SPIR 1.0 Specification for OpenCL

Khronos Group - OpenCL Working Group - SPIR subgroup

2012-08-24

Abstract

This document defines version 1.0 of the Standard Portable Intermediate Representation (SPIR) for OpenCL™. [1]

The Khronos Group Inc. ratified this document as a provisional specification on August 24, 2012.

Contents

[1] OpenCL and the OpenCL logo are trademarks of Apple Inc.

List of Tables

Copyright (c) 2011-2012 The Khronos Group Inc. All Rights Reserved.

This specification is protected by copyright laws and contains material proprietary to the Khronos Group, Inc. It or any components may not be reproduced, republished, distributed, transmitted, displayed, broadcast or otherwise exploited in any manner without the express prior written permission of Khronos Group. You may use this specification for implementing the functionality therein, without altering or removing any trademark, copyright or other notice from the specification, but the receipt or possession of this specification does not convey any rights to reproduce, disclose, or distribute its contents, or to manufacture, use, or sell anything that it may describe, in whole or in part.

Khronos Group grants express permission to any current Promoter, Contributor or Adopter member of Khronos to copy and redistribute UNMODIFIED versions of this specification in any fashion, provided that NO CHARGE is made for the specification and the latest available update of the specification for any version of the API is used whenever possible. Such distributed specification may be re-formatted AS LONG AS the contents of the specification are not changed in any way. The specification may be incorporated into a product that is sold as long as such product includes significant independent work developed by the seller. A link to the current version of this specification on the Khronos Group web-site should be included whenever possible with specification distributions.

Khronos Group makes no, and expressly disclaims any, representations or warranties, express or implied, regarding this specification, including, without limitation, any implied warranties of merchantability or fitness for a particular purpose or non-infringement of any intellectual property. Khronos Group makes no, and expressly disclaims any, warranties, express or implied, regarding the correctness, accuracy, completeness, timeliness, and reliability of the specification. Under no circumstances will the Khronos Group, or any of its Promoters, Contributors or Members or their respective partners, officers, directors, employees, agents or representatives be liable for any damages, whether direct, indirect, special or consequential damages for lost revenues, lost profits, or otherwise, arising from or in connection with these materials.

Khronos, StreamInput, WebGL, COLLADA, OpenKODE, OpenVG, OpenWF, OpenSL ES, OpenMAX, OpenMAX AL, OpenMAX IL and OpenMAX DL are trademarks and WebCL is a certification mark of the Khronos Group Inc. OpenCL is a trademark of Apple Inc. and OpenGL and OpenML are registered trademarks and the OpenGL ES and OpenGL SC logos are trademarks of Silicon Graphics International used under license by Khronos. All other product names, trademarks, and/or company names are used solely for identification and belong to their respective owners.

Acknowledgements

Editor: Boaz Ouriel, Intel

Contributors:

- David Neto, Altera
- Anton Lokhmotov, ARM
- Mike Houston, AMD
- Micah Villmow, AMD
- Tanya Lattner, Apple
- Aaftab Munshi, Apple
- Holger Waechtler, Broadcom
- Andrew Richards, Codeplay
- Guy Benyei, Intel
- Javier E. Martinez, Intel
- Vinod Grover, NVIDIA
- Kedar Patil, NVIDIA
- Sumesh Udayakumaran, QUALCOMM
- Chihong Zhang, QUALCOMM
- Henry Styles, Xilinx

1 Introduction

This document defines version 1.0 of the OpenCL Standard Portable Intermediate Representation (SPIR). SPIR is a mapping from the OpenCL C programming language into LLVM IR.

This version of the specification is based on LLVM 3.1 [4] [3], and on OpenCL C as specified in the OpenCL 1.2 Specification [2].

The goal of SPIR is to provide a portable interchange format for partly compiled OpenCL C programs. The format:

- Is vendor neutral.
- Is not C source code.
- Supports almost all core features and KHR extensions for version 1.2 of OpenCL C. (A small number of features of OpenCL C are not expressible in SPIR.)
- Is designed to support vendor extensions.
- Is compact.
- Is designed to be efficiently loaded by an OpenCL implementation.
- Supports targeting devices with either a 32- or 64-bit address space with a single SPIR IR instance of a program. (To achieve this, the program must already be 32- and 64-bit portable.)
- Is designed to be useful as a target format for compilers of programming languages other than OpenCL C. This is a secondary goal of SPIR.

1.1 One format, two notations

LLVM IR has three semantically equivalent representations:

- An in-memory data structure manipulated by the LLVM software.
- A compact external binary representation, known as *bitcode* [3]. [2]
- A human readable assembly language notation [4].

SPIR adopts two of these: the bitcode and assembly language notations from LLVM. For ease of exposition, the remainder of this document uses only the assembly language notation.

1.2 Name mangling

In SPIR, the names of many standard functions are mangled to encode type information. Mangling names disambiguates between different source language identifiers with the same root name, i.e. when a function is overloaded on its argument types. Mangling enables standard linkers and loaders to select the unique implementation of an overloaded function, and permits interface type checking across module boundaries.

For example, OpenCL C has many overloaded built-in functions, meaning the same function name is used with different argument and return types. For example, the `sin` built-in function is defined for both scalar and vector floating point argument and return types. SPIR distinguishes between all of variations of the `sin` function by mangling the root name `sin` with its argument types.

Specifically, the following categories of function names are mangled in SPIR:

[2] The LLVM 3.1 bitcode notation is only partly documented by [3]. However, bitcode notation is fully (but implicitly) defined by the behaviour of LLVM 3.1 software release.

- Functions mentioned in this specification.
- OpenCL C built-in functions that are overloaded on their argument types.

Other kinds of names are not mangled in SPIR. In particular, regular and kernel user functions from OpenCL C are not mangled when mapped into SPIR.

By *not* mangling the names of regular functions, SPIR supports being the target for language families (other than C/C++) having their own distinctive type systems. In other words, mangling of user-level functions is beyond the scope of SPIR, and is subject to coordination among third parties (compiler front end and library implementors).

For names that do require mangling, SPIR adopts and extends the name mangling scheme from Section 5.1 of the Itanium C++ ABI [1]. Extensions are required to support OpenCL concepts absent from ordinary C++. The SPIR mangling scheme is defined in Appendix A.

