this standard, but are not required to emit a diagnostic:

- Any identifier that contains a double underscore (__), or begins with an underscore (_) followed by a capital letter is reserved for any use by an implementation.
- Any identifier that begins with an underscore (_) is reserved for any implementation to use as a name in the global namespace.

5.10.Keywords [Lex.Keywords]

keyword: any of

```
auto bool break case cbuffer center centroid class column_major
const constexpr continue discard do double else enum export
false float for globallycoherent groupshared if in indices
inline inout int interface line lineadj linear namespace nointerpolation
noperspective operator out packoffset payload point precise
primitives reordercoherent return row_major sampler_state shared sizeof
snorm static struct switch tbuffer template this triangle
triangleadj true typedef typename uniform unorm unsigned using vertices
void while
```

1 The identifiers defined in the grammar above are reserved as keywords. During the phases of translation when preprocessing tokens are converted to tokens, keywords are unconditionally treated as keywords except when they appear in attribute specifiers (**??**).

5.11.Operators and Punctuators [Lex.Operators]

preprocessing-op-or-punc:
preprocessing-operator
operator-or-punctuator

preprocessing-operator:
`#`
`##`

operator-or-punctuator: one of

```
{ } [ ] ( ) ; :  ...
?  ::  .
!  + - * / % & |
= += -= *= /= %= ^= &= |=
== += < > <= >= && ||
<< >> <<= >>= ++ -- ,
```

1 HLSL inherits a set of operators and punctuators from ISO/IEC 14882:2011. The set of tokens in the *preprocessing-op-or-punc* grammar formation are either interpreted by the preprocessor (**??**), or they are converted to tokens.

2 Each *operator-or-punctuator* that is not handled completely by the preprocessor is converted to a single token.

5.12.Literals [Lex.Literals]

literal:
integer-literal
character-literal
floating-literal
string-literal
boolean-literal
vector-literal

5.12.1.Integer Literals [Lex.Literals.Int]

integer-literal:
decimal-literal *integer-suffix*$_{opt}$
octal-literal *integer-suffix*$_{opt}$
hexadecimal-literal *integer-suffix*$_{opt}$

decimal-literal:
nonzero-digit
decimal-literal *digit*

octal-literal: `0`
octal-literal *octal-digit*

hexadecimal-literal:
`0x` *hexadecimal-digit*

`0X` *hexadecimal-digit*
hexadecimal-literal hexadecimal-digit

nonzero-digit: one of
`1 2 3 4 5 6 7 8 9`

octal-digit: one of
`0 1 2 3 4 5 6 7`

hexadecimal-digit: one of
`0 1 2 3 4 5 6 7 8 9`
`a b c d e f`
`A B C D E F`

integer-suffix:
unsigned-suffix long-suffix$_{opt}$
long-suffix unsigned-suffix$_{opt}$

unsigned-suffix: one of
`u U`

long-suffix: one of
`l L`

1 An *integer-literal* is a *decimal-literal*, an *octal-literal* or a *hexidecimal-literal*, and an optional type suffix. An integer literal shall not contain a period or exponent specifier.

2 The type of an integer literal is the first of the corresponding list in the table below in which its value can be represented.

Suffix	Decimal constant	Octal or hexadecimal constant

none	`int32_t`
	`int64_t`
		`int32_t`
		`uint32_t`
		`int64_t`
		`uint64_t`

`u` or `U`	`uint32_t` `uint64_t`	`uint32_t` `uint64_t`
`l` or `L`	`int64_t`	`int64_t` `uint64_t`
Both `u` or `U` and `l` or `L`	`uint64_t`	`uint64_t`

3 If the specified value of an integer literal cannot be represented by any type in the corresponding list, the integer literal has no type and the program is ill-formed.

4 An implementation may support the integer suffixes `ll` and `ull` as equivalent to `l` and `ul` respectively.

5.12.2.Floating-point Literals [Lex.Literal.Float]

floating-literal:
fractional-constant exponent-part$_{opt}$ *floating-suffix*$_{opt}$
digit-sequence exponent-part floating-suffx$_{opt}$

fractional-constant:
digit-sequence$_{opt}$ `.` *digit-sequence*
digit-sequence `.`

exponent-part:
`e` *sign*$_{opt}$ *digit-sequence*
`E` *sign*$_{opt}$ *digit-sequence*

sign: one of
`+ -`

digit-sequence:
digit
digit-sequence digit

floating-suffix: one of
`h f l`
`H F L`
`f16 f32 f64`
`F16 F32 F64`

1 A floating literal is written either as a *fractional-constant* with an optional *exponent-part* and optional *floating-suffix*, or as an integer *digit-sequence* with a required *exponent-part* and optional *floating-suffix*.

2 The type of a floating literal is `float`, unless explicitly specified by a suffix. The suffixes `h` and `H` specify `half`, the suffixes `f` and `F` specify `float`, and the suffixes `l` and `L` specify `double`. The explicitly sized suffixes `f16`, `F16`, `f32`, `F32`, `f64`, and `F64` specify the types of the bit-width specified by the number in the suffix. The `f16` and `F16` suffixes will only be supported when an implementation supports native 16-bit types. If a value specified in the source is not in the range of representable values for its type, the program is ill-formed.

6.Basic Concepts [Basic]

NOTE	HLSL inherits a significant portion of its language semantics from C and C++. Some of this is a result of intentional adoption of syntax early in the development of the language and some a side-effect of the Clang-based implementation of DXC. For clarity, this specification includes all necessary definitions even if they are not different from the ISO/IEC 9899:2011 or ISO/IEC 14882:2011 standard.

6.1.Preamble [Basic.Preamble]

1 An *entity* is a value, object, function, enumerator, type, class member, bit-field, template, template specialization, namespace, or pack.

2 A *name* is a use of an *identifier* (8.2.4), *operator-function-id* (**??**), *conversion-function-id* (**??**), or *template-id* (**??**) that denotes any entity or *label* (**??**).

3 Every name that denotes an entity is introduced by a *declaration*. Every name that denotes a label is introduced by a *labeled statement* (**??**).

4 A *variable* is introduced by the declaration of a reference other than a non-static data member of an object. The variable's name denotes the reference or object.

5 *Name lookup* is the process of determining if a name refers to a type or template, and if so, which type or template it refers to. Name lookup occurs any time a name is encountered during parsing of a translation unit.

6 Two names are the same name if:

- they are identifiers composed of the same character sequence, or
- they are operator-function-ids formed with the same operator, or
- they are conversion-function-ids formed with the same type, or
- they are template-ids that refer to the same class or function.

6.2.Declarations and definitions [Basic.Decl]

1 A declaration (9) may introduce one or more names into a translation unit or redeclare names introduced by previous declarations. If a declaration introduces names, it specifies the interpretation and attributes of these names. A declaration may also have effects such as:

- verifying a static assertion (9),
- use of attributes (9), and

- controlling template instantiation (**??**).

2 A declaration is a *definition* unless:

- it declares a function without specifying the function's body (**??**),
- it is a parameter declaration in a function declaration that does not specify the function's body (**??**),
- it is a global or namespace member declaration without the `static` specifier,
- it declares a static data member in a class definition,
- it is a class name declaration,
- it is a template parameter,
- it is a `typedef` declaration (9),
- it is an *alias-declaration* (9),
- it is a *using-declaration* (9),
- it is a *static_assert-declaration* (9),
- it is an *empty-declaration* (9),
- or a *using-directive* (9).

NOTE	Global variable declarations are implicitly constant and external in HLSL, unless they are declared with the `static` storage specifier.

3 The two examples below are adapted from ISO/IEC 14882:2011 **[basic.def]**. All but one of the following are definitions:

```
int f(int x) { return x+1; } // defines f and x
struct S {int a;int b;};     // defines S, S::a, and S::b
struct X {                   // defines X
  int x;                     // defines non-static member x
  static int y;              // declares static data member y
};
int X::y = 1;                // defines X::y
enum { up, down };           // defines up and down
namespace N {                // defines N
int d;                       // declares N::d
static int i;                // defines N::i
}
```

4 All of the following are declarations:

```
int a;                       // declares a
const int c;                 // declares c
X anX;                       // declares anX
int f(int);                  // declares f
struct S;                    // declares S
typedef int Int;             // declares Int
using N::d;                  // declares d
using Float = float;         // declares Float
cbuffer CB {                 // does not declare CB
  int z;                     // declares z
}
tbuffer TB {                 // does not declare TB
  int w;                     // declares w
}
```

6.3.One-Definition Rule [Basic.ODR]

1 The ISO/IEC 14882:2011 *One-definition rule* is adopted as defined in ISO/IEC 14882:2011 **[basic.def.odr]**.

6.4.Scope [Basic.Scope]

1 A *valid* name is a name that may be used unqualified to refer to a specific entity.

2 The *declarative region* of a declaration is the largest part of a program in which the name is valid.

3 The *scope* of a name is the possibly discontiguous region of a program in which the name may be used as an unqualified name to refer to the same entity. The scope of a declaration is the same as its declarative region unless another declaration of the same name exists within that region, in which case the declarative region of the inner declaration is excluded from the outer declaration's scope.

