The SPIR™ Specification

Standard Portable Intermediate Representation

Version 2.0 – Provisional

Revision Date: 2014-06-05

Editor: Boaz Ouriel

Contents

List of Tables

Acknowledgements

Editor: Boaz Ouriel, Intel

Contributors:

- David Neto, Altera
- Anton Lokhmotov, ARM
- Mike Houston, AMD
- Micah Villmow, AMD
- Brian Sumner, AMD
- Richard Relph, AMD
- Yaxun Liu, AMD
- Tanya Lattner, Apple
- Aaftab Munshi, Apple
- Holger Waechtler, Broadcom
- Christopher Thomson-Walsh, Broadcom
- Andrew Richards, Codeplay
- Guy Benyei, Intel
- Javier E. Martinez, Intel
- Boaz Ouriel, Intel
- Dillon Sharlet, Intel
- Vinod Grover, NVIDIA
- Kedar Patil, NVIDIA
- Yuan Lin, NVIDIA
- Sumesh Udayakumaran, QUALCOMM
- Chihong Zhang, QUALCOMM
- Lee Howes, QUALCOMM
- Jack Liu, QUALCOMM
- James Price, University of Bristol
- Henry Styles, Xilinx

1 Introduction

This document defines version 2.0 of the Standard Portable Intermediate Representation (SPIR). SPIR 2.0 is a mapping from the OpenCL C programming language into LLVM IR.

This version of the specification is based on LLVM 3.4 [4] [3], and on OpenCL C as specified in the OpenCL 2.0 Specification [2].

The goal of SPIR 2.0 is to provide a portable interchange format for partially compiled OpenCL C programs. The format:

- Is vendor neutral.
- Is not C source code.
- Is designed to support vendor extensions.
- Is compact.
- Is designed to be efficiently loaded by an OpenCL implementation.
- 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]. [1]
- 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

OpenCL C has many overloaded built-in functions, meaning the same function name is used with different argument 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 the variations of the `sin` function by mangling the root name `sin` with its argument types.

This means that in SPIR all of the OpenCL C built-in functions are mangled based 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 user-defined 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.

[1] The LLVM 3.4 bitcode notation is only partly documented by [3]. However, bitcode notation is fully (but implicitly) defined by the behaviour of LLVM 3.4 software release.

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.

2.1.2 Built-in Vector Types

Table 2 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 2: Mapping for built-in vector types

The vector types allowed in SPIR are the vector types described in Table 2 and in addition the < n x i1 > vector type. Vectors of type < n x i1 > may only be used as the type of the result of `icmp` and `fcmp` instructions, operands to `sext` and `zext` instructions, operands and result of `extractelement`, `insertelement` and `shufflevector` and as operands and result of the bitwise operations `and`, `or` and `xor`. Any other use of < n x i1 > vectors is prohibited, including logical and arithmetical shift operations, arithmetic operations of any kind, store, load, and use as function argument or return type, or as structure field, array element or base type of a pointer.

2.1.3 Other Built-in Data Types

Table 3 defines the mapping of OpenCL image, sampler, event, queue_t, ndrange_t, clk_event_t, reserve_id_t, cl_mem_fence_flags, size_t, ptrdiff_t,uintptr_t,intptr_t data types to LLVM data types.

OpenCL C Type	**LLVM Type**	**LLVM Name**
image1d_t	opaque*	%opencl.image1d_t
image1d_array_t	opaque*	%opencl.image1d_array_t
image1d_buffer_t	opaque*	%opencl.image1d_buffer_t
image2d_t	opaque*	%opencl.image2d_t
image2d_array_t	opaque*	%opencl.image2d_array_t
image3d_t	opaque*	%opencl.image3d_t
image2d_msaa_t	opaque*	%opencl.image2d_msaa_t
image2d_array_msaa_t	opaque*	%opencl.image2d_array_msaa_t
image2d_msaa_depth_t	opaque*	%opencl.image2d_msaa_depth_t
image2d_array_msaa_depth_t	opaque*	%opencl.image2d_array_msaa_depth_t
image2d_depth_t	opaque*	%opencl.image2d_depth_t
image2d_array_depth_t	opaque*	%opencl.image2d_array_depth_t
sampler_t	i32	N/A
event_t	opaque*	%opencl.event_t
queue_t	opaque*	%opencl.queue_t
ndrange_t	structure	type { i32, [3 x i64], [3 x i64], [3 x i64] } when the device address width is equal to 64 bits, or type { i32, [3 x i32], [3 x i32], [3 x i32]} when the device address width is equal to 32 bits.
clk_event_t	opaque*	%opencl.clk_event_t
reserve_id_t	opaque*	%opencl.reserve_id_t
cl_mem_fence_flags	i32	N/A
atomic_flag	i32	N/A
atomic_int	i32	N/A
atomic_uint	i32	N/A
atomic_long	i64	N/A
atomic_ulong	i64	N/A
atomic_float	float	N/A
atomic_double	double	N/A
atomic_size_t	i32 or i64	N/A
atomic_ptrdiff_t	i32 or i64	N/A
atomic_uintptr_t	i32 or i64	N/A
atomic_intptr_t	i32 or i64	N/A
N/A	opaque*	%opencl.pipe_t
size_t	i32 or i64	N/A
ptrdiff_t	i32 or i64	N/A
uintptr_t	i32 or i64	N/A
intptr_t	i32 or i64	N/A