Note: For readability, this document uses the *unmangled* names for all identifiers. However actual SPIR instances always use the mangled form for names in the categories as specified above.

2 OpenCL C mapping to SPIR

2.1 Supported Data Types

The following LLVM data types are supported:

2.1.1 Built-in Scalar Data Types

Table 1 describes the mapping from the OpenCL C built-in scalar data types to SPIR built-in scalar data types.

OpenCL C Type	**LLVM Type**
bool	i1
char	i8
unsigned char, uchar	i8
short	i16
unsigned short, ushort	i16
int	i32
unsigned int, uint	i32
long	i64
unsigned long, ulong	i64
float	float
double	double
half	half
void	void

Table 1: Mapping for built-in scalar data types

Notes:

- Signed and unsigned values are sign extended or zero extended based on the deployed operation.
- While LLVM has many more primitive data types, only the ones described above are allowed in SPIR.

2.1.1.1 The `size_t` data type

Table 2 describes the mapping of OpenCL C `size_t`, `ptrdiff_t`, `intptr_t` and `uintptr_t` types to SPIR types.

OpenCL C Type	LLVM Type	LLVM Name
size_t	opaque structure type	%spir.size_t
ptrdiff_t	opaque structure type	%spir.size_t
intptr_t	opaque structure type	%spir.size_t
uintptr_t	opaque structure type	%spir.size_t

Table 2: Mapping of `size_t` related types

The `%spir.size_t` data type represents an integer type of either `i32` or `i64` depending on the address space of the target device. In addition to the new data type, SPIR adds a set of functions that handle operations using this data type.

Note: SPIR does not provide a way to encode direct arithmetical operations between:

- %spir.size_t and i32
- %spir.size_t and i64

Such an expression would be the obvious SPIR encoding of an OpenCL C arithemtic operation between:

- A size_t or uintptr_t operand, and a uint or ulong operand
- A ptrdiff_t or intptr_t operand, and a int or long operand

However, such OpenCL C expressions are not portable between devices with 32- and 64-bit address spaces.

2.1.1.1.1 Conversion functions

Table 3 describes the list of `%spir.size_t` conversion functions to and from SPIR scalar datatypes. We use the generic type name *gentype* to indicate `i1`,`i8`,`i16`,`i32`,`i64`, `half`, `float`, `double`

Unmangled Name	Ret Value	Arguments	Description
__spir_sizet_convert_size_t	%spir.size_t	gentype	Convert a scalar gentype type to `spir.size_t`
__spir_sizet_convert_size_t	%spir.size_t	addrspace(A) i8*	Convert a pointer to an i8 to a `spir.size_t` where A denotes the address space of the pointer.
__spir_sizet_convert_*gentype*	gentype	%spir.size_t	Convert a `spir.size_t` to a scalar gentype type
__spir_sizet_convert_ptr$A$	addrspace(A) i8*	%spir.size_t	Convert a a `spir.size_t` to a pointer to an i8 where A denotes the address space of the pointer.

Table 3: size_t conversion functions

2.1.1.1.2 Arithmetic functions

Table 4 describes `%spir.size_t` arithmetic functions:

Unmangled Name	**Ret Value**	**Arguments**	**Description**
__spir_sizet_add	%spir.size_t	%spir.size_t, %spir.size_t	Addition of two %spir.size_t values. The result is a %spir.size_t
__spir_sizet_sub	%spir.size_t	%spir.size_t a, %spir.size_t b	Subtraction of two `spir.size_t` values. The result is `a - b` which is of `spir.size_t` type
__spir_sizet_mul	%spir.size_t	%spir.size_t a, %spir.size_t b	Multiplication of two `spir.size_t` values. The result is `a * b` which is of `spir.size_t` type
__spir_sizet_div	%spir.size_t	%spir.size_t a, %spir.size_t b	Calculate the integer division of two `spir.size_t` values. The result is `a / b` which is of `spir.size_t` type
__spir_sizet_rem	%spir.size_t	%spir.size_t a, %spir.size_t b	Calculate the (unsigned) remainder of the division of two `spir.size_t` values. The result is `a % b` which is of `spir.size_t` type

Table 4: size_t arithmetic functions

Note that signed and unsigned versions of __spir_sizet_div and __spir_sizet_rem functions are distinguished by the mangling process. As described in Section A.1, source language types `size_t` and `uintptr_t` are mangled to specifier string `u2sz`, while source language types `ptrdiff_t` and `intptr_t` are mangled to specifier string `u2pd`.

2.1.1.1.3 Bitwise functions

Table 5 describes `%spir.size_t` bitwise functions:

Unmangled Name	**Ret Value**	**Arguments**	**Description**
__spir_sizet_or	%spir.size_t	%spir.size_t a, %spir.size_t b	Bitwise OR of two `spir.size_t` values. The result is a `spir.size_t`
__spir_sizet_and	%spir.size_t	%spir.size_t a, %spir.size_t b	Bitwise AND of two `spir.size_t` values. The result is a `spir.size_t`
__spir_sizet_xor	%spir.size_t	%spir.size_t a, %spir.size_t b	Bitwise XOR of two `spir.size_t` values. The result is a `spir.size_t`
__spir_sizet_not	%spir.size_t	%spir.size_t a	Bitwise NOT of a `spir.size_t` value. The result is a `spir.size_t`

Table 5: size_t bitwise functions

2.1.1.1.4 Shift functions

Table 6 describes `%spir.size_t` shift functions:

Unmangled Name	**Ret Value**	**Arguments**	**Description**
__spir_sizet_shl	%spir.size_t	%spir.size_t a, %spir.size_t b	Logical shift left: The `spir.size_t` value obtained by shifting `a` left by `b` bit positions, with zero fill.
__spir_sizet_lshr	%spir.size_t	%spir.size_t a, %spir.size_t b	Logical shift right: The `spir.size_t` value obtained by shifting `a` right by `b` bit positions, with zero fill.
__spir_sizet_ashr	%spir.size_t	%spir.size_t a, %spir.size_t b	The `spir.size_t` value obtained by shifting `a` right by `b` bit positions, where vacated positions are filled with the most significant bit of the original value of `a`. (Arithmetic shift right, treating `a` as if it were a signed value.)

