High-Level Shader Language Specification

Working Draft

August 18, 2026

Contents

1 Introduction **4**
- 1.1 Scope . . . 4
- 1.2 Normative References . . . 4
- 1.3 Terms and definitions . . . 4
- 1.4 Common Definitions . . . 4
  - 1.4.1 Correct Data . . . 5
  - 1.4.2 Diagnostic Message . . . 5
  - 1.4.3 Ill-formed Program . . . 5
  - 1.4.4 Implementation-defined Behavior . . . 5
  - 1.4.5 Implementation Limits . . . 5
  - 1.4.6 Undefined Behavior . . . 5
  - 1.4.7 Unspecified Behavior . . . 5
  - 1.4.8 Well-formed Program . . . 5
  - 1.4.9 Runtime Implementation . . . 5
- 1.5 Runtime Targeting . . . 5
- 1.6 Single Program Multiple Data Programming Model . . . 5
  - 1.6.1 SPMD Terminology . . . 6
  - 1.6.2 SPMD Execution Model . . . 7
  - 1.6.3 Optimization Restrictions . . . 7
- 1.7 HLSL Memory Models . . . 7
  - 1.7.1 Memory Spaces . . . 8
  - 1.7.2 Memory Spaces . . . 8

2 Lexical Conventions **9**
- 2.1 Unit of Translation . . . 9
- 2.2 Phases of Translation . . . 9
- 2.3 Character Sets . . . 9
- 2.4 Preprocessing Tokens . . . 10
- 2.5 Tokens . . . 11
- 2.6 Comments . . . 11
- 2.7 Header Names . . . 11
- 2.8 Preprocessing numbers . . . 11
- 2.9 Literals . . . 12
  - 2.9.1 Literal Classifications . . . 12
  - 2.9.2 Integer Literals . . . 12
  - 2.9.3 Floating-point Literals . . . 13
  - 2.9.4 Vector Literals . . . 14

3 Basic Concepts **15**
- 3.1 Preamble . . . 15
- 3.2 Declarations and definitions . . . 15
- 3.3 One-Definition Rule . . . 16
- 3.4 Scope . . . 17
- 3.5 Name Lookup . . . 17
- 3.6 Storage Duration . . . 17
  - 3.6.1 Static Storage Duration . . . 17
  - 3.6.2 Automatic Storage Duration . . . 17
  - 3.6.3 Program Storage Duration . . . 17
  - 3.6.4 Groupshared Storage Duration . . . 17
- 3.7 Program and linkage . . . 17
  - 3.7.1 Program Linkage . . . 18

3.7.2 External Linkage . . . 18
3.7.3 Internal Linkage . . . 18
3.7.4 No Linkage . . . 18
3.8 Start . . . 19
3.8.1 Execution Mode . . . 19
3.9 Types . . . 19
3.9.1 Arithmetic Types . . . 19
3.9.2 Scalarized Type Compatability . . . 20
3.9.3 Usage of Intangible Types . . . 20
3.10 Lvalues and rvalues . . . 21

4 Standard Conversions **22**
4.1 Lvalue-to-rvalue conversion . . . 22
4.2 Array-to-pointer conversion . . . 22
4.3 Integral promotion . . . 22
4.4 Floating point promotion . . . 22
4.5 Integral conversion . . . 23
4.6 Bit-field conversion . . . 23
4.7 Floating point conversion . . . 23
4.8 Floating point-integral conversion . . . 23
4.9 Boolean conversion . . . 23
4.10 Elementwise conversion . . . 23
4.11 Vector splat conversion . . . 24
4.12 Matrix splat conversion . . . 24
4.13 Array and Class splat conversion . . . 24
4.14 Vector and matrix truncation conversion . . . 24
4.15 Component-wise conversions . . . 25
4.16 Qualification conversion . . . 25
4.17 Conversion Rank . . . 25
4.17.1 Integer Conversion Rank . . . 25
4.17.2 Floating Point Conversion Rank . . . 25

5 Expressions **26**
5.1 Usual Arithmetic Conversions . . . 26
5.2 Primary Expressions . . . 26
5.2.1 Literals . . . 27
5.2.2 This . . . 27
5.2.3 Parenthesis . . . 27
5.2.4 Names . . . 27
5.3 Postfix Expressions . . . 28
5.4 Subscript . . . 28
5.5 Swizzle Expressions . . . 28
5.5.1 Vector Swizzle . . . 28
5.5.2 Matrix Swizzle . . . 29
5.6 Function Calls . . . 30

6 Statements **32**
6.1 Label Statements . . . 32
6.2 Attributes . . . 32
6.2.1 Unroll Attribute . . . 32
6.2.2 Loop Attribute . . . 32

7 Declarations **33**
7.1 Preamble . . . 33
7.2 Specifiers . . . 33
7.2.1 General . . . 33
7.2.2 Function specifiers . . . 33

7.3 Declarators . . . 34
7.4 Initializers . . . 34
7.4.1 Aggregate Initialization . . . 34
7.5 Function Definitions . . . 35
7.6 Attributes . . . 35
7.6.1 Semantic Annotations . . . 35
7.6.2 Entry Attributes . . . 35
7.7 Export Declarations . . . 35
7.8 Constant Buffer Declarations . . . 35

8 Overloading **37**
8.1 Overloadable Declarations . . . 37
8.2 Overload Resolution . . . 38
8.2.1 Candidate Functions and Argument Lists . . . 38
8.2.2 Viable Functions . . . 39
8.2.3 Best Viable Function . . . 39
8.2.4 Implicit Conversion Sequences . . . 39

9 Resource Types **42**
9.1 Typed Buffers . . . 42
9.1.1 Constructors . . . 42
9.1.2 Dimensions . . . 43
9.1.3 Element Access . . . 43
9.2 Raw Buffers . . . 43
9.3 Byte Access Buffers . . . 44
9.3.1 Constructors . . . 45
9.3.2 Dimensions . . . 46
9.3.3 Element Access . . . 46
9.3.4 Atomic Operations . . . 46
9.4 Structured Buffers . . . 48
9.4.1 Constructors . . . 49
9.4.2 Dimensions . . . 50
9.4.3 Element Access . . . 50
9.4.4 Counter Manipulation . . . 50
9.4.5 Append . . . 50
9.4.6 Consume . . . 51
9.5 Constant Buffers . . . 51
9.5.1 Constant Buffer Class . . . 51
9.5.2 Constant Buffer Layout . . . 51
9.5.3 Packoffset annotations . . . 52
9.5.4 Default Constant Buffer . . . 52
9.6 Samplers . . . 53
9.7 CheckAccessFullyMapped . . . 53
9.8 Resource Binding . . . 53

10 Templates **55**
10.1 Template Instantiation . . . 55
10.2 Partial Ordering of Function Templates . . . 55

11 Intangible Types **56**

12 Runtime **57**

Acronyms **58**

Glossary **59**

1 Introduction [Intro]

1 The High Level Shader Language (HLSL) is the GPU programming language 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. Despite its popularity and long history HLSL has never had a formal language specification. This document seeks to change that.

2 HLSL draws heavy inspiration originally from ISO C standard (2011) and later from ISO C++ standard (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.

1.1 Scope [Intro.Scope]

1 This document specifies the requirements for implementations of HLSL. The HLSL specification is based on and highly influenced by the specifications for the C Programming Language (C) and the C++ Programming Language (C++).

2 This document covers both describing the language grammar and semantics for HLSL, and (in later sections) the standard library of data types used in shader programming.

1.2 Normative References [Intro.Refs]

1 The following referenced documents provide significant influence on this document and should be used in conjunction with interpreting this standard.

- ISO C standard (2011), *Programming languages - C*
- ISO C++ standard (2011), *Programming languages - C++*
- DirectX Specifications, *https://microsoft.github.io/DirectX-Specs/*

1.3 Terms and definitions [Intro.Terms]

1 This document aims to use terms consistent with their definitions in ISO C standard (2011) and ISO C++ standard (2011). In cases where the definitions are unclear, or where this document diverges from ISO C standard (2011) and ISO C++ standard (2011), the definitions in this section, the remaining sections in this chapter, and the attached glossary (12) supersede other sources.

1.4 Common Definitions [Intro.Defs]

1 The following definitions are consistent between HLSL and the ISO C standard (2011) and ISO C++ standard (2011) specifications, however they are included here for reader convenience.

1.4.1 Correct Data [Intro.Defs.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.

1.4.2 Diagnostic Message [Intro.Defs.Diags]

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

1.4.3 Ill-formed Program [Intro.Defs.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.

1.4.4 Implementation-defined Behavior [Intro.Defs.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.

1.4.5 Implementation Limits [Intro.Defs.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.

1.4.6 Undefined Behavior [Intro.Defs.Undefined]

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

1.4.7 Unspecified Behavior [Intro.Defs.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.

1.4.8 Well-formed Program [Intro.Defs.WellFormed]

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

1.4.9 Runtime Implementation [Intro.Defs.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.

1.5 Runtime Targeting [Intro.Runtime]

1 HLSL emerged from the evolution of DirectX to grant greater control over GPU geometry and color processing. It gained popularity because it targeted a common hardware description which all conforming drivers were required to support. This common hardware description, called a Shader Model, is an integral part of the description for HLSL . Some HLSL features require specific Shader Model features, and are only supported by compilers when targeting those Shader Model versions or later.