4 Names declared by a declaration are introduced into the scope in which the declaration occurs except for members of *cbuffer-declarations* (**??**), and *using-declarations* (**??**) which do not follow this general behavior.

5 Within a declarative region, if multiple declarations introduce the same name either;

- all the declarations shall refer to the same entity; or
- all refer to functions or function templates; or
- one declaration shall declare a class or enumeration name; and
  - the other declarations shall refer to the same entity; or
  - all refer to functions or function templates, hiding the class or enumeration name.

6 Scopes nest within each other. A *containing scope*, is a scope that fully contains another scope which is said to be *contained*.

6.5.Name Lookup [Basic.Lookup]

6.6.Storage Duration [Basic.Storage]

1 The storage duration of an object is the portion of the program's execution time during which the object exists in memory. An object may have one of the following storage durations:

- *static storage duration*
- *automatic storage duration*
- *program storage duration*
- *groupshared storage duration*

6.6.1.Static Storage Duration [Basic.Storage.Static]

1 An object whose name is declared with the `static` storage specifier has *static storage duration*. Such an object is created when the thread begins execution and destroyed when the thread ends execution.

6.6.2.Automatic Storage Duration [Basic.Storage.Auto]

1 An object whose name is declared in a block (including function parameters) without the `static` storage specifier has *automatic storage duration*. Such an object is created when the block in which it is declared is entered and destroyed when the block is exited.

6.6.3.Program Storage Duration [Basic.Storage.Program]

1 An object whose name is declared in a global, namespace, or cbuffer scope without the `static` storage specifier has *program storage duration*. Such an object is created when the program begins execution and destroyed when the program ends execution.

6.6.4.Groupshared Storage Duration [Basic.Storage.Groupshared]

1 An object whose name is declared with the `groupshared` storage specifier has *groupshared storage duration*. Such an object is created when the thread group begins execution and destroyed when the thread group ends.

6.7.Program and linkage [Basic.Linkage]

1 A translation unit (5.1) is comprised of a sequence of declarations:

translation-unit:
declaration-sequence$_{opt}$

2 A *program* is one or more translation units *linked* together. A program built from a single translation unit, bypassing a linking step is called *freestanding*.

3 A program is said to be *fully linked*, when it contains no *unresolved external* declarations, and all *exported* declarations are entry point declarations (6.8). A program is said to be *partially linked*, when it contains at least one unresolved external declaration or at least one exported declaration that is not an entry point.

4 An implementation may generate programs as fully linked or partially linked as requested by the user, and a runtime may allow fully linked or partially linked programs as the implementation allows.

5 A name has *linkage* if it can refer to the same entity as a name introduced by a declaration in another scope. If a variable, function, or another entity with the same name is declared in several scopes, but does not have sufficient *linkage*, then several instances of the entity are generated.

- A name with *no linkage* may not be referred to by names from any other scope.
- A name with *internal linkage* may be referred to by names from other scopes within the same translation unit.
- A name with *external linkage* may be referred to by names from other scopes within the same translation unit, and by names from scopes of other translation units.
- A name with *program linkage* may be referred to by names from other scopes within the same translation unit, by names from scopes of other translation units, by names from scopes of other programs, and by a runtime implementation.

6 When merging translation units through linking or generating a freestanding program only names with program linkage must be retained in the final program.

6.7.1.Program Linkage [Basic.Linkage.Program]

1 Entities with *program linkage* can be referred to from other partially linked programs or a runtime implementation.

2 The following entities have program linkage:

- entry point functions (6.8)
- functions marked with `export` keyword (**??**)
- declarations contained within an *export-declaration-group* (**??**)

6.7.2.External Linkage [Basic.Linkage.External]

1 Entities with *external linkage* can be referred to from the scopes in the other translation units and enable linking between them.

2 The following entities in HLSL have *external linkage*:

- global variables that are not marked `static` or `groupshared`
- static data members of classes or template classes

3 Linkage of functions (including template functions) that are not entry points or marked with `export` keyword is implementation dependent.

6.7.3.Internal Linkage [Basic.Linkage.Internal]

1 Entities with *internal linkage* can be referred to from all scopes in the current translation unit.

2 The following entities in HLSL have *internal linkage*:

- global variables marked as `static` or `groupshared`
- all entities declared in an unnamed namespace or a namespace within an unnamed namespace
- enumerations
- classes or template classes, their member functions, and nested classes and enumerations

6.7.4.No Linkage [Basic.Linkage.NoLinkage]

1 An entity with *no linkage* can be referred to only from the scope it is in.

2 Any of the following entites declared at function scope or block scopes derived from function scope have no linkage:

- local variables
- local classes and their member functions
- other entities declared at function scope or block scopes derived from function scope that such as typedefs, enumerations, and enumerators

6.8.Start [Basic.Start]

1 A fully linked program shall contain one or more global functions, which are the designated starting points for the program. These global functions are called *entry points*, because they denote the location where execution inside the program begins.

2 Entry point functions have different requirements based on the target runtime and execution mode (6.8.1).

3 Parameters to entry functions and entry function return types must be of scalar, vector, matrix, non-intangible class type (6.9), or array of such types. Scalar and vector parameters and return types must be annotated with semantic annotations (**??**). Class type input and output parameters must have all fields annotated with semantic annotations.

6.8.1.Execution Mode [Basic.Start.Mode]

1 A runtime may define a set of execution modes in an implementation-defined way. Each execution mode will have a set of implementation-defined rules which restrict available language functionality as appropriate for the execution mode.

6.9.Types [Basic.Types]

1 The *object representation* of an object of type `T` is the sequence of *N* bytes taken up by the object of type `T`, where *N* equals `sizeof(T)`. The *object representation* of an object may be different based on the *memory space* it is stored in (**??**).

NOTE	`sizeof(T)` returns the size of the object as if it's stored in device memory, and determining the size of an object stored in another memory space is not currently possible.

2 The *value representation* of an object is the set of bits that hold the value of type `T`. Bits in the object representation that are not part of the value representation are *padding bits*.

3 An *object type* is a type that is not a function type, not a reference type, and not a void type.

4 A *class type* is a data type declared with either the `class` or `struct` keywords (**??**). A class type `T` may be declared as incomplete at one point in a translation unit via a *forward declaration*, and complete later with a full definition. The type `T` is the same type throughout the translation unit.

5 There are special implementation-defined types such as *handle types*, which fall into a category of *standard intangible types*. Intangible types are types that have no defined object representation or value representation, as such the size is unknown at compile time. Usage restrictions for objects of intangible type are documented in 6.9.3.

6 A class type `T` is an *intangible class type* if it contains a base class or members of intangible class type, standard intangible type, or arrays of such types. Standard intangible types and intangible class types are collectively called *intangible types*(**??**).

7 An object type is an *incomplete type* if the compiler lacks sufficient information to determine the size of an object of type `T`, and it is not an intangible type. It is a *complete type* if the compiler has sufficient information to determine the size of an object of type `T`, or if the type is known to be an intangible type. An object may not be defined to have an *incomplete* type.

8 Arithmetic types (6.9.1), enumeration types, and *cv-qualified* versions of these types are collectively called *scalar types*.

9 Vectors of scalar types declared with the built-in `vector<T,N>` template are *vector types*. An implementation must support vector lengths between 1 and 4 (i.e. $1 \leq N \leq 4$).

NOTE An implementaiton may support vector lengths greater than 4.

10 Matrices of scalar types declared with the built-in `matrix<T,N,M>` template are *matrix types*. Matrix dimensions, `N` and `M`, must be between 1 and 4 (i.e. $1 \leq N \leq 4$).

6.9.1.Arithmetic Types [Basic.Types.Arithmetic]

1 There are three *standard signed integer types*: `int16_t`, `int32_t`, and `int64_t`. Each of the signed integer types is explicitly named for the size in bits of the type's object representation. There is also the type alias `int` which is an alias of `int32_t`. There is one *minimum precision signed integer type*: `min16int`. The minimum precision signed integer type is named for the required minimum value representation size in bits. The object representation of `min16int` is `int`. The standard signed integer types and minimum precision signed integer type are collectively called *signed integer types*.