Table 6: size_t shift functions

2.1.1.1.5 Relational functions

Table 7 describes `%spir.size_t` relational functions:

Unmangled Name	**Ret Value**	**Arguments**	**Description**
__spir_sizet_cmp	i1	%spir.size_t, %spir.size_t, i32	Comparison of two `spir.size_t` values. The last i32 argument is used to decide what kind of comparison is performed and it has the same meaning as the enumeration `LLVM::ICmpInst::Predicate` ICMP_EQ = 32 (equal) ICMP_NE = 33 (not equal) ICMP_UGT = 34 (unsigned greater than) ICMP_UGE = 35 (unsigned greater or equal) ICMP_ULT = 36 (unsigned less than) ICMP_ULE = 37 (unsigned less or equal) ICMP_SGT = 38 (signed greater than) ICMP_SGE = 39 (signed greater or equal) ICMP_SLT = 40 (signed less than) ICMP_SLE = 41 (signed less or equal) The return type is a boolean i1, indicating the result true or false.

Table 7: size_t relational functions

2.1.1.1.6 Pointer Arithmetics functions

Table 8 describes pointer arithmetics functions using `%spir.size_t`:

Unmangled Name	**Ret Value**	**Arguments**	**Description**
__spir_sizet_add_ptr$A$	addrspace(A) i8*	addrspace(A) i8* base, %spir.size_t off	Add a size_t integer to any pointer type where A denotes the address space of the pointer. The resulting pointer is equal to base + off.

Table 8: size_t pointer arithmetics functions

2.1.1.1.7 Memory Access functions

Table 9 describes memory access functions:

Unmangled Name	**Ret Value**	**Arguments**	**Description**
__spir_sizet_alloca	spir.size_t*	i32 NumElements, i32 alignment	This function allocates `spir.size_t` memory on the stack frame of the currently executing function, to be automatically released when this function returns to its caller values. The **NumElements** constant value indicates the number of `spit.size_t` elements to be allocated. The constant **alignment** value is optional. If specified, the value result of the allocation is guaranteed to be aligned to at least that bounday. If not specified, or if zero, the target can choose to align the allocation on any convenient bounday compatible with size_t. The return value is a `spit.size_t*`

Table 9: size_t memory access functions

2.1.1.1.8 Device Specific functions

Table 10 describes device specific functions:

Unmangled Name	**Ret Value**	**Arguments**	**Description**
__spir_size_of_sizet	spir.size_t		Constant size in bytes of the `spir.size_t` type in the current device
__spir_size_of_pointer	spir.size_t		Constant size in bytes of a pointer on the current device

Table 10: size_t device specific constant functions

2.1.2 Built-in Vector Types

Table 11 describes the mapping from the OpenCL C built-in vector data types to SPIR built-in scalar data types. Supported values of n are 2, 3, 4, 8, and 16 for all vector data types.

OpenCL C Type	**LLVM Type**
char$n$	< n x i8 >
uchar$n$	< n x i8 >
short$n$	< n x i16 >
ushort$n$	< n x i16 >
int$n$	< n x i32 >
uint$n$	< n x i32 >
long$n$	< n x i64 >
ulong$n$	< n x i64 >
half$n$	< n x half >
float$n$	< n x float >
double$n$	< n x double >

Table 11: Mapping for built-in vector types

Note: LLVM supports many more vector data types, however only the ones described above are allowed in SPIR. Specifically, a vector of `i1`'s is disallowed in SPIR.

2.1.3 Other Built-in Data Types

The OpenCL image, sampler, and event data types are mapped to LLVM named `opaque` types, as defined in Table 12. These names are reserved for SPIR and shall not be used otherwise.

OpenCL C Type	**LLVM Type**	**LLVM Name**
image1d_t	opaque	%spir.image1d_t
image1d_array_t	opaque	%spir.image1d_array_t
image1d_buffer_t	opaque	%spir.image1d_buffer_t
image2d_t	opaque	%spir.image2d_t
image2d_array_t	opaque	%spir.image2d_array_t
image3d_t	opaque	%spir.image3d_t
sampler_t	opaque	%spir.sampler_t
event_t	opaque	%spir.event_t

Table 12: Mapping for other built-in data types

2.1.3.1 Declaring samplers as global constants

Each spir.sampler_t variable is a constant-qualified module scope variable in the constant address space.

An `opaque` variable of `spir.sampler_t` declared in the SPIR program must be initialized using a 32-bit unsigned integer constant, using the following SPIR function:

```
void __spir_sampler_initialize(%spir.sampler_t,  i32) readnone
```

Where the i32 value is interpreted as a bit-field specifiying the following properties:

Sampler State	**Init Values**
addressing mode	CLK_ADDRESS_NONE=0 CLK_ADDRESS_CLAMP=1 CLK_ADDRESS_CLAMP_TO_EDGE=2 CLK_ADDRESS_REPEAT=3 CLK_ADDRESS_MIRRORED_REPEAT=4
normalized coords	CLK_NORMALIZED_COORDS_FALSE=0 CLK_NORMALIZED_COORDS_TRUE=8
filter mode	CLK_FILTER_NEAREST=0 CLK_FILTER_LINEAR=16

Table 13: sampler state initialization values

In addition we introduce a function to initialize module scope variables:

```
void __spir_globals_initializer()
```

The call to the sampler initialization function is performed inside this function. Example:

```
%spir.sampler_t = type opaque

@SMP = constant %spir.sampler_t addrspace(2) zeroinitializer

define spirfnc void @__spir_globals_initializer() {
     call spirfnc void @__spir_sampler_initialize(%spir.sampler_t @SMP, i32 1)
}
```

Notes:

- The i32 argument of `__spir_sampler_initialize` must be a constant value.

2.1.3.2 Constructing a zero event type

Constructing a zero event type is achieved by calling the `%spir.event_t __spir_eventt_null()` SPIR function.

2.1.3.3 NULL pointer

SPIR introduces a set of new functions that generates a NULL pointer per address space: `i8 addrspace(`A`)* __spir_get_null_ptr`A`()`
where A denotes the numeric address space.

Note: The unmangled name of the function is shown.

2.1.4 Alignment of Types

SPIR follows the alignment rules of OpenCL. When the OpenCL specification does not define a minimum alignment, e.g. for size_t, a default alignment of 1 may be used. Therefore:

- Stack allocations and module scope variable declarations must follow the alignment rules defined in OpenCL specification, or use a default of 1.
- All `load` and `store` operations need to be aligned.