1.6 Single Program Multiple Data Programming Model [Intro.Model]

1 HLSL uses a Single Program Multiple Data (SPMD) programming model where a program describes operations on a single element of data, but when the program executes it executes across more than one element at a time. This

programming model is useful due to GPUs largely being Single Instruction Multiple Data (SIMD) hardware architectures where each instruction natively executes across multiple data elements at the same time.

2 There are many different terms of art for describing the elements of a GPU architecture and the way they relate to the SPMD program model. In this document we will use the terms as defined in the following subsections.

1.6.1 SPMD Terminology [Intro.Model.Terms]

1.6.1.1 Host and Device [Intro.Model.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.

1.6.1.2 Lane [Intro.Model.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 (1.6.1.7).

1.6.1.3 Wave [Intro.Model.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.

1.6.1.4 Quad [Intro.Model.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.

1.6.1.5 Thread Group [Intro.Model.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.

1.6.1.6 Dispatch [Intro.Model.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.

1.6.1.7 Lane States [Intro.Model.Terms.LaneState]

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

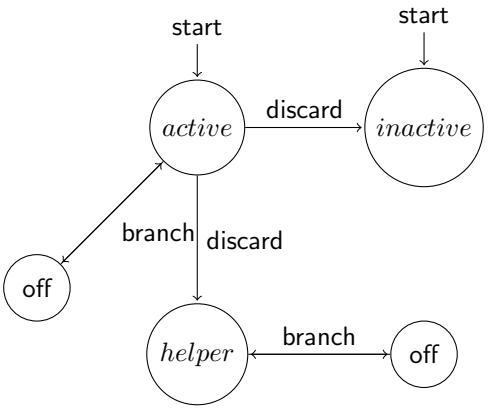
2 An *active* Lane is enabled to perform computations and produce output results based on the initial launch conditions and program control flow.

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 A *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:



1.6.2 SPMD Execution Model [Intro.Model.Exec]

1 A runtime implementation shall provide an implementation-defined mechanism for defining a Dispatch. A runtime shall manage hardware resources and schedule execution to conform to the behaviors defined in this specification in an implementation-defined way. A runtime implementation may sort the Thread Groups of a Dispatch into Waves in an implementation-defined way. During execution no guarantees are made that all Lanes in a Wave are actively executing.

2 Wave, Quad, and Thread Group operations require execution synchronization of applicable active and helper Lanes as defined by the individual operation.

1.6.3 Optimization Restrictions [Intro.Model.Restrictions]

1 An optimizing compiler may not optimize code generation such that it changes the behavior of a well-formed program except in the presence of *implementation-defined* or *unspecified* behavior.

2 The presence of Wave, Quad, or Thread Group operations may further limit the valid transformations of a program. Specifically, control flow operations which result in changing which Lanes, Quads, or Waves are actively executing are illegal in the presence of cooperative operations if the optimization alters the behavior of the program.

1.7 HLSL Memory Models [Intro.Memory]

1 Memory accesses for Shader Model 5.0 and earlier operate on 128-bit slots aligned on 128-bit boundaries. This optimized for the common case in early shaders where data being processed on the GPU was usually 4-element vectors of 32-bit data types.

2 On modern hardware memory access restrictions are loosened, and reads of 32-bit multiples are supported starting with Shader Model 5.1 and reads of 16-bit multiples are supported with Shader Model 6.0. Shader Model features are fully documented in the DirectX Specifications, and this document will not attempt to elaborate further.

1.7.1 Memory Spaces [Intro.Memory.Spaces]

1 HLSL programs manipulate data stored in four distinct memory spaces: thread, threadgroup, device and constant.

1.7.1.1 Thread Memory [Intro.Memory.Spaces.Thread]

1 Thread memory is local to the Lane. It is the default memory space used to store local variables. Thread memory cannot be directly read from other threads without the use of intrinsics to synchronize execution and memory.

1.7.1.2 Thread Group Memory [Intro.Memory.Spaces.Group]

1 Thread Group memory is denoted in HLSL with the `groupshared` keyword. The underlying memory for any declaration annotated with `groupshared` is shared across an entire Thread Group. Reads and writes to Thread Group Memory, may occur in any order except as restricted by synchronization intrinsics or other memory annotations.

1.7.1.3 Device Memory [Intro.Memory.Spaces.Device]

1 Device memory is memory available to all Lanes executing on the device. This memory may be read or written to by multiple Thread Groups that are executing concurrently. Reads and writes to device memory may occur in any order except as restricted by synchronization intrinsics or other memory annotations. Some device memory may be visible to the host. Device memory that is visible to the host may have additional synchronization concerns for host visibility.

1.7.1.4 Constant Memory [Intro.Memory.Spaces.Constant]

1 Constant memory is similar to device memory in that it is available to all Lanes executing on the device. Constant memory is read-only, and an implementation can assume that constant memory is immutable and cannot change during execution.

1.7.2 Memory Spaces [Intro.Memory.Alignment]

TODO

1 The alignment requirements of an offset into device memory space is the size in bytes of the largest scalar type contained in the given aggregate type.

2 Lexical Conventions [Lex]

2.1 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 C standard (2011) preprocessor specification.

2 An implementation may implicitly include additional sources as required to expose the HLSL library functionality as defined in (12).

2.2 Phases of Translation [Lex.Phases]

1 HLSL inherits the phases of translation from ISO C++ standard (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.

2.3 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.

2.4 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

[1]The preprocessor is inherited from C++ 11 with no grammar extensions. It is specified here only for completeness.

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 (2.2). 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.

2.5 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.

2.6 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.

2.7 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.

2.8 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* (2.9.4). In this situation the *pp-number* token is truncated to end before the period[2].

2.9 Literals [Lex.Literals]

2.9.1 Literal Classifications [Lex.Literal.Kinds]

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

2.9.2 Integer Literals [Lex.Literal.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`

[2] This grammar formulation is not context-free and requires an LL(2) parser.

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 an optional base prefix, a sequence of digits in the appropriate base, 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[3].

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.

2.9.3 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`

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*.

[3] This behavior matches ISO C standard (2011) but is reduced in scope because HLSL has fewer data types.

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`.[4] If a value specified in the source is not in the range of representable values for its type, the program is ill-formed.

2.9.4 Vector Literals [Lex.Literal.Vector]

vector-literal:
    integer-literal `.` *scalar-element-sequence*
    floating-literal `.` *scalar-element-sequence*

scalar-element-sequence:
    scalar-element-sequence-x
    scalar-element-sequence-r

scalar-element-sequence-x:
    `x`
    scalar-element-sequence-x `x`

scalar-element-sequence-r:
    `r`
    scalar-element-sequence-r `r`

1 A *vector-literal* is an *integer-literal* or *floating-point* literal followed by a period (`.`) and a *scalar-element-sequence*.

2 A *scalar-element-sequence* is a *vector-swizzle-sequence* where only the first vector element accessor is valid (`x` or `r`). A *scalar-element-sequence* is equivalent to a vector splat conversion performed on the *integer-literal* or *floating-literal* value (4.11).

[4] This substantially deviates from the implementations in FXC and DXC, but is consistent with the official documentation and the behavior of GLSL. It is also substantially simpler to implement and more regular than the existing behaviors.

3 Basic Concepts [Basic]

> 1 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.
>
> 2 This chapter includes a lot of definitions that are inherited from C and C++. Some are identical to C or C++, others are slightly different. HLSL is neither a subset nor a superset of C or C++, and cannot be simply described in terms of C or C++. This specification includes all necessary definitions for clarity.

3.1 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* (5.2.4), *operator-function-id* (**??**), *conversion-function-id* (**??**), or *template-id* (10) that denotes any entity or *label* (6.1).

3 Every name that denotes an entity is introduced by a *declaration*. Every name that denotes a label is introduced by a *labeled statement* (6.1)[1].

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 Whenever a name is encountered it is necessary to determine if the name denotes an entity that is a type or template. The process for determining if a name refers to a type or template is called *name lookup*.

6 Two names are the same name if:

- they are identifiers comprised 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.

7

> This section matches ISO C++ standard (2011) section **[basic]** except for the exclusion of `goto` and *literal operators*.

3.2 Declarations and definitions [Basic.Decl]

1 A declaration (7) 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 (7),
- use of attributes (7), and
- controlling template instantiation (10.1).

2 A declaration is a *definition* unless:

[1] HLSL does not have `goto`, and labeled statements are only valid within `switch` statements.

- it declares a function without specifying the function's body (7.5),
- it is a parameter declaration in a function declaration that does not specify the function's body (7.5),
- it is a global or namespace member declaration without the `static` specifier[2],
- 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 (7),
- it is an *alias-declaration* (7),
- it is a *using-declaration* (7),
- it is a *static_assert-declaration* (7),
- it is an *empty-declaration* (7),
- or a *using-directive* (7).