2 There are three *standard unsigned integer types*: `uint16_t`, `uint32_t`, and `uint64_t`. Each of the unsigned integer types is explicitly named for the size in bits of the type's object representation. There is also the type alias `uint` which is an alias of `uint32_t`. There is one *minimum precision unsigned integer type*: `min16uint`. The minimum precision unsigned integer type is named for the required minimum value representation size in bits. The object representation of `min16uint` is `uint`. The standard unsigned integer types and minimum precision unsigned integer type are collectively called *unsigned integer types*.

3 The minimum precision signed integer types and minimum precision unsigned integer types are collectively called *minimum precision integer types*. The standard signed integer types and standard unsigned integer types are collectively called *standard integer types*. The signed integer types and unsigned integer types are collectively called *integer types*. Integer types inherit the object representation of integers defined in Standard for Programming Language C., which requires twos' complement representation. Integer types shall satisfy the constraints defined in Standard for Programming Language C++., section **basic.fundamental**.

4 There are three *standard floating point types*: `half`, `float`, and `double`. The `float` type is a 32-bit floating point type, and has a type alias `float32_t`. The `double` type is a 64-bit floating point type, and has a type alias `float64_t`. Both the `float` and `double` types have object representations as defined in ISO/IEC 60559:2020. The `half` type may be either 16-bit or 32-bit as controlled by implementation-defined compiler settings. If `half` is 32-bit it will have an object representation as defined in ISO/IEC 60559:2020, otherwise it will have an object representation matching the **binary16** format defined in ISO/IEC 60559:2020. When `half` is 16-bit, it will have a type alias `float16_t`. There is one *minimum precision floating point type*: `min16float`. The minimum precision floating point type is named for the required minimum value representation size in bits. The object representation of `min16float` is `float`. The standard floating point types and minimum precision floating point type are collectively called *floating point types*.

5 Integer and floating point types are collectively called *arithmetic types*.

6 The `void` type is inherited from ISO/IEC 14882:2011, which defines it as having an empty set of values and being an incomplete type that can never be completed. The `void` type is used to signify the return type of a function that returns no value. Any expression can be explicitly converted to `void`.

6.9.2.Scalarized Type Compatability [Basic.Types.Scalarized]

1 All types `T` have a *scalarized representation*, $SR(T)$, which is a list of one or more types representing each scalar element of `T`.

2 Scalarized representations are determined as follows:

- The scalarized representation of an array `T[n]` is $SR(T_0), ..SR(T_{n-1})$.
- The scalarized representation of a vector `vector<T,n>` is $T_0, ..T_{n-1}$.
- The scalarized representation of a matrix `matrix<T,n, m>` is $T_0, ..T_{(n \times m)-1}$.
- The scalarized representation of a class type `T`, $SR(T)$ is computed recursively as $SR(T :: base), SR(T ::_0), ..SR(T ::_n)$ where (T::base) is `T`'s base class if it has one, and $T ::_n$ represents the *n* non-static members of `T`.
- The scalarized representation for an enumeration type is the underlying arithmetic type.
- The scalarized representation for arithmetic, intangible types, and any other type `T` is T.

3 Two types *cv1* `T1` and *cv2* `T2` are *scalar-layout-compatible types* if `T1` and `T2` are the same type or if the sequence of types defined by the scalar representation $SR(T1)$ and scalar representation $SR(T2)$ are identical.

6.9.3.Usage of Intangible Types [Basic.Types.Intangible]

1 The following usage restrictions apply to intangible types:

- Instances of objects of intangible type may only be declared in the Thread address space (**??**).
- An object of intangible type may not be loaded or stored to any address space other than the Thread address space (**??**).
- An object of intangible type may not be a parameter or return type of a function with program linkage or external linkage (6.7.1 and 6.7.2).

6.10.Lvalues and rvalues [Basic.lval]

1 Expressions are classified by the type(s) of values they produce. The valid types of values produced by expressions are:

1. An *lvalue* represents a function or object.
2. An *rvalue* represents a temporary object.
3. An *xvalue* (expiring value) represents an object near the end of its lifetime.
4. A *cxvalue* (casted expiring value) is an *xvalue* which, on expiration, assigns its value to a bound *lvalue*.
5. A *glvalue* is an *lvalue*, *xvalue*, or *cxvalue*.
6. A *prvalue* is an *rvalue* that is not an *xvalue*.

7.Standard Conversions [Conv]

1 HLSL inherits standard conversions similar to ISO/IEC 14882:2011. This chapter enumerates the full set of conversions. A *standard conversion sequence* is a sequence of standard conversions in the following order:

1. Zero or one conversion of either lvalue-to-rvalue, or array-to-pointer.
2. Zero or one conversion of either integral conversion, floating point conversion, floating point-integral conversion, boolean conversion, derived-to-base-lvalue, or flat conversion.
3. Zero or one conversion of scalar-vector splat, or vector/matrix truncation.

4. Zero or one qualification conversion.

NOTE	This differs from C++ with the addition of flat conversion, and the dimension altering conversions for scalar, vector and matrix types.

Standard conversion sequences are applied to expressions, if necessary, to convert it to a required destination type.

7.1.Lvalue-to-rvalue conversion [Conv.LVal]

1 A glvalue of a non-function type `T` can be converted to a prvalue. The program is ill-formed if `T` is an incomplete type. If the glvalue refers to an object that is not of type `T` and is not an object of a type derived from `T`, the program is ill-formed. If the glvalue refers to an object that is uninitialized, the behavior is undefined. Otherwise the prvalue is of type `T`.

2 If the glvalue refers to an array of type `T`, the prvalue will refer to a copy of the array, not memory referred to by the glvalue.

3 If the glvalue refers to a bit-field of type `T`, the conversion produces an object of the underlying type of the bit-field.

7.2.Array-to-pointer conversion [Conv.Array]

1 An lvalue or rvalue of type `T[]` (unsized array), can be converted to a prvalue of type pointer to `T`.

NOTE	HLSL does not support grammar for specifying pointer or reference types, however they are used in the type system and must be described in language rules.

NOTE	Array-to-pointer conversion of constant sized arrays is not supported.

7.3.Integral promotion [Conv.IPromote]

1 An integral promotion is a conversion of:

- a glvalue of integer type other than `bool` to a cxvalue of integer type of higher conversion rank, or
- a conversion of a prvalue of integer type other than bool to a prvalue of integer type of higher conversion rank, or
- a conversion of a glvalue of type `bool` to a cxvalue of integer type, or
- a conversion of a prvalue of type `bool` to a prvalue of integer type.

2 Integer conversion ranks are defined in section 7.17.1.

3 A conversion is only a promotion if the destination type can represent all of the values of the source type.

7.4.Floating point promotion [Conv.FPPromote]

1 A glvalue of a floating point type can be converted to a cxvalue of a floating point type of higher conversion rank, or a prvalue of a floating point type can be converted to a prvalue of a floating point type of higher conversion rank.

2 Floating point conversion ranks are defined in section 7.17.2.

7.5.Integral conversion [Conv.IConv]

1 A glvalue of an integer type can be converted to a cxvalue of any other non-enumeration integer type. A prvalue of an integer type can be converted to a prvalue of any other integer type.

2 If the destination type is unsigned, integer conversion maintains the bit pattern of the source value in the destination type truncating or extending the value to the destination type.

3 If the destination type is signed, the value is unchanged if the destination type can represent the source value. If the destination type cannot represent the source value, the result is implementation-defined.

4 If the source type is `bool`, the values `true` and `false` are converted to one and zero respectively.

7.6.Bit-field conversion [Conv.Bitfield]

1 A glvalue of a named bit-field with underlying integer type `T` can be converted to a cxvalue of type `T′` if there exists an implicit conversion from `T` to `T′`.

7.7.Floating point conversion [Conv.FConv]

1 A glvalue of a floating point type can be converted to a cxvalue of any other floating point type. A prvalue of a floating point type can be converted to a prvalue of any other floating point type.

2 If the source value can be exactly represented in the destination type, the conversion produces the exact representation of the source value. If the source value cannot be exactly represented, the conversion to a best-approximation of the source value is implementation defined.

7.8.Floating point-integral conversion [Conv.FPInt]

1 A glvalue of floating point type can be converted to a cxvalue of integer type. A prvalue of floating point type can be converted to a prvalue of integer type. Conversion of floating point values to integer values truncates by discarding the fractional value. The behavior is undefined if the truncated value cannot be represented in the destination type.

2 A glvalue of integer type can be converted to a cxvalue of floating point type. A prvalue of integer type can be converted to a prvalue of floating point type. If the destination type can exactly represent the source value, the result is the exact value. If the destination type cannot exactly represent the source value, the conversion to a best-approximation of the source value is implementation defined.