Table 3: Mapping for other built-in data types

Notes:

- The size of images and event data types is equal to 32 bits or 64 bits according to the device address width.
- The names given to opaque data types are reserved for SPIR and shall not be used otherwise.

- The OpenCL `size_t`,`ptrdiff_t`,`uintptr_t`, `intptr_t`, `atomic_size_t`, `atomic_ptrdiff_t`, `atomic_uintptr_t` and `atomic_intptr_t` data types are mapped to LLVM `i32` when the device address width is equal to 32 bits and to LLVM `i64` when the device address width is equal 64 bits
- Values that represent `sampler_t` and `atomic_*` objects, can only be passed as arguments to their corresponding built-in functions. Any other operation involving these values is implementation defined.

2.1.3.1 Declaring sampler variables

A sampler variable is an i32 constant-qualified module scope variable in the constant address space, initialized with an i32 constant value. The i32 constant value is interpreted as a bit-field specifying the following properties:

Sampler State	**Init Values**
addressing mode	CLK_ADDRESS_NONE=0x0000 CLK_ADDRESS_CLAMP_TO_EDGE=0x0002 CLK_ADDRESS_CLAMP=0x0004 CLK_ADDRESS_REPEAT=0x0006 CLK_ADDRESS_MIRRORED_REPEAT=0x0008
normalized coords	CLK_NORMALIZED_COORDS_FALSE=0x0000 CLK_NORMALIZED_COORDS_TRUE=0x0001
filter mode	CLK_FILTER_NEAREST=0x0010 CLK_FILTER_LINEAR= 0x0020

Table 4: sampler initialization values

2.1.3.2 Image channel data type values

The get_image_channel_data_type() built-in returns an integer value which represents the image channel data type. The following table indicates the valid values:

Channel order	**Value**
CLK_SNORM_INT8	0x10D0
CLK_SNORM_INT16	0x10D1
CLK_UNORM_INT8	0x10D2
CLK_UNORM_INT16	0x10D3
CLK_UNORM_SHORT_565	0x10D4
CLK_UNORM_SHORT_555	0x10D5
CLK_UNORM_INT_101010	0x10D6
CLK_SIGNED_INT8	0x10D7
CLK_SIGNED_INT16	0x10D8
CLK_SIGNED_INT32	0x10D9
CLK_UNSIGNED_INT8	0x10DA
CLK_UNSIGNED_INT16	0x10DB
CLK_UNSIGNED_INT32	0x10DC
CLK_HALF_FLOAT	0x10DD
CLK_FLOAT	0x10DE
CLK_UNORM_INT24	0x10DF

Table 5: image channel data type values

2.1.3.3 Image channel order values

The get_image_channel_order() built-in returns an integer value which represents the image channel order. The following table indicates the valid values:

Channel order	**Value**
CLK_R	0x10B0
CLK_A	0x10B1
CLK_RG	0x10B2
CLK_RA	0x10B3
CLK_RGB	0x10B4
CLK_RGBA	0x10B5
CLK_BGRA	0x10B6
CLK_ARGB	0x10B7
CLK_INTENSITY	0x10B8
CLK_LUMINANCE	0x10B9
CLK_Rx	0x10BA
CLK_RGx	0x10BB
CLK_RGBx	0x10BC
CLK_DEPTH	0x10BD
CLK_DEPTH_STENCIL	0x10BE
CLK_sRGB	0x10BF
CLK_sRGBx	0x10C0
CLK_sRGBA	0x10C1
CLK_sBGRA	0x10C2

Table 6: image channel order values

2.1.3.4 Zero events

Zero events are represented using the LLVM `null` keyword.