3 The two examples below are adapted from ISO C++ standard (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
}
```

3.3 One-Definition Rule [Basic.ODR]

1 The ISO C++ standard (2011) *One-definition rule* is adopted as defined in ISO C++ standard (2011) **[basic.def.odr]**.

[2]Global variable declarations are implicitly constant and external in HLSL.

3.4 Scope [Basic.Scope]

3.5 Name Lookup [Basic.Lookup]

3.6 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*

3.6.1 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.

3.6.2 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.

3.6.3 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.

3.6.4 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.

3.7 Program and linkage [Basic.Linkage]

1 A translation unit (2.1) 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 (3.8). 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.

3.7.1 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 (3.8)
- functions marked with `export` keyword (7.7)
- declarations contained within an *export-declaration-group* (7.7)

3.7.2 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` [3]
- 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. [4]

3.7.3 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

3.7.4 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

[3] These are not really linked with other translation units but rather their values are loaded indirectly based on cbuffer mapping.

[4] In DXC today functions that are not entry points or exported have *internal linkage* by default. This can be overriden by `-default-linkage` compiler option.

3.8 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 (3.8.1).

3 Parameters to entry functions and entry function return types must be of scalar, vector, or non-intangible class type (3.9). Scalar and vector parameters and return types must be annotated with semantic annotations (7.6.1). Class type input and output parameters must have all fields annotated with semantic annotations.

3.8.1 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.

3.9 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)`[5]. The *object representation* of an object may be different based on the *memory space* it is stored in (1.7.1).

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 3.9.3.

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*(11).

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 (3.9.1), 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*. Vector lengths must be between 1 and 4 (i.e. $1 \leq N \leq 4$).

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. $1 \leq N \leq 4$).

3.9.1 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

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

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 ISO/IEC 9899:2023: Standard for Programming Language C.[6]. Integer types shall satisfy the constraints defined in ISO/IEC 14882:2011: 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. The `double` type is a 64-bit floating point type. Both the `float` and `double` types have object representations as defined in IEEE Standard 754. 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 IEEE Standard 754, otherwise it will have an object representation matching the **binary16** format defined in IEEE Standard 754[7]. 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`[8]. 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 C++ standard (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`.

3.9.2 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 $SR(T_0), ..SR(T_n)$.
- The scalarized representation of a vector `vector<T,n>` is $T_0, ..T_n$.
- The scalarized representation of a matrix `matrix<T,n, m>` is $T_0, ..T_{n \times m}$.
- 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.

3.9.3 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 (1.7.1).

[6] C23 adopts two's compliment as the object representation for integer types.
[7] IEEE-754 only defines a binary encoding for 16-bit floating point values, it does not fully specify the behavior of such types.
[8] This means when stored to memory objects of type `min16float` are stored as **binary32** as defined in IEEE Standard 754.

- An object of intangible type may not be loaded or stored to any address space other than the Thread address space (1.7.1).[9]
- An object of intangible type may not be a parameter or return type of a function with program linkage or external linkage (3.7.1 and 3.7.2).

3.10 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*.

[9] DXC supports storing objects that contain resources in cbuffer declarations, but it does so by hoisting the resource declaration out to the global scope.

4 Standard Conversions [Conv]

1 HLSL inherits standard conversions similar to ISO C++ standard (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, or boolean conversion, derived-to-base-lvalue, or flat conversion[1].
3. Zero or one conversion of scalar-vector splat, or vector/matrix truncation. [2].
4. Zero or one qualification conversion.

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

4.1 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.

4.2 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`[3][4].

4.3 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 4.17.1.

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

4.4 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.

[1] This differs from C++ with the addition of flat conversion.

[2] C++ does not support dimension altering conversions for scalar, vector or matrix types.

[3] 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.

[4] Array-to-pointer conversion of constant sized arrays is not supported.

2 Floating point conversion ranks are defined in section **??**.

4.5 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.

4.6 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′`.

4.7 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.

4.8 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.

4.9 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`.

4.10 Elementwise conversion [Conv.flat]

1 For a given aggregate `A`, there is a flattened order of subojects, `A′`, as described in section 7.4.1. 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`.

4.11 Vector splat conversion [Conv.vsplat]

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

4.12 Matrix splat conversion [Conv.msplat]

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

4.13 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 **??** 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]`.

4.14 Vector and matrix truncation conversion [Conv.vtrunc]

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

- `vector<T,y>` only if $y < x$, or
- `T`

2 The resulting value of vector truncation is comprised of elements $[0..y)$, dropping elements $[y..x)$.

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

- `matrix<T,z,w>` only if $x \geq z$ and $y \geq w$,
- `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 $[0..z)$ which are each separately comprised of elements $[0..w)$. 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,x,1>`), can produce either a single row matrix, a vector of type `vector<T,x>`, or a scalar of type `T`.

4.15 Component-wise conversions [Conv.cwise]

1 A glvalue of type `vector<T,x>` can be converted to a cxvalue of type `vector<V,x>`, or a prvalue of type `vector<T,x>` can be converted to a prvalue of type `vector<V,x>`. 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 4.

2 A glvalue of type `matrix<T,x,y>` can be converted to a cxvalue of type `matrix<V,x,y>`, or a prvalue of type `matrix<T,x,y>` can be converted to a prvalue of type `matrix<V,x,y>`. 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 4.

4.16 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`".

4.17 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: **??** and **??**).

4.17.1 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`.

4.17.2 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`.

5 Expressions [Expr]

1 This chapter defines the formulations of expressions and the behavior of operators when they are not overloaded. Only member operators may be overloaded[1]. 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[2].

3 Whenever a *glvalue* appears in an expression that expects a *prvalue*, a standard conversion sequence is applied based on the rules in 4.

5.1 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 (4.15). 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 (4.11) 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 (4.14).
- 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 (4).
  - If both operands are scalar or vector elements of signed or unsigned 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.

5.2 Primary Expressions [Expr.Primary]

primary-expression:
literal

[1] This will change in the future, but this document assumes current behavior.

[2] 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.

`this`
`(` *expression* `)`
id-expression

5.2.1 Literals [Expr.Primary.Literal]

1 The type of a *literal* is determined based on the grammar forms specified in 2.9.1.

5.2.2 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 *prvalue* of non-*cv-qualified*type. [3]

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

5.2.3 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.

5.2.4 Names [Expr.Primary.ID]

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

id-expression:
unqualified-id
qualified-id

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

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

5.2.4.2 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* `::`

[3] HLSL Specs Proposal 0007 proposes adopting C++-like syntax and semantics for *cv-qualified* `this` references.

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

5.4 Subscript [Expr.Post.Subscript]

1 A *postfix-expression* followed by an expression in square brackets (`[ ]`) is a subscript expression. In an array subscript expression of the form `E1[E2]`, `E1` must either be a variable of array, vector, or matrix of `T[]`, or an object of type `T` where `T` provides an overloaded implementation of `operator[]` (8).[4]

2 If the postfix expression `E1` is of vector type, the expression *E2* must be a value of integer type. If the value is known at compile time to be outside the range `[0, N-1]` where `N` is the number of elements in the vector, the program is ill-formed. If the value is outside the range at runtime, the behavior is undefined.

5.5 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.

5.5.1 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`

[4] HLSL does not support the base address of a subscript operator being the expression inside the braces, which is valid in C and C++.

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.

5.5.2 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

5.6 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. [5]

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 (7.5). 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]

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 (3.10). 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[7]. The argument expression must be of a type or able to convert

[5] Today in DXC targeting DXIL matches the Microsoft C++ ABI and evaluates argument expressions right-to-left, while SPIR-V generation matches the Itanium ABI evaluating parameters left-to-right. There are good arguments for unifying these behaviors, and arguments for keeping them different.

[6] This results in *input* parameters of unsized arrays being modifiable by a function.

[7] The argument expression is not re-evaluated after the call, so any side effects of the call occur only before the call.

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.

9 If the lvalue passed to an *output* or *input/output parameter* does not alias any other parameter passed to that function, an implementation may avoid the creation of excess temporaries by passing the address of the lvalue instead of creating the cxvalue.

10 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.[8] The lifetime of a parameter ends on return of the function in which it is defined.[9] Initialization and destruction of each parameter occurs within the context of the calling function.

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

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

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

[8] HLSL *output* and *input/output parameters* are passed by value, so they must also have complete type.

[9] As stated above cxvalue parameters are passed-by-address, so the expiring parameter is the reference to the address, not the cxvalue. The cxvalue expires in the caller.

6 Statements [Stmt]

statement:
labeled-statement
attribute-specifier-sequence$_{opt}$ *expression-statement*
attribute-specifier-sequence$_{opt}$ *compound-statement*
attribute-specifier-sequence$_{opt}$ *iteration-statement*
attribute-specifier-sequence$_{opt}$ *selection-statement*
declaration-statement

6.1 Label Statements [Stmt.Label]

1 The optional *attribute-specifier-sequence* applies to the statement that immediately follows it.

6.2 Attributes [Stmt.Attr]

6.2.1 Unroll Attribute [Stmt.Attr.Unroll]

1 The *[unroll]* attribute is only valid when applied to *iteration-statements*. It is used to indicate that *iteration-statements* like `for`, `while` and `do while` can be unrolled. This attribute qualifier can be used to specify full unrolling or partial unrolling by a specified amount. This is a compiler hint and the compiler may ignore this directive.

2 The unroll attribute may optionally have an unroll factor represented as a single argument `n` that is an integer constant expression value greater than zero. If `n` is not specified, the compiler determines the unrolling factor for the loop. The *[unroll]* attribute can not be applied to the same *iteration-statement* as the `[loop]` attribute.

6.2.2 Loop Attribute [Stmt.Attr.Loop]

1 The Attribute `[loop]` tells the compiler to execute each iteration of the loop. In other words, its a hint to indicate a loop should not be unrolled. Therefore it is not compatible with the `[unroll]` attribute.

7 Declarations [Decl]

7.1 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:
;

7.2 Specifiers [Decl.Spec]

7.2.1 General [Decl.Spec.General]

1 The specifiers that can be used in a declaration are

decl-specifier:
function-specifier
...

7.2.2 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 (3.7.1).

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* (7.7).

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* (7.7).

7.3 Declarators [Decl.Decl]

7.4 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*

7.4.1 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.

7.5 Function Definitions [Decl.Function]

7.6 Attributes [Decl.Attr]

7.6.1 Semantic Annotations [Decl.Attr.Semantic]

7.6.2 Entry Attributes [Decl.Attr.Entry]

7.7 Export Declarations [Decl.Export]

1 One or more functions with *external linkage* can be also specified in the form of

export-declaration-group:
`export` { *function-declaration-seq*$_{opt}$ }

function-declaration-seq:
function-declaration function-declaration-seq$_{opt}$

2 The `export` specifier denotes that every *function-declaration* included in *function-declaration-seq* has *external linkage* (3.7.2).

3 The *export-declaration-group* declaration cannot appear directly or indirectly within an unnamed namespace.

4 Functions with *external linkage* can also be declared with an `export` specifier (7.2.2).

5 If a function is part of an *export-declaration-group* then all redeclarations of the same function must also be part on a *export-declaration-group* or be declared with an `export` specifier (7.2.2).

7.8 Constant Buffer Declarations [Decl.cbuffer]

cbuffer-declaration:
`cbuffer` *name resource-binding*$_{opt}$ { *cbuffer-member-seq*$_{opt}$ }

cbuffer-member-seq:
cbuffer-member-declaration
cbuffer-member-seq cbuffer-member-declaration

cbuffer-member-declaration:
block-declaration
function-definition
template-declaration
empty-declaration

1 A *cbuffer declaration* is declared with the `cbuffer` keyword. The name of the cbuffer declaration does not declare a name, and cannot be referenced from within the translation unit, nor is it required to be unique. Each cbuffer declaration refers to a unique constant buffer resource (9.5).

2 The cbuffer declaration itself does not declare a declaration scope. Declarations within a cbuffer declaration that declare names, declare their names in the scope containing the cbuffer declaration. A cbuffer declaration may not contain a *namespace-declaration* or *cbuffer-declaration*. [1]

3 Variable declarations with program storage duration (3.6.3) in the cbuffer declaration are called *shader constants*. Shader constants are implicitly `const` and cannot be modified in program code.[2]

[1] These declarations were previously allowed in HLSL reference compilers but are not supported in this specification.

[2] A future version of this specification will likely disallow variable declarations with storage durations other than program storage duration in cbuffer declarations.

8 Overloading [Overload]

1 HLSL inherits much of its overloading behavior from C++. This chapter is extremely similar to ISO C++ standard (2011) clause **[over]**. Notable differences exist around HLSL's parameter modifier keywords, program entry points, and overload conversion sequence ranking.

2 When a single name is declared with two or more different declarations in the same scope, the name is *overloaded*. A declaration that declares an overloaded name is called an *overloaded declaration*. The set of overloaded declarations that declare the same overloaded name are that name's *overload set*.

3 Only function and template declarations can be overloaded; variable and type declarations cannot be overloaded.

8.1 Overloadable Declarations [Overload.Decl]

1 This section specifies the cases in which a function declaration cannot be overloaded. Any program that contains an invalid overload set is ill-formed.

2 In overload set is invalid if:

- One or more declaration in the overload set only differ by return type.