7.9.Boolean conversion [Conv.Bool]

1 A glvalue of arithmetic type can be converted to a cxvalue of boolean type. A prvalue of arithmetic or unscoped enumeration type can be converted to a prvalue of boolean type. A zero value is converted to `false`; all other values are converted to `true`.

7.10.Elementwise conversion [Conv.Flat]

1 For a given aggregate `A`, there is a flattened order of subojects, `A′`, as described in section 9.4.1. A prvalue of `A`, can be elementwise converted to a prvalue of an aggregate `B`, whose flattened form is `B′`, if the following conditions are true:

- The length of `A′`, `N`, is greater than or equal to the length of `B′`, `M`.
- For every index `I` from `0` to `M-1`, there exists an implicit conversion from `A′[I]` to `B′[I]`.

2 A glvalue of type `T[N]` can be converted to a cxvalue of type `T`.

3 A glvalue of type `T[N]` can be converted to a cxvalue of type `T[M]`, if `N` is greater than `M`.

4 A glvalue of type `T[N]` can be converted to a cxvalue of type `vector<T,M>`, if `N` is greater than or equal to `M`.

5 A glvalue of type `vector<T,N>` can be converted to a cxvalue of type `T[M]`, if `N` is greater than or equal to `M`.

7.11.Vector splat conversion [Conv.vsplat]

1 A glvalue of type `T` can be converted to a cxvalue of type `vector<T,N>` or a prvalue of type `T` can be converted to a prvalue of type `vector<T,N>`. The destination value is the source value replicated into each element of the destination.

2 A prvalue of type `vector<T,1>` can be converted to a prvalue of type `vector<T,N>`. The destination value is the value in the source vector replicated into each element of the destination.

7.12.Matrix splat conversion [Conv.msplat]

1 A glvalue of type `T` can be converted to a cxvalue of type `matrix<T,M,N>` or a prvalue of type `T` can be converted to a prvalue of type `matrix<T,M,N>`. The destination value is the source value replicated into each element of the destination.

7.13.Array and Class splat conversion [Conv.ASplat]

1 A prvalue of type `T` can be converted to a prvalue of type `struct S`, if there are valid conversions from `T` to each `U` in flattened `S`, see section 9.4.1 for description of a flattened ordering. The destination value is the source value `T` replicated into each element of the flattened `S`.

2 A prvalue of type `vector<T,1>` can be converted to a prvalue of type `struct S`, if there are valid conversions from `T` to each `U` in flattened `S`. The destination value is the value in the source vector replicated into each element of the flattened `S`.

3 A prvalue of type `T` can be converted to a prvalue of type `U[N]`, if there are valid conversions from `T` to each `V` in flattened `U`. The destination value is the source value `T` replicated into each element of the flattened `U[N]`.

4 A prvalue of type `vector<T,1>` can be converted to a prvalue of type `U[N]`, if there are valid conversions from `T` to each `V` in flattened `U`. The destination value is the value in the source vector replicated into each element of the flattened `U[N]`.

7.14.Vector and matrix truncation conversion [Conv.VTrunc]

1 A prvalue of type `vector<T,N>` can be converted to a prvalue of type:

- `vector<T,N'>` only if $N\prime < N$, or
- `T`

2 The resulting value of vector truncation is comprised of elements $[0..N\prime)$, dropping elements $[N\prime..N)$.

3 A prvalue of type `matrix<T,M,N>` can be converted to a prvalue of type:

- `matrix<T,M',N'>` only if $M \geq M\prime$ and $N \geq N\prime$,
- `vector<T,z>` only if $x \geq z$, or
- `T`.

4 Matrix truncation is performed on each row and column dimension separately. The resulting value is comprised of vectors $[0..M\prime)$ which are each separately comprised of elements $[0..N\prime)$. Trailing vectors and elements are dropped.

5 Reducing the dimension of a vector to one (`vector<T,1>`), can produce either a single element vector or a scalar of type `T`. Reducing the rows of a matrix to one (`matrix<T,M,1>`), can produce either a single row matrix, a vector of type `vector<T,M>`, or a scalar of type `T`.

7.15.Component-wise conversions [Conv.CWise]

1 A glvalue of type `vector<T,M>` can be converted to a cxvalue of type `vector<V,M>`, or a prvalue of type `vector<T,M>` can be converted to a prvalue of type `vector<V,M>`. The source value is converted by performing the appropriate conversion of each element of type `T` to an element of type `V` following the rules for standard conversions in chapter 7.

2 A glvalue of type `matrix<T,M,N>` can be converted to a cxvalue of type `matrix<V,M,N>`, or a prvalue of type `matrix<T,M,N>` can be converted to a prvalue of type `matrix<V,M,N>`. The source value is converted by performing the appropriate conversion of each element of type `T` to an element of type `V` following the rules for standard conversions in chapter 7.

7.16.Qualification conversion [Conv.Qual]

A prvalue of type "*cv1* `T`" can be converted to a prvalue of type "*cv2* `T`" if type "*cv2* `T`" is more cv-qualified than "*cv1* `T`".

7.17.Conversion Rank [Conv.Rank]

1 Every integer and floating point type have defined conversion ranks. These conversion ranks are used to differentiate between *promotions* and other conversions (see: 7.3 and 7.4).

7.17.1.Integer Conversion Rank [Conv.Rank.Int]

- No two signed integer types shall have the same conversion rank even if they have the same representation.
- The rank of a signed integer type shall be greater than the rank of any signed integer type with a smaller size.
- The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type.
- The rank of `bool` shall be less than the rank of all other standard integer types.
- The rank of a minimum precision integer type shall be less than the rank of any other minimum precision integer type with a larger minimum value representation size.
- The rank of a minimum precision integer type shall be less than the rank of all standard integer types.
- For all integer types `T1`, `T2`, and `T3`: if `T1` has greater rank than `T2` and `T2` has greater rank than `T3`, then `T1` shall have greater rank than `T3`.

7.17.2.Floating Point Conversion Rank [Conv.Rank.Float]

- The rank `half` shall be greater than the rank of `min16float`.
- The rank `float` shall be greater than the rank of `half`.
- The rank `double` shall be greater than the rank of `float`.
- For all floating point types `T1`, `T2`, and `T3`: if `T1` has greater rank than `T2` and `T2` has greater rank than `T3`, then `T1` shall have greater rank than `T3`.

8.Expressions [Expr]

1 This clause defines the formulations of expressions and the behavior of operators when they are not overloaded. Operator overloading does not alter the rules for operators defined by this standard.

2 An expression may also be an *unevaluated operand* when it appears in some contexts. An *unevaluated operand* is a expression which is not evaluated in the program.

NOTE	The operand to `sizeof(...)` is a good example of an *unevaluated operand*. In the code `sizeof(Foo())`, the call to `Foo()` is never evaluated in the program.

3 Whenever a *glvalue* appears in an expression that expects a *prvalue*, a standard conversion sequence is applied based on the rules in 7.

8.1.Usual Arithmetic Conversions [Expr.Conv]

1 Binary operators for arithmetic and enumeration type require that both operands are of a common type. When the types do not match, the *usual arithmetic conversions* are applied to yield a common type. When *usual arithmetic conversions* are applied to vector operands they behave as component-wise conversions (7.15). The *usual arithmetic conversions* are:

- If either operand is of scoped enumeration type, no conversion is performed, and the expression is ill-formed if the types do not match.
- If either operand is a `vector<T,X>`, vector truncation or scalar extension is performed with the following rules:
  - If both vectors are of the same length, no dimension conversion is required.
  - If one operand is a vector and the other operand is a scalar, the scalar is extended to a vector via a Splat conversion (**??**) to match the length of the vector.
  - Otherwise, if both operands are vectors of different lengths, the vector of longer length is truncated to match the length of the shorter vector (7.14).
- If either operand is of type `double` or `vector<double, X>`, the other operator shall be converted to match.

- Otherwise, if either operand is of type `float` or `vector<float, X>`, the other operand shall be converted to match.
- Otherwise, if either operand is of type `half` or `vector<half, X>`, the other operand shall be converted to match.
- Otherwise, integer promotions are performed on each scalar or vector operand following the appropriate scalar or component-wise conversion (7).
  - If both operands are scalar or vector elements of signed or unsigned integer types, the operand of lesser integer conversion rank shall be converted to the type of the operand with greater rank.
  - Otherwise, if both the operand of unsigned scalar or vector element type is of greater rank than the operand of signed scalar or vector element type, the signed operand is converted to the type of the unsigned operand.
  - Otherwise, if the operand of signed scalar or vector element type is able to represent all values of the operand of unsigned scalar or vector element type, the unsigned operand is converted to the type of the signed operand.
  - Otherwise, both operands are converted to a scalar or vector type of the unsigned integer type corresponding to the type of the operand with signed integer scalar or vector element type.