2.1.3.5 NULL pointer

NULL pointers are represented using the LLVM `null` keyword.

2.1.3.6 The ndrange_t structure

`%opencl.ndrange_t` is an llvm structure representing an ND-range descriptor. The structure has the following signature when the device address width is equal to 64 bits:

```
type = { i32, [3 x i64], [3 x i64], [3 x i64] }
```

The structure has the following signature when the device address width is equal to 32 bits:

```
type = { i32, [3 x i32], [3 x i32], [3 x i32] }
```

The structure members are interpreted as follows:

```
type = {
  i32 work_dim := number of dimensions used to specify the global work-items
         and work-items in the work-group.
  [3 x i32 or i64] global_work_offset := an array of work_dim unsigned values that
                     describe the offset used to calculate the global ID of a work-item.
  [3 x i32 or i64] global_work_size := an array of work_dim unsigned values that
                   describe the number of global work-items in work_dim dimensions
                   that will execute the kernel function.
  [3 x i32 or i64] local_work_size := an array of work_dim unsigned values that describe
                   the number of work-items that make up a work-group
}
```

Note: This structure is initialized using the `ndrange_*D()` helper built-in functions and not directly by the SPIR generator.

2.1.3.7 Blocks

Blocks are represented in SPIR as a pointer to `%opencl.block` opaque type. The `spir_block_bind` built-in function is used to create and initialize a block literal. The prototype of this function is declared as follows:

```
opencl.block* @spir_block_bind(
i8* %invoke, ; The pointer to the block invoke function
i32 %captured_size, ; The size of the captured variables structure.
i32 %captured_alignment,; The alignment of the captured variables structure.
i8* %context   ; The pointer to the captured variables structure.
)
```

The first argument `%invoke` must be a bitcast from a block invoke function. The block invoke function must have the same signature as the block literal it implements, with an additional argument which is used to pass the captured variables structure to the invoke function. This argument is the first argument of the invoke function unless the block function returns an aggregate type. In this case the first argument is the pointer to the returned value type (see Section 3.6), and the second is the pointer to the captured variables structure. The context parameter must be an `alloca` instruction. Invoking a block in SPIR is achieved in the following steps:

1. Call `spir_get_block_context()` built in function. The prototype of this functions is declared as follows:

   ```
   i8* @spir_get_block_context(opencl.block* block)
   ```

   The `i8*` value returned by `spir_get_block_context()` is the pointer to the block context.

2. Call `spir_get_block_invoke()` built in function. The prototype of this function is declared as follows:

   ```
   i8* @spir_get_block_invoke(opencl.block* block)
   ```

   The `i8*` value returned by `spir_get_block_invoke()` is the pointer to the block invoke function. This `i8*` should be cast to a function pointer with the signature of the block invoke function.

Restrictions

1. A pointer to `%opencl.block` opaque type can only be used in the following cases:
   (a) Being initialized using the `spir_block_bind()` built-in function.
   (b) Being passed as an argument to a function.
   (c) Being loaded from or stored to a memory location, where the address must be the result of an `alloca` instruction whose type is a `%opencl.block**`. The `load` instruction must be dominated by a `store` instruction to the same memory location, such that for every path from the `store` instruction to the `load` instruction there is no other `store` instruction to the same memory location.
2. The return value of `spir_get_block_invoke()` can only be used as an operand of a `bitcast` instruction to the type of the block invoke function. The result of this `bitcast` can only be used as the first operand of a `call` instruction.
3. The return value of `spir_get_block_context()` can only be used as the second operand of the `call` instruction described in restriction 2.
4. The second operand of the `call` instruction described in restriction 2, can only be the return value of `spir_get_block_context()`.
5. The `spir_block_bind()` must be called after the context is populated with the captured state.
6. The `spir_get_block_invoke()` arguments `captured_size` and `captured_alignment` can only be immediate values.
7. The `spir_get_block_invoke()` argument `invoke` can only be an operand of a `bitcast` instruction from the type of the block invoke function.
8. The captured context resides in private memory

Implementation note: The restrictions above are intended to

- Allow the consumer to statically associate the uses of the `%opencl.block` with its `spir_block_bind()` function during the linkage phase.
- Guarantee that the memory used to store the opaque `%opencl.block` type (allocated by `spir_block_bind()`) and the memory for the captured context are valid when the `%opencl.block` is used.