```
int Yeet();
uint Yeet(); // ill-formed: decls differ only by return type
```

- An overload set contains more than one member function declarations with the same *parameter-type-list*, and one of those declarations is a `static` member function declaration (**??**).

```
class Doggo {
  static void pet();
  void pet();               // ill-formed: static pet has the same parameter-type-list
  void pet() const;         // ill-formed: static pet has the same parameter-type-list

  void wagTail();           // valid: no conflicting static declaration.
  void wagTail() const;     // valid: no conflicting static declaration.

  static void bark(Doggo D);
  void bark();              // valid: static bark parameter-type-list is different
  void bark() const;        // valid: static bark parameter-type-list is different
};
```

- An overload set contains more than one entry function declaration (7.6.2).

```
[shader("vertex")]
void VS();
void VS(int);               // valid: only one entry point.

[shader("vertex")]
void Entry();

[shader("compute")]
void Entry(int);            // ill-formed: an overload set cannot have more than one entry func
```

- An overload set contains more than one function declaration which only differ in parameter declarations of equivalent types.

```
void F(int4 I);
void F(vector<int, 4> I);  // ill-formed: int4 is a type alias of vector<int, 4>
```

- An overload set contains more than one function declaration which only differ in `const` specifiers.

```
void G(int);
void G(const int);          // ill-formed: redeclaration of G(int)
void G(int) {}
void G(const int) {}        // ill-formed: redefinition of G(int)
```

- An overload set contains more than one function declaration which only differ in parameters mismatching `out` and `inout`.

```
void H(int);
void H(in int);             // valid: redeclaration of H(int)
void H(inout int);          // valid: overloading between in and inout is allowed

void I(in int);
void I(out int);            // valid: overloading between in and out is allowed

void J(out int);
void J(inout int);          // ill-formed: Cannot overload based on out/inout mismatch
```

8.2 Overload Resolution [Overload.Res]

1 *Overload resolution* is process by which a function call is mapped to a the best overloaded function declaration. Overload resolution uses set of functions called the *candidate set*, and a list of expressions that comprise the argument list for the call.

2 Overload resolution selects the function to call in the following contexts[1]:

- invocation of a function named in a function call expression;
- invocation of a function call operator on a class object named in function call syntax;
- invocation of the operator referenced in an expression;
- invocation of a user-defined conversion for copy-initialization of a class object;
- invocation of a conversion function for initialization of an object of a nonclass type from an expression of class type.

3 In each of these contexts a unique method is used to construct the overload candidate set and argument expression list.

8.2.1 Candidate Functions and Argument Lists [Overload.Res.Sets]

ISO C++ standard (2011) goes into a lot of detail in this section about how candidate functions and argument lists are selected for each context where overload resolution is performed. HLSL matches C++ for the contexts that HLSL inherits. For now, this section will be left as a stub, but HLSL inherits the following sections from C++:

- **[over.call.func]**
- **[over.call.object]**
- **[over.match.oper]**
- **[over.match.copy]**
- **[over.match.conv]**

[1] DXC only supports overload resolution for function calls and invocation of operators during expressions. Clang will support all contexts listed.

8.2.2 Viable Functions [Overload.Res.Viable]

1 Given the candidate set and argument expressions as determined by the relevant context (8.2.1), a subset of viable functions can be selected from the candidate set.

2 A function candidate $F(P_0...P_m)$ is not a viable function for a call with argument list $A_0...A_n$ if:

- The function has fewer parameters than there are arguments in the argument list ($m < n$).
- The function has more parameters than there are arguments to the argument list ($m > n$), and function parameters $P_{n+1}...P_m$ do not all have default arguments.
- There is not an implicit conversion sequence that converts each argument A_i to the type of the corresponding parameter P_i.

8.2.3 Best Viable Function [Overload.Res.Best]

1 For an overloaded call with arguments $A_0...A_n$, each viable function $F(P_0...P_m)$, has a set of implicit conversion sequences $ICS_0(F)...ICS_m(F)$ defining the conversion sequences for each argument A_i to the type of parameter P_i.

2 A viable function F is defined to be a better function than another viable function F' if for all arguments $ICS_i(F)$ is not a worse conversion sequence than $ICS_i(F')$, and:

- for some argument j, $ICS_j(F)$ is a better conversion than $ICS_j(F')$ or,
- in the context of an initialization by user-defined conversion, the conversion sequence from the return type of F to the destination type is a better conversion sequence than the return type of F' to the destination type or,
- F is a non-template function and F' is a function template specialization, or
- F and F' are both function template specializations and F is more specialized than F' according to function template partial ordering rules (10.2).

3 If there is one viable function that is a better function than all the other viable functions, it is the selected function; otherwise the call is ill-formed.

4 If the resolved overload is a function with multiple declarations, and if at least two of these declarations specify a default argument that made the function viable, the program is ill-formed.