2.1.5 Structs

The alignment of structures data members is the alignment of the SPIR data type. Extra padding is disallowed. The alignment of the structure is the alignment of the member which requires the largest alignment. Pointer data members consume 64 bits and are aligned to 64 bits.

When mapping an OpenCL C struct data type to SPIR, the order of members shall be preserved.

2.2 Address space qualifiers

OpenCL C address spaces are mapped to the LLVM `addrspace(n)` qualifier using the following convention:

- 0 – private
- 1 – global
- 2 – constant
- 3 – local
- 4 – global with `endian(host)` attribute
- 5 – constant with `endian(host)` attribute

Note: Casts between address spaces is disallowed in SPIR.

Note: Each OpenCL C function-scope local variable is mapped into an LLVM module-level variable in address space 3. They are not allocated using `alloca` instruction. The name of the module-level variable consists of the (mangled) function name, followed by a period, followed by the (mangled) the source identifier.

Example OpenCL C program:

```
void foo(void) {
  local float4 lf4;
}
```

A valid SPIR mapping:

```
; Unmangled component names shown here.
; float4 must be 16 bytes aligned.
@foo.lf4 = internal addrspace(3) global <4 x float> zeroinitializer, align 16

define spirkrnl void @foo() nounwind {
entry:
  ret void
}
```

In OpenCL C, a kernel function can call another kernel. However, when the called kernel declares a variable in the __local address space, then the behaviour is implementation defined. SPIR supports a kernel calling another kernel, but does not allow the called kernel to have a variable in the __local address space. For example, the following example is not valid SPIR:

```
@bar.lf4 = internal addrspace(3) global <4 x float> zeroinitializer, align 16

define spirkrnl void @bar() nounwind {
```

```
entry:
  ret void
}

define spirkrnl void @callbar() nounwind {
entry:
  call spirkrnl void @bar()  ; This is not supported by SPIR
  ret void
}
```

2.3 Access qualifiers

In OpenCL C, each kernel argument that is an image object is associated with an access qualifier. In SPIR, access qualifiers are associated with kernel arguments by attaching a metadata object to the kernel function. The metadata object is a list of the following:

- A `"access_qualifier"` string metadata object
- One constant `i32` value per kernel argument, in the same order as the kernel arguments in OpenCL C.

The `i32` value mapping is as follows:

- 0 - read only
- 1 - write only
- 2 - Reserved for read-and-write
- 3 - none (Use this for kernel arguments tha are not image objects)

For example, consider the following OpenCL C function:

```
kernel void foo(
    read_only image2d_t A,
    image2d_t B,  // implicitly read_only
    write_only image2d_t C,
    int N ) { ... }
```

The corresponding access qualifier metadata object would be as follows (except it probably isn't `!0`):

```
!0 = metadata !{!"access_qualifier", i32 0, i32 0, i32 1, i32 3}
```

Notes:

- Failing to attach access qualifiers to image arguments is invalid.
- A mismatch between the number of kernel arguments and the `i32` values in the metadata object is invalid.
- Usage of values disallowed values in the metadata objects is invalid.

2.4 Function qualifiers

Functions in SPIR are divided into OpenCL kernels and user functions.

Adding qualifiers and attributes to a kernel or a user function and its arguments is achieved by usage of the LLVM metadata infrastructure. Each SPIR module has a `spir.functions` named metadata node containing a list of metadata objects. Each metadata object in `spir.functions` references a list of metadata objects, each of which represents a single kernel or a single user function. The first value in a SPIR function metadata object is the SPIR function that represents an OpenCL kernel or a user function. The rest of the metadata objects are additional attributes and information which is attached to the SPIR function. The description of each metadata object inside the SPIR function metadata list is described in the other sections.

The following LLVM textual representation shows how SPIR function attributes are represented:

```
!spir.functions = !{ !0,!1,...,!N }
; Note:  The first element is always an LLVM::Function signature
!0 = metadata !{ <function signature>, !0_1, !0_2, ..., , !0_i }
!1 = metadata !{ <function signature>, !1_1, !1_2, ..., , !1_j }
...
!N = metadata !{ <function signature>, !N_1, !N_2, ..., , !N_k }
```

2.4.1 Optional attribute qualifiers

2.4.1.1 Work group size information

Attaching `work_group_size_hint` and `reqd_work_group_size` information to kernels is achieved using LLVM metadata infrastructure. Two new metadata object are introduced. The first item in the metadata object is the string `"work_group_size_hint"` or `"reqd_work_group_size"` followed by three `i32` constant values. The three `i32` values specify the `(X,Y,Z)` group dimensions.

```
; work_group_size_hint(128,1,1)
!0 = metadata !{ metadata !"work_group_size_hint", i32 128, i32 1, i32 1}
; reqd_work_group_size(128,1,1)
!1 = metadata !{ metadata !"reqd_work_group_size", i32 128, i32 1, i32 1}
```

Note:

- Attaching the work group size hint to a non-kernel SPIR function is invalid.

2.4.1.2 Vector type hint information

Attaching `vec_type_hint` information to kernels is achieved using LLVM metadata infrastructure. The first argument in each metadata object is the string `"vec_type_hint"` followed by a typed `undef` LLVM value and an additional `i1` value representing the signedness of the value.

```
; vec_type_hint(float)
!0 = metadata !{ metadata !"vec_type_hint", float undef, i1 1}
; vec_type_hint(uint8)
!1 = metadata !{ metadata !"vec_type_hint", <8 x i32> undef, i1 0}
...
; vec_type_hint(<type>)
!H = metadata !{ metadata !"vec_type_hint", <type> undef, i1 isSigned}
```

Note:

- Attaching vector type hint information to a non-kernel SPIR function is invalid.

- The `double` data type is an optional type and using it requires marking the SPIR module as using the `cl_doubles` optional core feature. See Section 2.12.1.