8.2.Primary Expressions [Expr.Primary]

primary-expression:
literal
`this`
`(` *expression* `)`
id-expression

8.2.1.Literals [Expr.Primary.Literal]

1 The type of a *literal* is determined based on the grammar forms specified in **??**.

8.2.2.This [Expr.Primary.This]

1 The keyword `this` names a reference to the implicit object of non-static member functions. The `this` parameter is always a *glvalue*.

2 A `this` expression shall not appear outside the declaration of a non-static member function.

8.2.3.Parenthesis [Expr.Primary.Paren]

1 An expression (*E*) enclosed in parenthesis has the same type, result and value category as *E* without the enclosing parenthesis. A parenthesized expression may be used in the same contexts with the same meaning as the same non-parenthesized expression.

8.2.4.Names [Expr.Primary.ID]

id-expression:
unqualified-id
qualified-id

NOTE	The grammar and behaviors of this section are almost identical to C/C++ with some subtractions (notably lambdas and destructors).

8.2.4.Unqualified Identifiers [Expr.Primary.ID.Unqual]

unqualified-id:
identifier
operator-function-id
conversion-function-id
template-id

8.2.4. Qualified Identifiers [Expr.Primary.ID.Qual]

qualified-id:
nested-name-specifier `template`$_{opt}$ *unqualified-id*

nested-name-specifier:
`::`
type-name `::`
namespace-name `::`
nested-name-specifier identifier `::`
nested-name-specifier `template`$_{opt}$ *simple-template-id* `::`

8.3. Postfix Expressions [Expr.Post]

postfix-expression:
primary-expression
postfix-expression `[` *expression* `]`
postfix-expression `[` *braced-init-list* `]`
postfix-expression `(` *expression-list*$_{opt}$ `)`
simple-type-specifier `(` *expression-list*$_{opt}$ `)`
typename-specifier `(` *expression*$_{opt}$ `)`
postfix-expression `.` `template`$_{opt}$ *id-expression*
postfix-expression `.` *vector-swizzle-component-sequence*
postfix-expression `.` *matrix-swizzle-component-sequence*
postfix-expression `++`
postfix-expression `--`

8.4. Subscript [Expr.Post.Subscript]

8.5. Swizzle Expressions [Expr.Post.Swizzle]

1 A *postfix-expression* followed by a dot (`.`) and a sequence of one or more *swizzle components* is a postfix expression. The postfix expression before the dot is evaluated and must be of vector or matrix type. If the postfix expression before the dot is an lvalue, the swizzle expression may produce either an lvalue or a prvalue; otherwise it produces a prvalue.

2 If the postfix expression before the dot is an lvalue and the *swizzle-component-sequence* contains no repeated components, the swizzle expression is an lvalue; otherwise it is a prvalue.

8.5.1. Vector Swizzle [Expr.Post.Swizzle.Vector]

vector-swizzle-component-sequence:
swizzle-component-sequence-rgba
swizzle-component-sequence-xyzw

swizzle-component-sequence-rgba:
swizzle-component-rgba
swizzle-component-sequence-rgba swizzle-component-rgba

swizzle-component-sequence-xyzw:
swizzle-component-xyzw
swizzle-component-sequence-xyzw swizzle-component-xyzw

swizzle-component-rgba: one of
`r g b a`

swizzle-component-xyzw: one of
`x y z w`

1 A *vector-swizzle-component-sequence* is a sequence of one or more swizzle components of either the *swizzle-component-rgba* or *swizzle-component-xyzw* forms; the two forms may not be mixed in the same *vector-swizzle-*

component-sequence. The type of a swizzle expression is a vector of the same element type as the postfix expression before the dot, with a number of elements equal to the number of components in the *vector-swizzle-component-sequence*.

2 Vector swizzle components map to elements of the vector in the following way:

Element Index	RGBA component	XYZW component
0	r	x
1	g	y
2	b	z
3	a	w

3 A program is ill-formed if any component in the *swizzle-component-sequence* refers to an element index that is out of range of the vector type of the postfix expression before the dot.

8.5.2.Matrix Swizzle [Expr.Post.Swizzle.Matrix]

matrix-swizzle-component-sequence:
matrix-zero-indexed-swizzle-sequence
matrix-one-indexed-swizzle-sequence

matrix-zero-indexed-swizzle-sequence:
matrix-zero-indexed-swizzle
matrix-zero-indexed-swizzle-sequence matrix-zero-indexed-swizzle

matrix-zero-indexed-swizzle:
`_m` *zero-index-value zero-index-value*

zero-index-value: one of
`0 1 2 3`

matrix-one-indexed-swizzle-sequence:
matrix-one-indexed-swizzle
matrix-one-indexed-swizzle-sequence matrix-one-indexed-swizzle

matrix-one-indexed-swizzle:
`_` *one-index-value one-index-value*

one-index-value: one of
`1 2 3 4`

- **._** (math-style): subsequent subscripts use **1-based** indexing for both row and column.
- **._m** (memory-style): subsequent subscripts use **0-based** indexing for both row and column.

1 A *matrix-swizzle-component-sequence* is a sequence of one but less than or equal to four swizzle components of either the *matrix-zero-indexed-swizzle* or *matrix-one-indexed-swizzle* forms; the two forms may not be mixed in the same *matrix-swizzle-component-sequence*. The type of a swizzle expression is a vector of the same element type as the postfix expression before the dot, with a number of elements equal to the number of components in the *matrix-swizzle-component-sequence*.

2 Matrix swizzle components map to elements of the matrix in the following way:

Element Index	0 indexed component	1 indexed component
0,0	_m00	_11
0,1	_m01	_12
0,2	_m02	_13
0,3	_m03	_14
1,0	_m10	_21
1,1	_m11	_22
1,2	_m12	_23
1,3	_m13	_24
2,0	_m20	_31
2,1	_m21	_32
2,2	_m22	_33
2,3	_m23	_34
3,0	_m30	_41
3,1	_m31	_42
3,2	_m32	_43
3,3	_m33	_44

8.6.Function Calls [Expr.Post.Call]

1 A function call may be an *ordinary function*, or a *member function*. In a function call to an *ordinary function*, the *postfix-expression* must be an lvalue that refers to a function. In a function call to a *member function*, the *postfix-expression* will be an implicit or explicit class member access whose *id-expression* is a member function name.

2 When a function is called, each parameter shall be initialized with its corresponding argument. The order in which parameters are initialized is unspecified.

3 If the function is a non-static member function the `this` argument shall be initialized to a reference to the object of the call as if casted by an explicit cast expression to an lvalue reference of the type that the function is declared as a member of.

4 Parameters are either *input parameters*, *output parameters*, or *input/output parameters* as denoted in the called function's declaration (**??**). For all types of parameters the argument expressions are evaluated before the function call occurs.

5 *Input parameters* are passed by-value into a function. If an argument to an *input parameter* is of constant-sized array type, the array is copied to a temporary and the temporary value is converted to an address via array-to-pointer decay. If an argument is an unsized array type, the array lvalue directly decays via array-to-pointer decay.

6 Arguments to *output* and *input/output parameters* must be lvalues. *Output parameters* are not initialized prior to the call; they are passed as an uninitialized cxvalue (**??**). An *output parameter* is only initialized explicitly inside the called function. It is undefined behavior to not explicitly initialize an *output parameter* before returning from the function in which it is defined. The cxvalue created from an argument to an *input/output parameter* is initialized through copy-initialization from the lvalue argument expression. Overload resolution shall occur on argument initialization as if the expression `T Param = Arg` were evaluated. In both cases, the cxvalue shall have the type of the parameter and the argument can be converted to that type through implicit or explicit conversion.

7 If an argument to an *output* or *input/output parameter* is a constant sized array, the array is copied to a temporary cxvalue following the same rules for any other data type. If an argument to an *output* or *input/output parameter* is an unsized array type, the array lvalue directly decays via array-to-pointer decay. An argument of a constant sized array of type `T[N]` can be converted to a cxvalue of an unsized array of type `T[]` through array to pointer decay. An unsized array of type `T[]`, cannot be implicitly converted to a a constant sized array of type `T[N]`.

8 On expiration of the cxvalue, the value is assigned back to the argument lvalue expression using a resolved assignment expression as if the expression `Arg = Param` were written. The argument expression must be of a type or able to convert to a type that has defined copy-initialization to and assignment from the parameter type. The lifetime of the cxvalue begins at argument expression evaluation, and ends after the function returns. A cxvalue argument is passed by-address to the caller.