```
void F(int X = 1);
void F(float Y = 2.0f);

void Fn() {
  F(1);      // Okay.
  F(3.0f);   // Okay.
  F();       // Ill-formed.
}
```

8.2.4 Implicit Conversion Sequences [Overload.ICS]

1 An *implicit conversion sequence* is a sequence of conversions which converts a source value to a prvalue of destination type. In overload resolution the source value is the argument expression in a function call, and the destination type is the type of the corresponding parameter of the function being called.

2 When a parameter is a cxvalue an *inverted implicit conversion sequence* is required to convert the parameter type back to the argument type for writing back to the argument expression lvalue. An inverted implicit conversion sequence must be a well-formed implicit conversion sequence where the source value is the implicit cxvalue of the parameter type, and the destination type is the argument expression's lvalue type.

3 A well-formed implicit conversion sequence is either a *standard conversion sequence*, or a *user-defined conversion sequence*.

4 In the following contexts an implicit conversion sequence can only be a standard conversion sequence:

- Argument conversion for a user-defined conversion function.

- Copying a temporary for class copy-initialization.
- When passing an initializer-list as a single argument.
- Copy-initialization of a class by user-defined conversion.

5 An implicit conversion sequence models a copy-initialization unless it is an inverted implicit conversion sequence when it models an assignment. Any difference in top-level cv-qualification is handled by the copy-initialization or assignment, and does not constitute a conversion[2].

6 When the source value type and the destination type are the same, the implicit conversion sequence is an *identity conversion*, which signifies no conversion.

7 Only standard conversion sequences that do not create temporary objects are valid for implicit object parameters or left operand to assignment operators.

8 If no sequence of conversions can be found to convert a source value to the destination type, an implicit conversion sequence cannot be formed.

9 If several different sequences of conversions exist that convert the source value to the destination type, the implicit conversion sequence is defined to be the unique conversion sequence designated the *ambiguous conversion sequence*. For the purpose of ranking implicit conversion sequences, the ambiguous conversion sequence is treated as a user-defined sequence that is indistinguishable from any other user-defined conversion sequence. If overload resolution selects a function using the ambiguous conversion sequence as the best match for a call, the call is ill-formed.

8.2.4.1 Standard Conversion Sequences [Overload.ICS.SCS]

1 The conversions that comprise a standard conversion sequence and the composition of the sequence are defined in Chapter 4.

2 Each standard conversion is given a category and rank as defined in the table below:

Conversion	Category	Rank	Reference
No conversion	Identity	Exact Match	
Lvalue-to-rvalue	Lvalue Transformation		4.1
Array-to-pointer			4.2
Qualification	Qualification Adjustment		4.16
Vector Scalar splat conversion	Scalar Extension Conversion	Conversion Extension	4.11
Matrix Scalar splat conversion	Scalar Extension Conversion	Conversion Extension	4.12
Integral promotion	Promotion	Promotion	4.5 & 4.17.1
Floating point promotion			4.7 & 4.17.2
Component-wise promotion			4.15
Scalar splat promotion	Scalar Extension Promotion	Promotion Extension	4.11
Integral conversion	Conversion	Conversion	4.5
Floating point conversion			4.7
Floating-integral conversion			4.8
Boolean conversion			4.9
Component-wise conversion			4.15
Scalar splat conversion	Scalar Extension Conversion	Conversion Extension	4.11
Vector truncation (without conversion)	Dimensionality Reduction	Truncation	4.14
Vector truncation promotion	Dimensionality Reduction Promotion	Promotion Truncation	4.14
Vector truncation conversion	Dimensionality Reduction Conversion	Conversion Truncation	4.14
Matrix truncation (without conversion)	Dimensionality Reduction	Truncation	4.14
Matrix truncation promotion	Dimensionality Reduction Promotion	Promotion Truncation	4.14
Matrix truncation conversion	Dimensionality Reduction Conversion	Conversion Truncation	4.14

3 If a scalar splat conversion occurs in a conversion sequence where all other conversions are **Exact Match** rank, the conversion is ranked as **Extension**. If a scalar splat occurs in a conversion sequence with a **Promotion** conversion, the

[2] "Top-level" cv-qualification refers to the qualification of the value. This means an parameter of type `T` can be initialized by a argument of type `const T`. This does not mean that a parameter of type `inout T` can be initialized with a argument of type `const T` because there is no valid inverted conversion system to assign back to a value of type `const T`.

conversion is ranked as **Promotion Extension**. If a scalar splat occurs in a conversion sequence with a **Conversion** conversion, the conversion is ranked as **Conversion Extension**.

4 If a vector truncation conversion occurs in a conversion sequence where all other conversions are **Exact Match** rank, the conversion is ranked as **Truncation**. If a vector truncation occurs in a conversion sequence with a **Promotion** conversion, the conversion is ranked as **Promotion Truncation**. If a vector truncation occurs in a conversion sequence with a **Conversion** conversion, the conversion is ranked as **Conversion Truncation**.

5 Otherwise, the rank of a conversion sequence is determined by considering the rank of each conversion.

6 Conversion sequence ranks are ordered from better to worse as:

1. **Exact Match**
2. **Extension**
3. **Promotion**
4. **Promotion Extension**
5. **Conversion**
6. **Conversion Extension**
7. **Truncation**
8. **Promotion Truncation**
9. **Conversion Truncation**

8.2.4.2 Comparing Implicit Conversion Sequences [Overload.ICS.Comparing]

1 A partial ordering of implicit conversion sequences exists based on defining relationships for *better conversion sequence*, and *better conversion*. If an implicit conversion sequence $ICS(f)$ is a better conversion sequence than $ICS(f`)$, then the inverse is also true: $ICS(f`)$ is a *worse conversion sequence* than $ICS(f)$. If $ICS(f)$ is neither better nor worse than $ICS(f`)$, the conversion sequences are *indistinguishable conversion sequences*.

2 A standard conversion sequence is always better than a user-defined conversion sequence.

3 Standard conversion sequences are ordered by their ranks. Two conversion sequences with the same rank are indistinguishable unless one of the following rules applies:

- If class `B` is derived directly or indirectly from class `A` and class `C` is derived directly or indirectly from class `B`,
  - binding of a expression of type `C` to a cxvalue of type `B` is better than binding an expression of type `C` to a cxvalue of type `A`,
  - conversion of `C` to `B` is better than conversion or `C` to `A`,
  - binding of a expression of type `B` to a cxvalue of type `A` is better than binding an expression of type `C` to a cxvalue of type `A`,
  - conversion of `B` to `A` is better than conversion of `C` to `A`.

9 Resource Types [Resources]

Resources are built-in intangible types representing memory with an external interface.

These take the form of Typed Buffers, Raw Buffers, Textures, Constant Buffers and Samplers.

Buffer and Texture types can be read-only or writable.

9.1 Typed Buffers [Resources.tybufs]

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 Types may have their formats converted upon load.

The element type may be any type up to 16 bytes in size. Elements may be padded to 16 bytes if the element type is smaller than 16 bytes.

All typed buffers can be read through subscript operators or Load methods. Writable typed buffers can be written through subscript operators.

Where `T` can be any arithmetic type 3.9.1 or a vector with a maximum of four elements containing such an arithmetic type. The total size of a single contained element must be less than 16 bytes.

Typed buffers perform format conversions on load such that the underlying data gets converted to the destination type.

```
template <typename T = float4>
 class Buffer {
 public:
   Buffer();
   Buffer(const Buffer &buf);
   Buffer &operator=(Buffer &buf);

   void GetDimensions(out uint width) const;

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


template <typename T>
 class RWBuffer : public Buffer {
 public:
   RWBuffer();
   RWBuffer(RWBuffer buf);
   RWBuffer &operator=(RWBuffer &buf);

   // element access
   T &operator[](int n);
};
}
```

9.1.1 Constructors [Resources.tybufs.ctrs]

Read-only Buffers can be defined with explicit or implicit template parameter types.

```
  Buffer buf1;
  Buffer<> buf2;
  Buffer<float4> buf3;
```

Since `float4` is the default type, all of these definitions are equivalent.

RWBuffers must be defined with explicit template parameter types.

```
RWBuffer<float4> buf;
```

When defined at global scope, typed buffers are bound to externally-defined backing stores using the explicit binding location if provided or the implicit binding if not (9.8).

When defined at local scope, typed buffers represent local references that can be associated with global buffers when assigned, but must be resolvable to a unique global buffer declaration.

> In the case of a buffer array declaration, a unique global declaration references all the buffers in the array. Local buffer assignments can map to different resource array elements depending on runtime values.

```
Buffer<int> grobuf;
RWBuffer<int> grwbuf;
void main() {
  Buffer<int> lrobuf = grobuf;
  RWBuffer<int> lrwbuf = grwbuf;
}
```

Buffers cannot be implicitly nor explicitly cast from one type to another. This means that local buffers can only be assigned from buffers with the same element type.

9.1.2 Dimensions [Resources.tybufs.dims]

The size of a typed buffer can be retrieved using the `GetDimensions` method.

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

This returns the full length, in elements of the buffer through the `width` out parameter.

9.1.3 Element Access [Resources.tybufs.access]

The contents of typed buffers can be retrieved using the `Load` methods or the subscript operator.

```
T Load(in int index) const;
T operator[](int index) const;
```

These each return the element of the buffer's type at the given index `index`.

An additional `Load` method returns the indicated element just as the other, but also takes a second `status` parameter that returns information about the accessed resource.

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

The `status` parameter returns an indication of whether all of the retrieved value came from fully mapped parts of the resource. This parameter must be passed to the built-in function `CheckAccessFullyMapped` (9.7) in order to interpret it.

Writable buffers have an additional subscript operator that allows assignment to an element of the buffer. It behaves as if it returns a reference to that element.

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

Partial writes aren't allowed. Assignment to individual vector elements will result in an error.

9.2 Raw Buffers [Resources.rawbufs]

Raw buffers are one-dimensional resources of arbitrary memory layout. They are either ByteAddressBuffers or StructuredBuffers. ByteAddressBuffers enable per-byte access to the raw data while StructuredBuffers have an associated structure type that determines how they are indexed. This type can be a scalar, vector, matrix, or user-defined struct.

9.3 Byte Access Buffers [Resources.babufs]

```
class ByteAddressBuffer {
public:
  ByteAddressBuffer();
  ByteAddressBuffer(const ByteAddressBuffer &buf);
  ByteAddressBuffer &operator=(ByteAddressBuffer &buf);

  void GetDimensions(out uint size) const;