2.5 Kernel Arg Info

When compiling a kernel with the -cl-kernel-arg-info option the following information is preserved using the LLVM metadata infrastructure. This metadata is added to the list of metadata objects in the kernel list of metadata objects. The first argument in the metadata object is the string `"cl-kernel-arg-info"` followed by a list of metadata objects based on the number kernel arguments. Each metadata object contains a list of metadata object pairs. Each pair consists of a string and its value. The following table shows the valid pairs:

ARG Info	**Type**	**Values**
`"address_qualifier"`	i32	0 – private 1 – global 2 – constant 3 – local 4 – global with endian(host) attribute 5 – constant with endian(host) attribute
`"access_qualifier"`	i32	0 – read only 1 – write only 2 – read and write 3 – none
`"arg_type_name"`	string metadata	The type name specified for the argument. The type name returned will be the argument type name as it was declared with any whitespace removed. If argument type name is an unsigned scalar type (i.e. unsigned char, unsigned short, unsigned int, unsigned long), uchar, ushort, uint and ulong will be returned. The argument type name returned does not include any type qualifiers.
`"arg_type_qualifier"`	i32	0 – none 1 – const 2 – restrict 4 – volatile
`"arg_name"`	string metadata	the name specified for the argument

Table 14: Kernel Arg Info metadata description

2.6 Storage class specifier

The OpenCL C `extern` and `static` storage class specifiers map to the LLVM `external` and `internal` linkage types, respectively.

2.7 Type qualifiers

OpenCL C Type Qualifier	**LLVM Mapping**
const	constant
restrict	noalias
volatile	Certain memory accesses, such as loads, stores, and spir memcpys may be marked volatile. (See Notes below.)

Table 15: Mapping of type qualifiers

Notes for the `volatile` qualifier:

1. The optimizers must not change the number of volatile operations or change their order of execution relative to other volatile operations.
2. The optimizers may change the order of volatile operations relative to non-volatile operations.

2.8 Attribute Qualifiers

2.8.1 Type Attributes

SPIR provides structure types to describe unions and structures. The layout of structures in SPIR must take into consideration the alignment rules of OpenCL C. Optimizers are not allowed to do any modifications to structures.

2.8.1.1 aligned attribute

SPIR structures can be aligned at declaration time. This applies both to module level structures and stack allocations using the `alloca` instruction.

2.8.1.2 packed attribute

SPIR structures are marked as packed when `__attribute__((packed))` is used in OpenCL C.

`Example:`

<{i8 , i32}> is a packed structure known to be 5 bytes in size.

2.8.2 Variable Attributes

2.8.2.1 aligned attribute

- SPIR variables can be aligned at declaration time. This applies both to module level variables and stack allocations using the `alloca` instruction.
- SPIR does not provide a mechanism to reflect the alignment of structure members. Instead the SPIR generator is expected to create a structure definition taking into consideration this attribute, for example by inserting dummy members to occupy the extra space. Optimizers are not allowed to modify the data layout of structures.

2.8.2.2 endian attribute

Marking endianess of variables in the program `endian(host)` attribute is achieved using LLVM address space mechanism.

- 4 - global memory with `endian(host)` attribute
- 5 - constant memory with `endian(host)` attribute

Note: Casts between address spaces is disallowed in SPIR.

2.9 Compiler Options

Compiler optinos are represented in SPIR using a named metadata node `spir.compiler.options`. The named metadata node will contain a single metadata node that holds a list of string metadata objects. Each string metadata object corresponds to a single standard OpenCL compiler option. Preprocessor options are not saved in SPIR and the list of the allowed options are as follows:

- -cl-single-precision-constant
- -cl-denorms-are-zero
- -cl-fp32-correctly-rounded-divide-sqrt
- -cl-opt-disable
- -cl-mad-enable
- -cl-no-signed-zeros
- -cl-unsafe-math-optimizations
- -cl-finite-math-only
- -cl-fast-relaxed-math
- -w
- -Werror
- -cl-kernel-arg-info
- -create-library
- -enable-link-options

Note: The `-cl-std` option is propagated to the `spir.ocl.version` as defined in Section 2.14, OpenCL Version.

This example indicates that both `-cl-mad-enable` and `-cl-denorms-are-zero` standard compile options were used to compile the module:

```
!spir.compiler.options = !{!2}
!2 = metadata !{metadata !"-cl-mad-enable", metadata !"-cl-denorms-are-zero"}
```

Compilation options which are not part of the OpenCL specification are stored via the named metadata node `spir.compiler.ext.options`. The named metadata node contains a single metadata node that holds a list of string metadata objects. Each string metadata object corresponds to a non-standard compile option. Compilation options which appear in `spir.compiler.ext.options` shall not affect functional portability of the SPIR module.

This example indicates that the (hypothetical) non-standard option `-opt-arch-pdp11` was used to compile the module:

```
!spir.compiler.ext.options = !{!5}
!5 = metadata !{metadata !"-opt-arch-pdp11"}
```

2.10 Preprocessor Directives and Macros

2.10.1 Preprocessor Directives

The named metadata `spir.disable.FP_CONTRACT` can be used to disable contractions at module level.

Note: This is one case where some valid OpenCL C programs are not expressible in SPIR. OpenCL C permits control over the FP_CONTRACT pragma at a granular level: at various points in program scope, and within functions. In contrast, SPIR only supports a single module-wide setting.

2.10.2 Macros

It is the SPIR generator's responsibility to deal with the following macros:

- Replace user macros
- Replace `__FILE__` with a character string literal
- Replace `__LINE__` with an i32 contsant
- Replace `CL_VERSION_1_0` with the i32 constant 100
- Replace `CL_VERSION_1_1` with the i32 constant 110
- Replace `CL_VERSION_1_2` with the i32 constant 120
- Replace `CL_VERSION_2_0` with the i32 constant 200
- Replace `__OPENCL_C_VERSION__` with the i32 constant described in -cl-std build option. If the -cl-std build option is not specified the behavior of this Macro follows the `__OPENCL_VERSION__` rules.
- Replace `__OPENCL_VERSION__` with call to the new `i32 __spir_opencl_version()` builtin function which exposes the OpenCL "C" version supported by the device. The return value of this function is 100, 110, or 120 for OpenCL version 1.0, 1.1 and 1.2 (respectively).
- Replace `__ENDIAN_LITTLE__` with a call to the new `i32 __spir_endian_little()` builtin function which is used to determine if the OpenCL device is a little endian architecure or a big endian architecture. The return value of this function is 1 if the device is little endian and undefined otherwise.
- Replace `__IMAGE_SUPPORT__` with a call to the new `i32 __spir_image_support()` builtin function which is used to determine if the OpenCL device supports images. The return value of this function is 1 if the device supports images and is undefined otherwise.
- Replace `__FAST_RELAXED_MATH__` with an i32 constant 1 if the -cl-fast-relaced-math build option is used.
- **TBD:** Device Specific C99 Macros are resolved the same way we will decide to deal with size_t (not sure this is allowed in OpenCL C)

Note: The builtin functions described in this subsection are shown with their unmangled names.