NOTE	If the argument to an *output* or *input/output parameter* is the result of a side-effecting expression, the side-effecting expression is evaluated only once, and the result is stored in a temporary cxvalue which is passed by-address to the caller. The temporary cxvalue is then assigned back to the argument expression after.

9 When a function is called, any parameter of object type must have completely defined type, and any parameter of array of object type must have completely defined element type. The lifetime of a parameter ends on return of the function in which it is defined. Initialization and destruction of each parameter occurs within the context of the calling function.

10 The value of a function call is the value returned by the called function.

11 A function call is an lvalue if the result type is an lvalue reference type; otherwise it is a prvalue.

NOTE	Functions that return lvalue references are unspellable in HLSL, but the language requires them for some built-in functions. For example, the `operator[]` overloads for writable resource types.

12 If a function call is a prvalue of object type, the type of the prvalue must be complete.

9. Declarations [Decl]

9.1. Preamble [Decl.Pre]

1 Declarations generally specify how names are to be interpreted. Declarations have the form

declaration-seq:
declaration
declaration-seq declaration

declaration:
name-declaration
special-declaration

name-declaration:
block-declaration
function-definition
template-declaration
namespace-definition
empty-declaration
attribute-declaration
cbuffer-declaration

special-declaration:
export-declaration-group
cbuffer-member-declaration

block-declaration:
simple-declaration
namespace-alias-definition
using-declaration
using-directive
static_assert-declaration
alias-declaration
opaque-enum-declaration

empty-declaration:
;

9.2.Specifiers [Decl.Spec]

9.2.1.General [Decl.Spec.General]

1 The specifiers that can be used in a declaration are

decl-specifier:
storage-class-specifier
type-specifier
function-specifier
`typedef`

decl-specifier-seq:
decl-specifier
decl-specifier decl-specifier-seq

9.2.2.Function specifiers [Decl.Spec.Fct]

1 A *function-specifier* can be used only in a function declaration.

function-specifier:
`export`

2 The `export` specifier denotes that the function has program linkage (6.7.1).

3 The `export` specifier cannot be used on functions directly or indirectly within an unnamed namespace.

4 Functions with program linkage can also be specified in *export-declaration-group* (**??**).

5 If a function is declared with an `export` specifier then all redeclarations of the same function must also use the `export` specifier or be part of *export-declaration-group* (**??**).

9.3.Declarators [Decl.Decl]

9.4.Initializers [Decl.Init]

1 The process of initialization described in this section applies to all initializers regardless of the context.

initializer:
brace-or-equal-initializer
(*expression-list*)

brace-or-equal-initializer:
= *initializer-clause*
braced-init-list

initializer-clause:
assignment-expression
braced-init-list

braced-init-list:
{ *initializer-list* ,$_{opt}$ }
{ }

initializer-list:
initializer-clause
initializer-list , *initializer-clause*

9.4.1.Aggregate Initialization [Decl.Init.Agg]

1 An *aggregate* is a vector, matrix, array, or class.

2 The subobjects of an aggregate have a defined order. For vectors and arrays the order is increasing subscript order. For matrices it is increasing subscript order with the subscript nesting such that in the notation `Mat[M][N]`, the ordering is $Mat[0][0]...Mat[0][N]...Mat[M][0]...Mat[M][N]$. For classes the order is base class, followed by member subobjects in declaration order.

3 A *flattened ordering* of subobjects can be produced by performing a depth-first traversal of the subobjects of an object following the defined subobject ordering.

4 Each *braced initializer list* is comprised of zero or more *initializer-clause* expressions, which is either another braced initializer list or an expression which generates a value that either is or can be implicitly converted to an rvalue. Each assignment-expression is an object, which may be a scalar or aggregate type. A *flattened initializer sequence* is a sequence of expressions constructed by a depth-first traversal over each assignment-expression in an initializer-list and performing a depth-first traversal accessing each subobject of the assignment-expression.

5 If the target object is an array of unknown size, the object is assumed to have m possible elements during parsing, where $m > 0$.

6 An initializer-list is a valid initializer if for each element $E_{n \bmod m}$ in the target object's flattened ordering there is a corresponding expression E_n in the flattened initializer sequence, which can be implicitly converted to the element's type. For arrays of unknown size, the total number of expressions in the flattened initializer sequence must be a multiple of the array's base element type.

7 An initializer-list is invalid if the flattened initializer sequence contains more or fewer elements than the target object's flattened ordering, or if any initializer I_n cannot be implicitly converted to the corresponding element E_n's type.

10.Library Overview [Library]

1 This clause serves as an overview to the remaining clauses of the specification which detail HLSL's standard library of functions and objects.

10.1.Syntax Notations [Library.Notation]

1 Some of the constructs in the standard library are not expressible with spellable syntax. This section describes syntax constructs which are non-standard but are used throughout the standard library specification.

2 Code blocks in the remaining clauses of the specification decribe a hypothetical implementation and are provided for clarity. A conforming implementation need not rely on the specific code in any of the blocks.

10.1.1.Constructors [Library.Notation.Constructors]

1 Some standard library types have behaviors associated in ISO/IEC 14882:2011 with constructors. These syntaxes are intended to describe equivalent behavior with C++ constructors.

10.1.2.Reference Types [Library.Notation.References]

1 Some standard library functions or member functions behave as if a parameter or return value are references (passed by address or returned by address). To simplify this notation the syntax *address-space*$_{opt}$ `T &` is used where the optional *address-space* is one of `device` or `groupshared`.

10.1.3.Concepts [Library.Notation.Concepts]

1 Some standard library template functions or objects have constraints to which types are valid as template arguments. To simplify this notation the concepts syntax from ISO/IEC 14882:2020 is used.

10.2.Method of Description (Informative) [Library.Method]

10.2.1.Clause Structure [Libarry.Method.Structure]

1 Each library clause will contain all applicable subclauses of the list:

- Introduction
- Optional features
- Special semantics

- Element subclauses

2 The clause introduction shall provide an overview of the contents of the clause. Some features detailed in a clause may be optional, a subclause describing the optional features will be included if applicable.

3 Each library clause will optionally contain a subclause describing additional semantic behaviors required for the clause. This subclause is omitted if empty.

4 Each library clause will contain one or more element subclauses detailing the declarations required in a conforming implementation of the specification, and any normative optional declarations based on optional features described in the clause.

10.2.1.Element Subclauses [Library.Method.Structure.Element]

1 Each element subclause contain the following:

- Summary
- Detailed specifications

10.2.1.Summary [Library.Method.Structure.Summary]

1 The summary will include a description of the category as well as a listing of any introduced headers, macros, and declarations.

10.2.1.Detailed Specification [Library.Method.Structure.Detailed]

1 The detailed specification for each entity will contain each of the below that apply:

- Name and description
- Class definition or function prototype
- Template argument constraints
- Class invariants
- Function semantics

2 Description of class member functions will follow the order, omitting sections that are empty:

- Constructors
- Copying and assignment
- Modifying functions
- Inspection functions
- Operators
- Other non-member functions

3 Descriptions of function semantics will contain all of the following elements that apply in the order as specified:

- *Requires:* the preconditions for calling the function.
- *Effects:* the actions performed by the function.
- *Synchronization:* the requirements for execution or memory synchronization.
- *Returns:* a description of the value returned by the function.
- *Remarks:* additional normative information about the function.
- *Notes:* non-normative information provided about the function.

11.Resource Types [Resources]

1 Resource types are intangible types representing memory or configuration with an external interface.

2 These take the form of typed buffers, raw buffers, textures, constant buffers and samplers.

3 Typed buffer, raw buffer, and Texture resources may be read-only or read-write as denoted by the type of the resource, where the names of read-write resource types are prefixed with `RW`.

4 Global resource declarations are *bound* to device memory managed by an application through a runtime implementation. A resource declaration that is not bound is said to be *unbound*. Operations on unbound resources are evaluated as no-ops.

5 An implementation may impose restrictions on how local resource objects may be used (**??**).

11.1.Optional Features [Resources.Optional]

1 *Dynamic resources* is an optional feature which allows resource objects to be initialized by indexing directly into a heap of descriptors. A conforming implementation must either support all interfaces related to dynamic resources or none, but may not partially support them.

2 The `__DynamicResourceHandle` type in the code examples in this clause is a placeholder for the implementation-specific type returned by subscript indexing into the `ResourceDescriptorHeap` (**??**).

11.2.Special Semantics [Resources.Special]

11.2.1.Resource Initialization [Resources.Special.Initialization]

1 A resource declaration in a global scope is called a *global resource declaration* and represents a binding for external data into a program. Global resource declarations are implicitly initialized in an implementation-defined way.