  // Element access.
  uint Load(in uint offset) const;
  uint2 Load2(in uint offset) const;
  uint3 Load3(in uint offset) const;
  uint4 Load4(in uint offset) const;
  template<typename T>
  T Load(in uint offset) const;

  uint Load(in uint offset, out uint status) const;
  uint2 Load2(in uint offset, out uint status) const;
  uint3 Load3(in uint offset, out uint status) const;
  uint4 Load4(in uint offset, out uint status) const;
  template<typename T>
  T Load(in uint offset, out uint status) const;
};

class RWByteAddressBuffer : public ByteAddressBuffer {
public:
  RWByteAddressBuffer();
  RWByteAddressBuffer(const RWByteAddressBuffer &buf);
  RWByteAddressBuffer &operator=(RWByteAddressBuffer &buf);

  // Element assignment.
  void Store(in uint offset, in uint value);
  void Store2(in uint offset, in uint2 value);
  void Store3(in uint offset, in uint3 value);
  void Store4(in uint offset, in uint4 value);
  template<typename T>
  void Store(in uint offset, in T value);

  // 32-bit integer atomic arithmetic/bitwise operations.
  void InterlockedAdd(in uint offset, in uint value);
  void InterlockedAdd(in uint offset, in uint value, out uint original);
  void InterlockedAnd(in uint offset, in uint value);
  void InterlockedAnd(in uint offset, in uint value, out uint original);
  void InterlockedOr(in uint offset, in uint value);
  void InterlockedOr(in uint offset, in uint value, out uint original);
  void InterlockedXor(in uint offset, in uint value);
  void InterlockedXor(in uint offset, in uint value, out uint original);

  // 32-bit integer atomic comparison operations
  void InterlockedMin(in uint offset, in int value);
  void InterlockedMin(in uint offset, in uint value);
  void InterlockedMin(in uint offset, in int value, out int original);
  void InterlockedMin(in uint offset, in uint value, out uint original);
  void InterlockedMax(in uint offset, in int value);
  void InterlockedMax(in uint offset, in uint value);
  void InterlockedMax(in uint offset, in int value, out int original);
  void InterlockedMax(in uint offset, in uint value, out uint original);

  // 32-bit integer atomic compare/exchange operations
```

```
    void InterlockedCompareStore(in uint offset, in uint compare, in uint value);
    void InterlockedExchange(in uint offset, in uint value, out uint original);
    void InterlockedCompareExchange(in uint offset, in uint compare, in uint value,
                                    out uint original);

    // 64-bit integer atomic arithmatic/bitwise operations
    void InterlockedAdd64(in uint offset, in uint64_t value);
    void InterlockedAdd64(in uint offset, in uint64_t value, out uint64_t original);
    void InterlockedAnd64(in uint offset, in uint64_t value);
    void InterlockedAnd64(in uint offset, in uint64_t value, out uint64_t original);
    void InterlockedOr64(in uint offset, in uint64_t value);
    void InterlockedOr64(in uint offset, in uint64_t value, out uint64_t original);
    void InterlockedXor64(in uint offset, in uint64_t value);
    void InterlockedXor64(in uint offset, in uint64_t value, out uint64_t original);

    // 64-bit integer atomic comparison operations
    void InterlockedMin64(in uint offset, in int64_t value);
    void InterlockedMin64(in uint offset, in int64_t value, out int64_t original);
    void InterlockedMin64(in uint offset, in uint64_t value);
    void InterlockedMin64(in uint offset, in uint64_t value, out uint64_t original);
    void InterlockedMax64(in uint offset, in int64_t value);
    void InterlockedMax64(in uint offset, in int64_t value, out int64_t original);
    void InterlockedMax64(in uint offset, in uint64_t value);
    void InterlockedMax64(in uint offset, in uint64_t value, out uint64_t original);

    // 64-bit integer atomic compare/exchange operations
    void InterlockedCompareStore64(in uint offset, in uint64_t compare,
                                   in uint64_t value);
    void InterlockedExchange64(in uint offset, in int64_t value,
                               out int64_t original);
    void InterlockedCompareExchange64(in uint offset, in uint64_t compare,
                                      in uint64_t value, out int64_t original);

    // 32-bit float atomic compare/exchange operations
    void InterlockedCompareStoreFloatBitwise(in uint byteOffest,
                                             in float compare, in float value);
    void InterlockedExchangeFloat(in uint byteOffest, in float value,
                                  out float original);
    void InterlockedCompareExchangeFloatBitwise(in uint byteOffest,
                                                in float compare,
                                                in float value,
                                                out float original);
};
```

9.3.1 Constructors [Resources.babufs.ctrs]

Since their contents are viewed as raw bytes, ByteAddressBuffers are defined without any type parameters.

```
ByteAddressBuffer robuf;
RWByteAddressBuffer rwbuf;
```

When defined at global scope, ByteAccessBuffers are bound to externally-defined backing stores using the explicit binding location if provided or the implicit binding if not (9.8).

When defined at local scope, ByteAccessBuffers represent local references that can be associated with global ByteAccessBuffers when assigned, but must be resolvable to a unique global buffer declaration.

```
ByteAddressBuffer grobuf;
RWByteAddressBuffer grwbuf;
void main() {
  ByteAddressBuffer lrobuf = grobuf;
  RWByteAddressBuffer lrwbuf = grwbuf;
}
```

9.3.2 Dimensions [Resources.babufs.dims]

The size of a ByteAddressBuffer can be retrieved using the `GetDimensions` method.

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

This returns the full size of the buffer in bytes through the `size` out parameter.

9.3.3 Element Access [Resources.babufs.access]

The contents of ByteAddressBuffers can be retrieved using the `Load` methods.

```
uint Load(in uint offset) const;
uint2 Load2(in uint offset) const;
uint3 Load3(in uint offset) const;
uint4 Load4(in uint offset) const;
```

These each return the bytes at the given byte offset into the buffer `offset` in the form of one or more unsigned integers. The `offset` address must be a multiple of 4. Each of the variants returns a number of bytes corresponding to the size of the uint vector returned.

An additional templated load method allows returning the bytes at the given byte offset in the form of the type given in the template parameter. This can be a scalar, vector, matrix, or user-defined struct.

```
template<typename T>
T Load(in uint offset) const;
```

The alignment requirements of `offset` for this operation is the alignment requirement of an object of type `T` in the device memory space (1.7.2).

Additional `Load` methods return the same values as the others, but also take a second `status` parameter that returns information about the accessed resource.

```
uint Load(in uint offset, out uint status) const;
uint2 Load2(in uint offset, out uint status) const;
uint3 Load3(in uint offset, out uint status) const;
uint4 Load4(in uint offset, out uint status) const;
template<typename T>
T Load(in uint offset, out uint status) const;
```

The `status` parameter returns an indication of whether all of the retrieved value came from fully mapped parts of the resource. This parameter must be passed to the built-in function `CheckAccessFullyMapped` (9.7) in order to interpret it.

9.3.4 Atomic Operations [Resources.babufs.atomics]

RWByteAddressBuffers have a suite of atomic methods that perform the given operation on type-appropriate-sized element data at the given buffer offset in a way that ensures no contention from other threads performing other operations. Each of these operates on two 32 or 64 bit values: the given parameter value and the buffer value. The buffer value is a 32 or 64 bit value at a given offset into the buffer. Each method has an overload that returns the original buffer value before the atomic operation was performed.

```
void InterlockedAdd(in uint offset, in uint value);
void InterlockedAdd(in uint offset, in uint value, out uint original);
void InterlockedAnd(in uint offset, in uint value);
void InterlockedAnd(in uint offset, in uint value, out uint original);
void InterlockedOr(in uint offset, in uint value);
void InterlockedOr(in uint offset, in uint value, out uint original);
void InterlockedXor(in uint offset, in uint value);
void InterlockedXor(in uint offset, in uint value, out uint original);
```

Perform atomic addition, bitwise and, bitwise or, or bitwise exclusive or on the 32-bit integer buffer value at the byte offset `offset` and the provided `value`. Store the result to that offset buffer location, optionally returning the original buffer value through `original`.

```
void InterlockedMin(in uint offset, in int value);
void InterlockedMin(in uint offset, in uint value);
void InterlockedMin(in uint offset, in int value, out int original);
void InterlockedMin(in uint offset, in uint value, out uint original);
void InterlockedMax(in uint offset, in int value);
void InterlockedMax(in uint offset, in uint value);
void InterlockedMax(in uint offset, in int value, out int original);
void InterlockedMax(in uint offset, in uint value, out uint original);
```

Perform the sign-approrpriate minimum or maximum comparison on the 32-bit integer buffer value at the byte offset `offset` and the provided `value`. Store the lesser or greater value, as appropriate, to that offset buffer location, optionally returning the original buffer value through `original`.

```
void InterlockedExchange(in uint offset, in uint value, out uint original);
```

Performs an atomic exchange of the 32-bit integer buffer value at the byte offset `offset` with the provided `value`. Stores `value` at the offset buffer location and returns the original buffer value through `original`.

```
void InterlockedCompareStore(in uint offset, in uint compare, in uint value);
void InterlockedCompareExchange(in uint offset, in uint compare, in uint value,
                                out uint original);
```

Perform an atomic exchange of or store to the 32-bit integer buffer value at the byte offset `offset` with the provided `value` if the value at that location matches the value of `compare`. `InterlockedCompareExchange` returns the original buffer value through `original`.