2.11 Built-ins

2.11.1 Name Mangling

Notes:

- Reminder: All of the built-in names described in this document are shown in their unmangled form.

2.11.1.1 Synchronization Functions

Synchronization functions accept `cl_mem_fence_flags` enumeration as an argument. In SPIR this maps to a constant `i32` value which is a bitwise OR between CLK_LOCAL_MEM_FENCE = 1 and CLK_GLOBAL_MEM_FENCE = 2.

Note: The legal values are 1, 2, and 3

2.11.1.2 Built-ins with size_t arguments

Many builtin functions have variants taking arguments of type `int`, `uint`, `long`, or `ulong`. Each of these variants is representable in SPIR by mangling the original builtin function name along with the source language types of the arguments and the return value.

However, for each such builtin, SPIR also supports a variation of such a builtin as if `size_t` were substituted with `int`, `uint`, `long`, or `ulong` in any argument type or return value.

For example, OpenCL C supports the the `clz` function over types *itype* and all vector types over *itype* where *itype* is one of `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`. As with all builtins, SPIR supports all these variants of `clz` via mangling of the argument type and the return type. However, SPIR also supports the variant of `clz` as if it were declared in OpenCL as `size_t clz( size_t )`.

2.11.2 The printf function

The `printf` function is supported, and is mangled according to its prototype as follows:

```
int printf(constant char * restrict fmt, ... )
```

Note that the ellipsis formal argument (`...`) is mangled to argument type specifier `z`.

Use of a scalar `float` argument with floating point conversion specifiers (**f**, **F**, **e**, **E**, **g**, **G**, **a**, or **a**) is not portable. The OpenCL specification says that on devices supporting doubles, the float value is converted to a double before calling the function; on devices that do not support doubles, no conversion is performed. SPIR allows either alternative to be expressed, but does not require consistent behaviour across devices for this case.

2.12 KHR Extensions

2.12.1 Declaration of used optional core features

The named metadata object `spir.used.optional.core.features` contains a single metadata object. The metadata object should contain a list of metdata strings, each of which encodes the name of an optional core feature used by the SPIR module.

This is the list of valid strings and their meaning:

- `"cl_images"` - indicates that images are used
- `"cl_doubles"` - indicates that doubles are used

A device may reject a SPIR module using an unsupported optional core feature.
This example indicates that the module uses both images and doubles.

```
!spir.used.optional.core.features = !{!0}
!0 = metadata !{metadata !"cl_doubles", metadata !"cl_images"}
```

2.12.2 Declaration of used KHR extensions

A SPIR module using one or more KHR extension, must declare them inside the SPIR module. The named metadata object `spir.used.extensions` is used to declare this list. The named metadata object contains a metadata object consisting of a list of metadata strings, where each string indicates a usage of a KHR extension inside the SPIR module.

This is the list of extension strings:

- cl_khr_int64_base_atomics
- cl_khr_int64_extended_atomics
- cl_khr_fp16
- cl_khr_gl_sharing
- cl_khr_gl_event
- cl_khr_d3d10_sharing
- cl_khr_media_sharing
- cl_khr_d3d11_sharing
- cl_khr_global_int32_base_atomics
- cl_khr_global_int32_extended_atomics
- cl_khr_local_int32_base_atomics
- cl_khr_local_int32_extended_atomics
- cl_khr_byte_addressable_store
- cl_khr_3d_image_writes

This example shows that `cl_khr_fp16` and `cl_khr_int64_base_atomics` standard extensions are used in the module.

```
!spir.used.extensions  = !{!6}
!6 = metadata !{metadata !"cl_khr_fp16", metadata !"cl_khr_int64_base_atomics"}
```

Notes:

- A device may reject a SPIR module using an unsupported KHR extension.
- A device using `cl_khr_3d_image_writes` must also declare its use of `cl_images` inside `spir.used.optional.core.features`.
- **cl_khr_fp64** doesn't exist in SPIR. Instead SPIR generators should use the `cl_doubles` optional core features.

2.13 SPIR Version

The SPIR version used by the module is stored in the `spir.version` named metadata. The named metadata contains a metadata node consisting of a list of two `i32` constant values denoting the major and minor version numbers.

The following example indicates the module uses SPIR version 1.0:

```
!spir.version = !{!3}
!3 = metadata !{i32 1, i32 0}
```

2.14 OpenCL Version

The OpenCL version used by the module is stored in the `spir.ocl.version` named metadata node. The named metadata node contains a metadata node consisting of a list of two `i32` constant values denoting the major and minor version numbers.

This example indicates the module is compiled for OpenCL 1.0:

```
!spir.ocl.version = !{!4}
!4 = metadata !{i32 1, i32 0}
```

This example indicates the module is compiled for OpenCL 1.1:

```
!spir.ocl.version = !{!4}
!4 = metadata !{i32 1, i32 1}
```

2.15 memcpy functions

The `__spir_memcpy` functions copy a block of memory from an i8* (void) source location to an i8* (void) destination location.

Syntax:
declare void @__spir_memcpy(i8 addrspace($A1$)* dst, i8 addrspace($A2$)* src, i32 len, i32 align, i1 isVolatile)
declare void @__spir_memcpy(i8 addrspace($A1$)* dst, i8 addrspace($A2$)* src, i64 len, i32 align, i1 isVolatile)

Where $A1$ and $A2$ can denote the address space of the first and second pointer arguments accordingly.

Arguments: the first argument is a pointer to the destination, the second is a pointer to the source. The third argument is an integer argument specifying the number of bytes to copy, the fourth argument is the alignment of the source and destination locations, and the fifth is a boolean indicating a volatile access.

If the call to this intrinsic has an alignment value that is not 0 or 1, then the caller guarantees that both the source and destination pointers are aligned to that boundary.

Semantics: Copy a block of memory from the source location to the destination location, which are not allowed to overlap. It copies "len" bytes of memory over. If the argument is known to be aligned to some boundary, this can be specified as the fourth argument, otherwise it should be set to 0 or 1.