2 Use of an uninitialized resource is undefined.

11.3.Typed Buffers [Resources.TypedBuffers]

1 The typed buffer class template represents a one-dimensional resource containing an array of a single given type. Its contents are indexed by typed access to each element. The memory backing a resource need not be formatted in the same type as the resource's element type. If required, typed buffer elements are converted upon load in an implementation-defined way.

2 There are two typed buffer class templates `Buffer` and `RWBuffer`, which respectively represent read-only and read-write typed buffer objects.

```
template <typename T = float4>
requires hlsl::is_vector<T>::value || hlsl::is_arithmetic<T>::value
class Buffer {
  Buffer();
  Buffer(const Buffer &buf);
  Buffer &operator=(Buffer &buf);
  Buffer(const __DynamicResourceHandle &handle);

  void GetDimensions(out uint width) const;

  // Element access.
  T operator[](int index) const;
  T Load(int index) const;
  T Load(int index, out uint status) const;
};


template <typename T>
requires hlsl::is_vector<T>::value || hlsl::is_arithmetic<T>::value
class RWBuffer {
  RWBuffer();
  RWBuffer(RWBuffer buf);
  RWBuffer &operator=(RWBuffer &buf);
```

```
  Buffer(const __DynamicResourceHandle &handle);

  void GetDimensions(out uint width) const;

  // Element access.
  T operator[](int index) const;
  device T &operator[](int n) const;
  T Load(int index) const;
  T Load(int index, out uint status) const;
};
```

3 The element type of a typed buffer, `T`, must be an arithmetic type (6.9.1) or vector of arithmetic type containing no more than 4 elements, and the total size of the element (`sizeof(T)`) must not exceed 16-bytes in size.

4 The `Buffer` type's template argument `T` has a default value of `float4`.

11.3.1. Typed Buffer Constructors [Resources.TypedBuffers.Ctrs]

```
Buffer();
RWBuffer();
```

- *Effects:* Constructs an uninitialized resource handle.

```
Buffer(const Buffer &buf);
RWBuffer(const RWBuffer &buf);
```

- *Effects:* Initializes a resource to refer to the same resource as the input argument `buf`.

```
Buffer(const __DynamicResourceHandle &handle);
```

- *Feature:* Dependent on dynamic resources (**??**).
- *Effects:* Initializes a resource to refer to the dynamic resource handle provided as the input argument `handle`.

11.3.2. Typed Buffer Copying and Assignment [Resources.TypedBuffers.Copy]

```
Buffer &operator=(Buffer &buf);
RWBuffer &operator=(RWBuffer &buf);
```

- *Effects:* Initializes a resource to refer to the same resource as the input argument `buf`.
- *Returns:* `this`.

11.3.3. Typed Buffer Inspecting Functions [Resources.TypedBuffers.Inspect]

```
void GetDimensions(out uint width) const;
```

- *Effects:* Writes the total number of elements contained in the typed buffer to the `width` argument.

```
T Load(uint index) const;
```

- *Requires:* `index` $<$ the number of elements in the typed buffer.
- *Returns:* The element in the typed buffer at the index specified by `index`.

```
T Load(int index, out uint status) const;
```

- *Requires:* `index` $<$ the number of elements in the typed buffer.
- *Returns:* The element in the typed buffer at the index specified by `index`.
- *Effects:* Initializes the `status` argument to an implementation-defined status signal (11.4).
- *Remarks:* The `status` parameter returns an implementation-defined value that indicates if the loaded value came from fully mapped parts of the resource. This parameter must be passed to the built-in function `CheckAccessFullyMapped` (11.4) in order to interpret it.

11.3.4. Typed Buffer Operators [Resources.TypedBuffers.Operators]

```
T operator[](uint index) const;
```

- *Requires:* `index` < the number of elements in the typed buffer.
- *Returns:* The element in the typed buffer at the index specified by `index`.

```
device T &operator[](int index);
```

- *Requires:* `index` < the number of elements in the typed buffer.
- *Returns:* An lvalue-reference to the element in the typed buffer at the index specified by `index`.
- *Remarks:* Writing to typed buffer elements overwrite the entire element, even if only part of the data is written. A partial write to a typed buffer is a load-insert-store operation.

NOTE	Accurately modeling the memory operations for writable typed buffers would require a proxy-object interface since the address written to isn't actually the device memory since it may be converted or otherwise normalized by the implementation.

11.4. CheckAccessFullyMapped [Resources.Mapcheck]

1 The implementation-defined status value returned by texture and buffer load methods is interpreted by a built-in function:

```
bool CheckAccessFullyMapped(uint status);
```

- *Requires:* `status` is an initialized value from a resource `Load` function's `status` output parameter.
- *Returns:* `true` if the value was accessed from memory that was fully mapped by the runtime implementation; otherwise returns `false`.

Annex A

(informative)

Grammar Summary [AnnexA]

preprocessing-token:
header-name
identifier
pp-number
character-literal
string-literal
preprocessing-op-or-punc
each non-whitespace character from the *translation character set* that cannot be one of the above

token:
identifier
keyword
literal
operator-or-punctuator

header-name:
< *h-char-sequence* >
" *q-char-sequence* "

h-char-sequence:
h-char
h-char-sequence h-char

h-char:
any character in the *translation character set* except newline or >

q-char-sequence:
q-char
q-char-sequence q-char

q-char:
any character in the *translation character set* except newline or "

pp-number:
digit
. *digit*
pp-number ' *digit*
pp-number ' *non-digit*
pp-number `e` *sign*
pp-number `E` *sign*
pp-number `p` *sign*
pp-number `P` *sign*
pp-number .

identifier:
identifier-start
identifier identifier-continue

identifier-start:
nondigit
an element of the translation character set of class XID_Start

identifier-continue:
digit
nondigit
an element of the translation character set of class XID_Continue

nondigit: one of
`a b c d e f g h i j k l m`

```
n o p q r s t u v w z y z
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z _
```

digit: one of

```
0 1 2 3 4 5 6 7 8 9
```

keyword: any of

```
auto bool break case cbuffer center centroid class column_major
const constexpr continue discard do double else enum export
false float for globallycoherent groupshared if in indices
inline inout int interface line lineadj linear namespace nointerpolation
noperspective operator out packoffset payload point precise
primitives reordercoherent return row_major sampler_state shared sizeof
snorm static struct switch tbuffer template this triangle
triangleadj true typedef typename uniform unorm unsigned using vertices
void while
```

preprocessing-op-or-punc:

preprocessing-operator
operator-or-punctuator

preprocessing-operator:

```
#
##
```

operator-or-punctuator: one of

```
{ } [ ] ( ) ; :  ...
?  ::  .
!  + - * / % & |
= += -= *= /= %= ^= &= |=
== += < > <= >= && ||
<< >> <<= >>= ++ -- ,
```

literal:

integer-literal
character-literal
floating-literal
string-literal
boolean-literal
vector-literal

integer-literal:

decimal-literal *integer-suffix*$_{opt}$
octal-literal *integer-suffix*$_{opt}$
hexadecimal-literal *integer-suffix*$_{opt}$

decimal-literal:

nonzero-digit
decimal-literal *digit*

octal-literal: `0`

octal-literal *octal-digit*

hexadecimal-literal:

`0x` *hexadecimal-digit*
`0X` *hexadecimal-digit*
hexadecimal-literal *hexadecimal-digit*

nonzero-digit: one of

```
1 2 3 4 5 6 7 8 9
```

octal-digit: one of

```
0 1 2 3 4 5 6 7
```

hexadecimal-digit: one of

`0 1 2 3 4 5 6 7 8 9`
`a b c d e f`
`A B C D E F`

integer-suffix:
unsigned-suffix long-suffix$_{opt}$
long-suffix unsigned-suffix$_{opt}$

unsigned-suffix: one of
`u U`

long-suffix: one of
`l L`

floating-literal:
fractional-constant exponent-part$_{opt}$ *floating-suffix*$_{opt}$
digit-sequence exponent-part floating-suffx$_{opt}$

fractional-constant:
digit-sequence$_{opt}$ `.` *digit-sequence*
digit-sequence `.`

exponent-part:
`e` *sign*$_{opt}$ *digit-sequence*
`E` *sign*$_{opt}$ *digit-sequence*

sign: one of
`+ -`

digit-sequence:
digit
digit-sequence digit

floating-suffix: one of
`h f l`
`H F L`
`f16 f32 f64`
`F16 F32 F64`

translation-unit:
declaration-sequence$_{opt}$

primary-expression:
literal
`this`
`(` *expression* `)`
id-expression

id-expression:
unqualified-id
qualified-id

unqualified-id:
identifier
operator-function-id
conversion-function-id
template-id