```
void InterlockedAdd64(in uint offset, in uint64_t value);
void InterlockedAdd64(in uint offset, in uint64_t value, out uint64_t original);
void InterlockedAnd64(in uint offset, in uint64_t value);
void InterlockedAnd64(in uint offset, in uint64_t value, out uint64_t original);
void InterlockedOr64(in uint offset, in uint64_t value);
void InterlockedOr64(in uint offset, in uint64_t value, out uint64_t original);
void InterlockedXor64(in uint offset, in uint64_t value);
void InterlockedXor64(in uint offset, in uint64_t value, out uint64_t original);
```

Perform atomic addition, bitwise and, bitwise or, or bitwise exclusive or on the 64-bit integer buffer value at the byte offset `offset` and the provided `value`. Store the result to that offset buffer location, optionally returning the original buffer value through `original`.

```
void InterlockedMin64(in uint offset, in int64_t value);
void InterlockedMin64(in uint offset, in uint64_t value);
void InterlockedMin64(in uint offset, in int64_t value, out int64_t original);
void InterlockedMin64(in uint offset, in uint64_t value, out uint64_t original);
void InterlockedMax64(in uint offset, in int64_t value);
void InterlockedMax64(in uint offset, in uint64_t value);
void InterlockedMax64(in uint offset, in int64_t value, out int64_t original);
void InterlockedMax64(in uint offset, in uint64_t value, out uint64_t original);
```

Perform the sign-approrpriate minimum or maximum comparison on the 64-bit integer buffer value at the byte offset `offset` and the provided `value`. Store the lesser or greater value, as appropriate, to that offset buffer location, optionally returning the original buffer value through `original`.

```
void InterlockedExchange64(in uint offset, in uint64_t value, out uint64_t original);
```

Performs an atomic exchange of the 64-bit integer buffer value at the byte offset `offset` with the provided `value`. Stores `value` at the offset buffer location and returns the original buffer value through `original`.

```
void InterlockedCompareStore64(in uint offset, in uint64_t compare,
                               in uint64_t value);
void InterlockedCompareExchange64(in uint offset, in uint64_t compare,
                                  in uint64_t value, out uint64_t original);
```

Perform an atomic exchange of or store to the 64-bit integer buffer value at the byte offset `offset` with the provided `value` if the value at that location matches the value of `compare`. `InterlockedCompareExchange` returns the original buffer value through `original`.

```
void InterlockedExchangeFloat(in uint byteOffest, in float value,
                              out float original);
```

Performs an atomic exchange of the 32-bit float buffer value at the byte offset `offset` with the provided `value`. Stores `value` at the offset buffer location and returns the original buffer value through `original`.

```
void InterlockedCompareStoreFloatBitwise(in uint byteOffest, in float compare,
                                         in float value);
void InterlockedCompareExchangeFloatBitwise(in uint byteOffest,
                                            in float compare,
                                            in float value,
                                            out float original);
```

Perform an atomic exchange of or store to the 32-bit float buffer value at the byte offset `offset` with the provided `value` if the value at that location matches the value of `compare` in a bitwise comparison. `InterlockedCompareExchangeFloatBitwise` returns the original buffer value through `original`.

Unlike standard floating point comparisons, values will only match if the bit representation matches which can miss some matching values that have different bit repesantations in addition to ignoring floating point rules surrounding NaN and infinite values.

9.4 Structured Buffers [Resources.stbufs]

Structured buffers are raw buffers with associated types that facilitate indexing. Unlike typed buffers, they perform no format conversions. The buffer contents is treated as raw data. Accessors return the contained bytes as the target type with no conversions or interpolations applied. Structured buffers can be defined with scalar, vector, matrix, or user-defined struct elements.

Structured buffers can be read-only, writable, appendable, or consumable. Writable buffers can have their elements assigned in arbitrary locations. Append structured buffers can only have elements added to the end. Consume structured buffers can only have elements removed from the end.

```
template <typename T>
 class StructuredBuffer {
 public:
   StructuredBuffer();
   StructuredBuffer(const StructuredBuffer &buf);
   StructuredBuffer &operator=(StructuredBuffer &buf);

   void GetDimensions(out uint count, out uint stride) const;

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

template <typename T>
 class RWStructuredBuffer : public StructuredBuffer {
 public:
   RWStructuredBuffer();
   RWStructuredBuffer(const RWStructuredBuffer &buf);
   RWStructuredBuffer &operator=(RWStructuredBuffer &buf);

   // Element assignment.
   T &operator[](int index);
```

```
    // Hidden counter increment/decrement.
    uint IncrementCounter();
    uint DecrementCounter();
};

template <typename T>
 class AppendStructuredBuffer {
 public:
    AppendStructuredBuffer();
    AppendStructuredBuffer(const AppendStructuredBuffer &buf);
    AppendStructuredBuffer &operator=(AppendStructuredBuffer &buf);

    void GetDimensions(out uint count, out uint stride) const;

    void Append(in T value);
};

template <typename T>
 class ConsumeStructuredBuffer {
 public:
    ConsumeStructuredBuffer();
    ConsumeStructuredBuffer(const ConsumeStructuredBuffer &buf);
    ConsumeStructuredBuffer &operator=(ConsumeStructuredBuffer &buf);

    void GetDimensions(out uint count, out uint stride) const;

    T Consume();
};
```

9.4.1 Constructors [Resources.stbufs.ctrs]

Structured buffers must be defined with explicit template parameter types.

```
struct S {int i; float f;};
StructuredBuffer<S> robuf1;
StructuredBuffer<float4> robuf2;
StructuredBuffer<float2x3> robuf3;

RWStructuredBuffer<S> rwbuf1;
RWStructuredBuffer<float4> rwbuf2;
RWStructuredBuffer<float2x3> rwbuf3;

AppendStructuredBuffer<S> apbuf1;
AppendStructuredBuffer<float4> apbuf2;
AppendStructuredBuffer<float2x3> apbuf3;

ConsumeStructuredBuffer<S> cobuf1;
ConsumeStructuredBuffer<float4> cobuf2;
ConsumeStructuredBuffer<float2x3> cobuf3;
```

When defined at global scope, structured buffers are bound to externally-defined backing stores using the explicit binding location if provided or the implicit binding if not (9.8).

When defined at local scope, structured buffers represent local references that can be associated with global buffers when assigned, but must be resolvable to a unique global buffer declaration.

> Resource types contained in structs are only allocated registers when they are explicitly used. This includes elements of arrays of resources as such array elements must be indexed with literals.

```
StructuredBuffer<int3x3> grobuf;
```

```
RWStructuredBuffer<int3x3> grwbuf;
AppendStructuredBuffer<int3x3> gapbuf;
ConsumeStructuredBuffer<int3x3> gcobuf;
void main() {
  StructuredBuffer<int3x3> lrobuf = grobuf;
  RWStructuredBuffer<int3x3> lrwbuf = grwbuf;
  AppendStructuredBuffer<int3x3> lbuf = gapbuf;
  ConsumeStructuredBuffer<int3x3> lcobuf = gcobuf;
}
```

Structured buffer operands to the assignment operator must have the same element type.

9.4.2 Dimensions [Resources.stbufs.dims]

The structure count and stride of a structured buffer can be retrieved using the `GetDimensions` method.

```
void GetDimensions(out uint count, out uint stride);
```

This returns number of structured elements of the buffer through the `count` out parameter and the size of each element in bytes through the `stride` out parameter.

9.4.3 Element Access [Resources.stbufs.access]

The contents of read-only StructuredBuffers and writable RWStructuredBuffers can be retrieved using the `Load` methods or the subscript operator.

```
T Load(in int index) const;
T operator[](int index) const;
```

These each return the element of the buffer's type at the given index `index`.

An additional `Load` method returns the indicated element just as the other, but also takes a second `status` parameter that returns information about the accessed resource.

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

The `status` parameter returns an indication of whether all of the retrieved value came from fully mapped parts of the resource. This parameter must be passed to the built-in function `CheckAccessFullyMapped` (9.7) in order to interpret it.

Writable RWStructuredBuffers have an additional subscript operator that allows assignment to a structure element of the buffer. It behaves as if it returns a reference to that element.

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

9.4.4 Counter Manipulation [Resources.stbufs.counter]

RWStructuredBuffers may have a hidden counter that can be incremented and decremented using methods.

```
uint IncrementCounter();
uint DecrementCounter();
```

Increment or decrements the hidden counter associated with the RWStructuredBuffer. `IncrementCounter` returns the pre-increment value of the counter. `DecrementCounter` returns the post-decrement value of the counter.

9.4.5 Append [Resources.stbufs.append]

AppendStructuredBuffers can only be appended to as an output stream.

```
void Append(in T value);
```

Appends the `value` of the template parameter types to the end of the AppendStructuredBuffer. Subsequent appends will continue to add elements to the end of the buffer.

9.4.6 Consume [Resources.cnstbufs.consume]

ConsumeStructuredBuffers can only have values pulled from them as an input stream.

```
T Consume();
```

Removes and returns a value of the template parameter types from the end of the ConsumeStructuredBuffer.

9.5 Constant Buffers [Resources.cnbuf]

1 Constant buffers represent resources that contain read-only constant data in a well-defined memory layout. A constant buffer is declared either as a cbuffer-declaration (7.8) or as a declaration of Constant Buffer Class type (9.5.1).

9.5.1 Constant Buffer Class [Resources.cnbuf.cbclass]

1 Another way of declaring constant buffers is by using the `ConstantBuffer<T>` resource class.

2 The template parameter `T` must be a class type (**??**).