Examples

Example 1: Block invokation Consider the following OpenCL-C source:

```
void foo( int (^bl)(int) ); /* possibly implemented in another compilation unit */
...
kernel void ker(int q, int p, long16 l) {
  ...
  int (^myBlock)(int) = ^(int num) { return num * q + p + (int)l.a; };
  ...
  foo(myBlock);
  ...
}
```

This is the forward declaration of the function `foo` in SPIR:

```
declare void @foo(%opencl.block*)
```

The block literal referenced by `myBlock` is represented in SPIR as an invoke function with the following signature:

```
define internal i32 @__ker_block_invoke(i8* %.block_captured, i32 %num)
```

The kernel `ker` is represented in SPIR in the following way:

```
define void @ker(i32 %q, i32 %p, <16 x i64> %l) {
  ...
  ; allocate a block reference myBlock, the captured variables structure
  ; and the block literal
  %myBlock = alloca %opencl.block*, align 4
  %captured = alloca <{ <16 x i64>, i32, i32 }>, align 128

  ; capture the variables to be used in the block
  %block.captured1 = getelementptr inbounds <{ <16 x i64>, i32, i32 }>* %captured, i32 0, i32 1
  store i32 %q, i32* %block.captured1, align 4
  %block.captured2 = getelementptr inbounds <{ <16 x i64>, i32, i32 }>* %captured, i32 0, i32 2
  store i32 %p, i32* %block.captured2, align 4
  %block.captured3 = getelementptr inbounds <{ <16 x i64>, i32, i32 }>* %captured, i32 0, i32 0
  store <16 x i64> %l, <16 x i64>* %block.captured3, align 128

  ; initialize the block literal
  %block = call %opencl.block* @spir_block_bind (
                 i8* bitcast (i32  (i8*, i32)* @__ker_block_invoke to i8*),
                 i32 136, i32 128,
                 i8* bitcast <{ <16 x i64>, i32, i32 }>* %captured to i8*)

  ; initialize the block reference
  store %opencl.block* %block, %opencl.block** %myBlock, align 4
  %1 = load %opencl.block** %myBlock, align 4

  ; call foo with myBlock as argument
  call void @foo(%opencl.block* %1)
  ...
}
```

In order to call the block from within the implementation of the function `foo`. The following SPIR code should be generated:

```
define void @foo(%opencl.block* %bl) {
  ...
  ; load the block invoke function address
  %0 = call i8* @ spir_get_block_invoke (%opencl.block* %bl)

  ; load the address of the captured variables structure
  %1 = call i8* @ spir_get_block_context (%opencl.block* %bl)

  ; cast the block invoke function pointer to its actual type
  %2 = bitcast i8* %0 to i32 (i8*, i32)*

  ; call the block invoke function with the arguments required by the block
  %call = call i32 %2(i8* %1, i32 8)
  ...
}
```

Example 2: Enqueue kernel
Consider the following OpenCL-C source:

```
void
my_func_A(global int *a, global int *b, global int *c);

kernel void
my_func_B(global int *a, global int *b, global int *c)
{
  ndrange_t ndrange;
  // build ndrange information
  ...
  // enqueue a kernel as a block
  enqueue_kernel(get_default_queue(), 0, ndrange,
  ^{my_func_A(a, b, c);});
}
```