qualified-id:
nested-name-specifier `template`$_{opt}$ *unqualified-id*

nested-name-specifier:
`::`
type-name `::`
namespace-name `::`

nested-name-specifier identifier `::`
nested-name-specifier `template`$_{opt}$ *simple-template-id* `::`

postfix-expression:
primary-expression
postfix-expression `[` *expression* `]`
postfix-expression `[` *braced-init-list* `]`
postfix-expression `(` *expression-list*$_{opt}$ `)`
simple-type-specifier `(` *expression-list*$_{opt}$ `)`
typename-specifier `(` *expression*$_{opt}$ `)`
postfix-expression `.` `template`$_{opt}$ *id-expression*
postfix-expression `.` *vector-swizzle-component-sequence*
postfix-expression `.` *matrix-swizzle-component-sequence*
postfix-expression `++`
postfix-expression `--`

vector-swizzle-component-sequence:
swizzle-component-sequence-rgba
swizzle-component-sequence-xyzw

swizzle-component-sequence-rgba:
swizzle-component-rgba
swizzle-component-sequence-rgba swizzle-component-rgba

swizzle-component-sequence-xyzw:
swizzle-component-xyzw
swizzle-component-sequence-xyzw swizzle-component-xyzw

swizzle-component-rgba: one of
`r g b a`

swizzle-component-xyzw: one of
`x y z w`

matrix-swizzle-component-sequence:
matrix-zero-indexed-swizzle-sequence
matrix-one-indexed-swizzle-sequence

matrix-zero-indexed-swizzle-sequence:
matrix-zero-indexed-swizzle
matrix-zero-indexed-swizzle-sequence matrix-zero-indexed-swizzle

matrix-zero-indexed-swizzle:
`_m` *zero-index-value zero-index-value*

zero-index-value: one of
`0 1 2 3`

matrix-one-indexed-swizzle-sequence:
matrix-one-indexed-swizzle
matrix-one-indexed-swizzle-sequence matrix-one-indexed-swizzle

matrix-one-indexed-swizzle:
`_` *one-index-value one-index-value*

one-index-value: one of
`1 2 3 4`

declaration-seq:
declaration
declaration-seq declaration

declaration:
name-declaration
special-declaration

name-declaration:
block-declaration
function-definition
template-declaration
namespace-definition
empty-declaration
attribute-declaration
cbuffer-declaration

special-declaration:
export-declaration-group
cbuffer-member-declaration

block-declaration:
simple-declaration
namespace-alias-definition
using-declaration
using-directive
static_assert-declaration
alias-declaration
opaque-enum-declaration

empty-declaration:
;
decl-specifier:
storage-class-specifier
type-specifier
function-specifier
`typedef`

decl-specifier-seq:
decl-specifier
decl-specifier decl-specifier-seq

function-specifier:
`export`

initializer:
brace-or-equal-initializer
`(` *expression-list* `)`

brace-or-equal-initializer:

= *initializer-clause*
braced-init-list

initializer-clause:
assignment-expression
braced-init-list

braced-init-list:
{ *initializer-list* ,$_{opt}$ }
{ }

initializer-list:
initializer-clause
initializer-list , *initializer-clause*

Annex B

(informative)

Implementation Documentation [AnnexB]

B.1.Pre-standard Language Versions [AnnexB.Versions]

1 HLSL has evolved over time, with various versions of the language being released alongside different versions of DirectX. Beginning with the DirectX Shader Compiler (DXC), the compiler began supporting language modes corresponding to specific historical versions of HLSL. These historical versions are useful for understanding the evolution of the language and for documenting observed behaviors in specific implementations, and they are named for the years in which they were released. The following historical versions of HLSL are relevant to this specification:

- **HLSL 2015**: Roughly equivalent to the version of HLSL supported by the Effects Compiler (FXC).
- **HLSL 2016**: The initial version of HLSL supported by the DirectX Shader Compiler (DXC).
- **HLSL 2017**: Added enumerations similar to C++11.
- **HLSL 2018**: Added support for 16-bit data types and intrinsic templates.
- **HLSL 2021**: Added user-defined templates, operator overloading, short-circuiting boolean operators, and stricter implicit casting rules for user-defined structures.

NOTE	Both the DirectX Shader Compiler (DXC) and Clang support a language mode called "HLSL 202x" which refers to this draft specification.

B.2.Known Implementations [AnnexB.Implementations]

1 This section provides a list of known implementations of HLSL, including both conforming and non-conforming implementations. This is not meant to be comprehensive, but is provided to supplement later sections in this annex which discuss details of specific implementations.

- **Clang**: The open-source C/C++ compiler developed by the LLVM project. Clang contains a partial implementation of HLSL this specification and is actively being developed. It is available at `https://clang.llvm.org/`.
- **DirectX Shader Compiler (DXC)**: The open-source reference implementation of HLSL, developed by Microsoft and based on the LLVM/Clang infrastructure. It is available at `https://github.com/microsoft/DirectXShaderCompiler`.
- **Effects Compiler (FXC)**: The legacy implementation of HLSL, developed by Microsoft. It is included with the Windows SDK and is available at `https://learn.microsoft.com/en-us/windows/apps/windows-sdk/`.
- **GLSLang**: An open-source implementation of the OpenGL Shading Language (GLSL), developed by The Khronos Group. GLSLang contains a (now deprecated) partial implementation of HLSL 2018. It is available at `https://github.com/KhronosGroup/glslang`.
- **Slang**: An open-source shader language and compiler framework developed by The Khronos Group. Slang contains a partial implementation of HLSL 2018. It is available at `https://github.com/shader-slang/slang`.

B.3.Implementation Limits [AnnexB.Limits]

1 This section describes reasonable implementation limits that may be assumed by a conforming implementation. These limits specify minimum required or maximum allowed values for various implementation-defined parameters.

2 6.9 declares a vector type as having required supported sizes of 1 to 4 components. Implementations may support larger vector sizes but are not required to do so. An implementation that supports larger vector sizes must define its vector size limit.

B.4.Observed Deviations [AnnexB.Documentation]

1 This section provides documentation around implementation behaviors that are known to differ between specific implementations. This includes breaking changes between historical HLSL and standard HLSL where some compilers or versions of compilers may not conform to standard HLSL, as well as observed non-conformant behaviors in such compilers. This is not meant to be a comprehensive list of bugs.

B.4.1.Resource Objects [AnnexB.Resources]

1 In violation of 6.9.3, DXC supports storing objects that contain resources in `cbuffer` declarations. In this use case the resource objects are not actually stored in the `cbuffer`, but instead the resource is treated as a global resource declaration and name lookup resolves the value inside the structure declaration to the implied global resource declaration.

Acronyms

API Application Programming Interface. 41

DXC DirectX Shader Compiler. iii

FXC Legacy DirectX Shader Compiler. iii

HLSL High Level Shader Language. iii, 1–4, 16, 17, 27

SIMD Single Instruction Multiple Data. 2, 3

SIMT Single Instruction Multiple Thread. 3

SPMD Single Program Multiple Data. i, 2, 41

Glossary

DirectX DirectX is the multimedia API introduced with Windows 95.. iii

Dispatch A group of one or more Thread Groups which comprise the largest unit of a shader execution. Also called: grid, compute space or index space.. 3

ISO/IEC 10646:2020 Universal coded character set.. 1, 7

ISO/IEC 14882:2011 Standard for Programming Language C++.. iii, 1, 4, 8, 10–12, 15, 16, 27

ISO/IEC 14882:2020 Standard for Programming Language C++.. 27

ISO/IEC 60559:2020 IEEE-754 Standard For Floating Point Arithmetic. 15

ISO/IEC 9899:2011 Standard for Programming Language C.. iii, 1, 4, 10

ISO/IEC 9899:2023 Standard for Programming Language C.. 15

Lane The computation performed on a single element as described in the SPMD program. Also called: thread.. 2–4, 41

Quad A group of four Lanes which form a cluster of adjacent computations in the data topology. Also called: quad-group or quad-wave. . 3

Thread Group A group of Lanes which may be subdivided into one or more Waves and comprise a larger computation. Also known as: group, workgroup, block or thread block.. 3, 41

UAX #31 Unicode Standard Annex, UAX #31, Unicode Identifiers and Syntax.. 1, 7

UAX #44 Unicode Standard Annex, UAX #44, Unicode Character Database.. 1, 7

Wave A group of Lanes which execute together. The number of Lanes in a Wave varies by hardware implementation. Also called: warp, SIMD-group, subgroup, or wavefront.. 2–4, 41