3 SPIR and LLVM IR

3.1 LLVM Triple

SPIR introduces a new LLVM triple called named "`spir`":

```
target triple = "spir"
```

3.2 LLVM Target data layout

The `spir` triple datalayout is as follows:

```
target datalayout = "i1:8-i8:8-i16:16-i32:32-i64:64-f32:32-f64:64-v16:16-v24:32-v32:32
  -v48:64-v64:64-v96:128-v128:128-v192:256-v256:256-v512:512-v1024:1024"
```

Note: Endianness, pointers, stack objects and native integer sizes are not a part of the datalayout.

3.3 LLVM Supported Instructions

The following tables show which LLVM instructions are may be used in SPIR:

LLVM Instruction Family	**Instruction name**	**Supported**
Terminator	ret	yes
Terminator	br	yes
Terminator	switch	yes
Terminator	indirectbr	no, required for GNU extension (array of pointer of functions)
Terminator	invoke	no, exception handling related
Terminator	unwind	no, exception handling related
Terminator	resume	no, exception handling related
Terminator	unreachable	yes, might be used for switch statements
Binary	add	yes
Binary	fadd	yes
Binary	sub	yes
Binary	fsub	yes
Binary	mul	yes
Binary	fmul	yes
Binary	udiv	yes
Binary	sdiv	yes
Binary	fdiv	yes
Binary	urem	yes
Binary	srem	yes
Binary	frem	yes
Bitwise Binary	shl	yes, left-shifted by log2(N), where N is the number of bits used to represent the data type of the shifted value
Bitwise Binary	lshr	yes, right-shifted by log2(N), where N is the number of bits used to represent the data type of the shifted value.
Bitwise Binary	ashr	yes, right-shifted by log2(N), where N is the number of bits used to represent the data type of the shifted value. `exact` is disallowed and used for trap values
Bitwise Binary	and	yes
Bitwise Binary	or	yes
Bitwise Binary	xor	yes
Vector	extractelement	yes
Vector	insertelement	yes
Vector	shufflevector	yes
Aggregate	extractvalue	yes
Aggregate	insertvalue	yes
Memory Access & Addressing	alloca	yes
Memory Access & Addressing	load	yes, `atomic` is disallowed
Memory Access & Addressing	store	yes, `atomic` is disallowed
Memory Access & Addressing	fence	no, use built-ins instead
Memory Access & Addressing	cmpxchg	no, use built-ins instead
Memory Access & Addressing	atomicrmw	no, use built-ins instead
Memory Access & Addressing	getelementptr	yes

Table 16: Instructions, part 1

LLVM Instruction Family	**Instruction name**	**Supported**
Conversion Operations	trunc .. to	yes, but only for scalars
Conversion Operations	zext .. to	yes, but only for scalars
Conversion Operations	sext .. to	yes, but only for scalars
Conversion Operations	fptrunc .. to	yes, but only for scalars
Conversion Operations	fpext .. to	yes, but only for scalars
Conversion Operations	fptoui .. to	yes, but only for scalars
Conversion Operations	fptosi .. to	yes, but only for scalars
Conversion Operations	uitofp .. to	yes, but only for scalars
Conversion Operations	sitofp .. to	yes, but only for scalars
Conversion Operations	ptrtoint .. to	no, use size_t intrinsics instead
Conversion Operations	inttoptr .. to	no, use size_t intrinsics instead
Conversion Operations	bitcast .. to	yes
Other Operations	icmp	yes
Other Operations	fcmp	yes
Other Operations	phi	yes
Other Operations	select	yes
Other Operations	call	yes, but not to pointers to functions
Other Operations	va_arg	no, not supported by OpenCL
Other Operations	landingpad_arg	no

Table 17: Instructions, part 2

3.4 LLVM Supported Intrinsic Functions

None of the LLVM intrinsics are allowed in SPIR.

3.5 LLVM Linkage Types

The following table shows the LLVM linkage types allowed in SPIR:

Linkage type	**Supported**
private	yes
linker_private	no
linker_private_weak	no
linker_private_weak_def_auto	no
available_externally	yes (describes C99 inline definition)
linkonce	no
internal	yes (maps to static)
weak	no
common	yes
appending	no
extern_weak	no
linkonce_odr	no
weak_odr	no
external	yes (will be required for libraries)
dllimport	no
dllexport	no

Table 18: Linkage types

3.6 Calling Conventions

SPIR kernels should use `"spirkrnl"` calling convention. Non-kernel functions use `"spirfnc"` calling convention. All other calling conventions are disallowed. [3]

3.7 Visibility Styles

Visibility styles are not used in SPIR and should be set to **"default"**. Other values are disallowed.

3.8 Parameter Attributes

The following table defines which parameter attributes are usable in SPIR:

Parameter Attribute	**Supported**
zeroext	yes
signext	yes
inreg	no
byval	yes
sret	yes
nocapture	yes
nest	no

Table 19: Parameter attributes

3.9 Garbage Collection Names

Garbage collection is not part of SPIR, hence functions are not allowed to specify a garbage collector name.

[3] If we are unable to add this calling convention to the LLVM open source project, we will switch back to the "ccc" calling convention for SPIR.

3.10 Function Attributes

Every SPIR function should use the `nounwind` attribute. In addition the following optional attributes could be used: `alwaysinline`, `inlinehint`, `noinline`, `readnone`, `readonly`. The rest of the function attributes are disallowed.

Function Attribute	**Supported**
alignstack	no
alwaysinline	yes
nonlazybind	no
inlinehint	yes
naked	no
noimplicitfloat	no
noinline	yes
noredzone	no
noreturn	no
nounwind	yes, needs to be always set
optsize	no
readnone	yes
readonly	yes
ssp	no
sspreq	no
uwtable	no
returns_twice	no

Table 20: Function attributes

3.11 Reserved identifiers

All identifiers that begin with `__spir` and `spir.*` are reserved and shall not be used by SPIR generators (for user source identifiers).

3.12 Module Level Inline Assembly

LLVM module level inline assembly is not allowed in SPIR.

3.13 Pointer Aliasing Rules

SPIR follows the pointer aliasing rules of LLVM.

3.14 Volatile Memory Accesses

SPIR requires use of volatile memory accesses and follows LLVM IR rules for `load`'s, `store`'s and `spir.memcpy`'s.