```
template <typename T>
 class ConstantBuffer {
 public:
   ConstantBuffer();
   ConstantBuffer(const ConstantBuffer &buf);
   ConstantBuffer &operator=(ConstantBuffer &buf);
};
```

3 The *shader constants* are the fields of the class type `T`. They can be referenced from anywhere within the translation unit after the `ConstantBuffer<T>` declaration as if they were fields of the `ConstantBuffer<T>` class.

4 For example:

```
struct MyConstants {
  float4 CameraPos;
};

ConstantBuffer<MyConstants> CB;

float4 getCameraPosition() {
  return CB.CameraPos;
}
```

5 The layout rules for constant buffer declared with the `ConstantBuffer<T>` syntax are the same as for named `cbuffer` declaration groups.

6 `ConstantBuffer<T>` can be defined at local scope to represent a local reference to a constant buffer that can be associated with a global buffer when assigned. The reference must be resolvable to a unique global buffer declaration before use.

7 Constant buffers cannot be implicitly nor explicitly cast from one type to another. This means that local `ConstantBuffer` can only be assigned from `ConstantBuffer` with the same template type `T`.

9.5.2 Constant Buffer Layout [Resources.cnbuf.lay]

1 Layout of the constant buffer is implementation dependent. This section describes the layout that is used by DirectX.

2 A constant buffer is arranged like an array of 16-byte rows, or 4-component vectors of 32-bit elements. Shader constants are arranged into the buffer in the order they were declared based on following rules.

3 Specific basic types are packed into the last used row of the buffer if they fit into the available space, starting at the next available aligned position. These types include:

- *scalar types*

- *vector types*
- *matrix types* with only one row in *column_major* storage layout

4 If they do not fit the remaining space of the row, they will placed at the start a new 16-byte aligned row.

5 Vectors are aligned by the size of a single vector element type unless that alignment results in crossing the 16-byte row boundary, in which case it is aligned to the next row.

6 Aggregate types like arrays and structures are always 16-byte row-aligned. If the aggregate ends with an element that does not completely fill a row, the remaining space on the last row may be used by the next value.

7 Individual array elements are always 16-byte row aligned.

8 Matrix types with more than one row in *column_major* storage layout and matrix types in *row_major* storage layout in are aligned to the 16-byte row. Each row in storage layout (each column for *column_major* matrix) is aligned to a 16-byte row.

9 Members of structured types used in constant buffers follow these same rules. Because structures are always 16-byte row-aligned, the offset layout within the structure is independent of the particular structure instance location.

9.5.3 Packoffset annotations [Resources.cnbuf.po]

1 Shader constants declared in `cbuffer` declaration group can have an optional `packoffset` annotation added before their `;` terminal. If this annotation is added to one shader constant in a declaration group then it must be added to all shader constants in that group.

packoffset-annotation:
`:` `packoffset(` *packoffset-id packoffset-row packoffset-element*$_{opt}$ `)`

packoffset-id:one of
`c C`

packoffset-row:
digit
packoffset-row digit

packoffset-element:
`.` *packoffset-element-id*

packoffset-element-id:one of
`x y z w r g b a`

2 The `packoffset` annotation defines specific offset in the constant buffer where the shader constant is located.

3 The *packoffset-row* number is the index into the array of 16-byte rows of the constant buffer. Each row treated as a vector of four 32-bit elements.

4 The *packoffset-element* identifies specific location within the vector where `x` or `r` is the first element of the vector, `y` or `g` is the second one, `z` or `b` is the third one and `w` or `a` is the fourth, and last element of the vector.

5 Shader constant that has a structure, array or matrix type must always be 16-byte row-aligned. It means that if it specifies a *packoffset-element*, it must have a value of `x` or `r`.

6 For example

```
cbuffer MyConstants {
  float2 Pos : packoffset(c1.z);
};
```

The `Pos` shader constant will be located at byte offset 24 in the constant buffer `MyConstants`.

9.5.4 Default Constant Buffer [Resources.cnbuf.defcb]

1 All variables declarations in the global scope are implicitly added to default constant buffer called `$Globals`, unless they are marked `static`, `groupshared`, or declare a resource or array of resources.

2 Layout rules for a default constant buffer are the same as for a named `cbuffer` declaration group.

3 Default constant buffer declarations can have an optional *resource-binding* annotation with a *register-type* `c` or `C` added before their `;` terminal. This annotation does not define a virtual register binding but rather an index into an array of 16-byte rows that represents the default constant buffer, similar to the `packoffset` annotation.

4 For example

```
float4 CameraPos : register(c2);
```

is equivalent to

```
cbuffer CB {
  float4 CameraPos : packoffset(c2);
}
```

9.6 Samplers [Resources.samp]

9.7 CheckAccessFullyMapped [Resources.mapcheck]

The mapped status value returned by certain texture and buffer methods can be intrepreted by a built-in function:

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

This function returns true if the value was accessed from a fully committed resource, or from fully mapped pages of a sparse resource. If any part of the return value was from unmapped memory, false is returned.

9.8 Resource Binding [Resources.binding]

Resources are bound to external memory using virtual registers within logical registers spaces. These can be specified explicitly using attributes after the global resource declaration or implicitly by leaving them off.

resource-binding:
: `register(` *register-type bind-number* `)`
: `register(` *register-type bind-number* `, space` *bind-number* `)`

register-type: one of
`t u b s c T U B S C`

bind-number:
digit
bind-number digit

The register type indicates whether the binding is for a read-only (`t`/`T`), a writable (`u`/`U`), a constant buffer (`b`/`B`), or a sampler (`s`/`S`). The register bind number indicates the register number at which the resource variable begins. The optional second parameter indicates the virtual register space that the register belongs to.

Register type `c`/`C` defines layout information for the default constant buffer (9.5.4).

Sample syntax of virtual register binding attributes:

```
Buffer<float> robuf : register(t0, space0);
RWBuffer<float> rwbuf : register(u0, space0);
SamplerState samp : register(s0, space0);
cbuffer cbuf : register(b0, space0) { float f; };
```

Each resource must have a unique register. Different resources cannot occupy the same register, but read-only and writable buffers have separate namespaces and different logical register spaces have independent sets of register numbers. Meaning the following is all allowed:

```
Buffer<float> mybuf : register(t0, space0);
Buffer<float> yourbuf : register(t0, space1);
RWBuffer<float> hisbuf : register(u0, space0);
RWBuffer<float> herbuf : register(u0, space1);
```

An aggregate resource type can be:

- a struct containing one or more resources types
- an array of resource types
- an array or struct containing either of the above

Aggregate resource types with register annotations occupy the first register they specify and however many additional sequential registers the aggregate requires to assign a register to each of its elements that correspond to the given register type. Multiple register annotations can be placed on aggregates involving structs where each corresponds to a distinct register type that will apply to all the registers of that type in the aggregate.

> Resource types contained in structs are only allocated registers when they are explicitly used. This includes elements of arrays of resources since such array elements must be indexed with literals.

```
// Occupies t0, t1, and t2.
Buffer<float> fbuf[3] : register(t0, space0);
// Occupies registers t3 - t14.
Buffer<int4> ibuf[4][3] : register(t3, space0);
// Occupies registers t15 and u0.
struct {RWBuffer<int> rwbuf; Buffer buf;} sbufs : register(t15) : register(u0);
// Occupies registers t16 - t21.
struct {Buffer bufs[5]; Buffer buf;} robufs : register(t16);
// Occupies registers t22-24 and u1-u2
struct {Buffer r; RWBuffer<int> w;} bibufs[2] : register(t22) : register(u1);
```

If the register binding or register space is not specified, implicit values are used. Whenever not specified, the space defaults to space0.

```
Buffer<float2> buf1 : register(t0, space0);
Buffer<float2> buf2 : register(t1); // defaults to space0
```

When the register is not specified, resources will recieve implementation-dependent register assignments.

10 Templates [Template]

10.1 Template Instantiation [Template.Inst]

10.2 Partial Ordering of Function Templates [Template.Func.Order]

11 Intangible Types [Intangible]

12 Runtime [Runtime]

Acronyms

API Application Programming Interface. 59

C C Programming Language. 4

C++ C++ Programming Language. 4

DXC DirectX Shader Compiler. 4, 14

FXC Legacy DirectX Shader Compiler. 4, 14

HLSL High Level Shader Language. 1, 4–9, 22

SIMD Single Instruction Multiple Data. 6

SIMT Single Instruction Multiple Thread. 6

SPMD Single Program Multiple Data. 1, 5–7, 59

Glossary

DirectX DirectX is the multimedia API introduced with Windows 95.. 4, 5, 8

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.. 6, 7

IEEE Standard 754 IEEE Standard For Floating Point Arithmetic. 20

ISO C standard (2011) ISO/IEC 9899:2011: Standard for Programming Language C.. 4, 9, 13

ISO C standard (2023) ISO/IEC 9899:2023: Standard for Programming Language C.. 20

ISO C++ standard (2011) ISO/IEC 14882:2011: Standard for Programming Language C++.. 4, 9, 15, 16, 20, 22, 37, 38

Lane The computation performed on a single element as described in the SPMD program. Also called: thread.. 6–8, 59

Quad A group of four Lanes which form a cluster of adjacent computations in the data topology. Also called: quad-group or quad-wave. . 6, 7

Shader Model Versioned hardware description included as part of the DirectX specification, which is used for code generation to a common set of features across a range of vendors.. 5, 7, 8

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.. 6–8, 59

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.. 6, 7, 59