3.15 Memory Model for Concurrent Operations

SPIR does not use the LLVM atomic intrinsics, because OpenCL has its own set of intrinsics.

3.16 Atomic Memory Ordering Constraints

The LLVM atomic orderings are disallowed in SPIR.

A SPIR name mangling

In order to support cross device compatibility of SPIR, the name mangling scheme must be standarized across vendors. SPIR adopts and extends the name mangling scheme in Section 5.1 of the Itanium C++ ABI [1]. There are three major issues to deal with, along with many minor items. The major items are data types, address spaces, and overloaded 'C' functions.

Normally, 'C' functions require no overloading, and their names are not mangled. When generating SPIR, functions must use this mangling scheme if and only if they:

- Are defined in the SPIR specification, or
- Are OpenCL C built-in functions that are overloaded over their argument types.

A.1 Data types

The following table shows the mapping from OpenCL C data types to the type names used in the mangling scheme:

OpenCL C type	**Mangling scheme type name**
bool	b
unsigned char, char	h
char	c
unsigned short, short	t
short	s
unsigned int, uint	j
int	i
unsigned long, ulong	m
long	l
half	Dh
float	f
double	d
Vector types with up to 8 elements	u2v*N*<*mangled-element-type-name*> (where N is one of 2, 3, 4, 8)
Vector types with 16 elements	u3v16<*mangled-element-type-name*>
image1d_t	u3i1d
image1d_array_t	u3i1a
image1d_buffer_t	u3i1b
image2d_t	u3i2d
image2d_array_t	u3i2a
image3d_t	u3i3d
event_t	u2ev
sampler_t	u2sa
size_t, uintptr_t	u2sz
ptrdiff_t, intptr_t	u2pd

Table 21: Mapping of OpenCL C builtin type names to mangled type names

A.2 Type attributes

The following table shows the mapping from OpenCL C specific type qualifiers to their mangled encoding:

OpenCL C type attribute	**Mangled encoding**
read_only	U1R
write_only	U1W
read_write (Reserved)	U1B
LLVM address space *N*	U2A*N*

Table 22: Mapping of OpenCL C type attributes to mangled names

A.3 The restrict qualifier

The Itanium ABI states:

> The restrict qualifier is part of the C99 standard, but is strictly an extension to C++ at this time. There is no standard specification of whether the restrict attribute is part of the type for overloading purposes. An implementation should include its encoding in the mangled name if and only if it also treats it as a distinguishing attribute for overloading purposes. This ABI does not specify that choice."

SPIR encodes the "restrict" qualifier as part of the mangled name using the 'r' token in the CV-qualifiers. Hence SPIR treats the "restrict" qualifier as significant for overloading.

A.4 Summary of changes

The following is a summary of the mangling of builtin types:

```
<builtin-type> ::= v  # void (Maps to OpenCL void)
          ::= w   # wchar_t (*Not valid)
          ::= b   # bool (Maps to OpenCL bool)
          ::= c   # char(Maps to OpenCL char)
          ::= a   # signed char (*Not valid)
          ::= h   # unsigned char (Maps to OpenCL uchar)
          ::= s   # short (Maps to OpenCL short)
          ::= t   # unsigned short (Maps to OpenCL ushort)
          ::= i   # int (Maps to OpenCL int)
          ::= j   # unsigned int (Maps to OpenCL uint)
          ::= l   # long (Maps to OpenCL long)
          ::= m   # unsigned long(Maps to OpenCL ulong)
          ::= x   # long long, __int64(*Not valid)
          ::= y   # unsigned long long, __int64(*Not valid)
          ::= n   # __int128 (*Not valid)
          ::= o   # unsigned __int128(*Not valid)
          ::= f   # float (Maps to OpenCL float)
          ::= d   # double (Maps to OpenCL double)
          ::= e   # long double, __float80(*Not valid)
          ::= g   # __float128 (*Not valid)
          ::= z   # ellipsis (*Valid only for printf*)
          ::= Dd # IEEE 754r decimal floating point (64 bits) (*Not valid)
          ::= De # IEEE 754r decimal floating point (128 bits) (*Not valid)
          ::= Df # IEEE 754r decimal floating point (32 bits) (*Not valid)
          ::= Dh # IEEE 754r half-precision floating point (16 bits) (Maps to OpenCL Half)
          ::= Di # char32_t(*Not valid)
          ::= Ds # char16_t(*Not valid)
          ::= Da # auto (in dependent new-expressions)
          ::= Dn # std::nullptr_t (i.e., decltype(nullptr))
```

```
            ::= u[23]vN <builtin-type> # An OpenCL vector of length 'N' of the specified type.
                                       # Only values of 2, 3, 4, 8 and 16 are valid.
            ::= u3i1d # A 1d image type
            ::= u3i1a # A 1d image array type
            ::= u3i1b # A 1d image buffer type
            ::= u3i2d # A 2d image type
            ::= u3i2a # A 2d image array type
            ::= u3i3d # A 3d image type
            ::= u2ev # A event type
            ::= u2sa # A sampler type
            ::= u2sz # A size_t type
            ::= u2pd # A ptrdiff_t type
            ::= u <source-name> # vendor extended type
```

SPIR also extends the CV-qualifier list as follows. All CV-qualifiers are order-insensitive.

```
<CV-qualifiers> ::= [r] [V] [K]   # restrict (C99), volatile, const
                ::= [U1W] [U1B] [U1R]   # write_only, read_write("both"; Reserved), read_only
                ::= [U2A[N]]            # Address space N, where N is valid for 1-5.
                                        # These are order-insensitive.
```

Note: By default, objects reside in the `private` address space (number 0). No address space qualification is used to indicate the private address space.

References

[1] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, SGI, and others. Itanium C++ ABI. `http://mentorembedded.github.com/cxx-abi/abi.html`.

[2] Khronos OpenCL Working Group. The OpenCL Specification, version 1.2. `http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`, November 2011.

[3] LLVM Team. LLVM Bitcode File Format. `http://www.llvm.org/releases/3.1/docs/BitCodeFormat.html`, 2012. Version 3.1.

[4] LLVM Team. LLVM Language Reference Manual. `http://www.llvm.org/releases/3.1/docs/LangRef.html`, 2012. Version 3.1.