This is the generated SPIR code:

```
define spir_kernel void @my_func_B(i32 addrspace(1)* %a,
                                   i32 addrspace(1)* %b,
                                   i32 addrspace(1)* %c) #0 {
  ...
  %ndrange = alloca %struct.ndrange_t, align 4
  %captured = alloca <{ i32 addrspace(1)*, i32 addrspace(1)*, i32 addrspace(1)* }>, align 4
  ...
  %block = alloca %struct.__block_literal_generic
  ...
  %call = call spir_func %opencl.queue_t* @get_default_queue()
  ...
  %block.captured1 = getelementptr inbounds <{ i32 addrspace(1)*, i32 addrspace(1)*,
                                  i32 addrspace(1)* }>* %captured, i32 0, i32 0
  store i32 addrspace(1)* %a, i32 addrspace(1)** %block.captured1, align 4
  %block.captured2 = getelementptr inbounds <{ i32 addrspace(1)*, i32 addrspace(1)*,
```

```
                                  i32 addrspace(1)* }>* %captured, i32 0, i32 1
  store i32 addrspace(1)* %b, i32 addrspace(1)** %block.captured2, align 4
  %block.captured3 = getelementptr inbounds <{ i32 addrspace(1)*, i32 addrspace(1)*,
                                   i32 addrspace(1)* }>* %captured, i32 0, i32 2
  store i32 addrspace(1)* %c, i32 addrspace(1)** %block.captured3, align 4
  %block = call %opencl.block* @spir_block_bind (
                     i8* bitcast (void (i8*)* @__my_func_B_block_invoke to i8*), i32 12, i32 4,
                           bitcast <{ i32 addrspace(1)*, i32 addrspace(1)*, i32 addrspace(1)* }>*
                                     %captured to i8*)
  %call4 = call spir_func i32 @_Z14enqueue_kernel9ocl_queuei9ndrange_tU13block_pointerFvvE (
                                      %opencl.queue_t* %call, i32 0,
                                      %struct.ndrange_t* byval %ndrange,
                                      %opencl.block* %block)
  ...
  ret void
}

define internal void @__my_func_B_block_invoke(i8* %.block_captured) #0
```

Example 3: A block returning an aggregate type Consider the following block:

```
typedef struct {
  int x, y;
} coors;

coors (^myBlock)(int) = ^(int num) { coors s = {num, num}; return s; };
```

The block literal referenced by `myBlock` is represented in SPIR as an invoke function of the following signature:

```
define internal void @__ker_block_invoke(%struct.coors * noalias sret %agg.result, i8* %.block_captu
```

Example 4

Program scope blocks literals are also created using `spir_block_bind` built function. The `captured_size`, `captured_alignment` and the `context` arguments should be all zero or NULL. Consider the following OpenCL-C source:

```
void foo(int (^bl)(int)); /* possibly implemented in another compilation unit */
...
int (^myGlobBlock)(int) = ^(int num) { return num * gint; };
...
kernel void ker(int i) {
  ...
  foo(myGlobBlock);
  ...
  int li = myGlobBlock(i);
  ...
}
```

This is the forward declaration of the function `foo` in SPIR:

```
declare void @foo(%opencl.block*)
```

The block literal referenced by `myGlobBlock` is repsernted in SPIR as an invoke function of the following signature:

```
define internal i32 @myGlobBlock_block_invoke(i8* %.block_captured, i32 %num)
```

The kernel `ker` is represented in SPIR in the following way:

```
define void @ker(i32 %q, i32 %p, <16 x i64> %l) {
  ...
  ; allocate a block reference myGlobBlock
  %myGlobBlock = alloca %opencl.block*, align 4

  ; initialize the block literal
  %block = call %opencl.block* @spir_block_bind (
                            i8* bitcast (i32  (i8*, i32)* myGlobBlock_block_invoke i8*),
                            i32 0, i32 0, i8* null)

  ; initialize the block reference
  store %opencl.block* %block, %opencl.block** % myGlobBlock, align 4
  %1 = load %opencl.block** % myGlobBlock, align 4

  ; call foo with myGlobBlock as argument
  call void @foo(%opencl.block* %1)
  ...
```

2.1.4 Alignment of Types

SPIR follows the alignment rules of OpenCL. Therefore:

- Stack allocations and module scope variable declarations must follow the alignment rules defined in OpenCL specification.
- 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.

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 – generic

Note: Casting between address spaces follows the OpenCL 2.0 rules.

Note: Every OpenCL C function-scope local variable is mapped to an LLVM module-level variable in address space 3. They are not allocated using the `alloca` instruction. The name of the module-level variable consists of the name of the function in which the variable is declared followed by a period and then followed by the the name of the variable.

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 spir_kernel 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 spir_kernel void @bar() nounwind {
entry:
  ret void
}

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

2.3 Kernel qualifiers

Adding qualifiers and attributes to a kernel and its arguments is achieved by usage of the LLVM metadata infrastructure. Each SPIR module has a `opencl.kernels` named metadata node containing a list of metadata objects. Each metadata object in `opencl.kernels` references a list of metadata objects, each of which represents a single kernel. The first value in a SPIR function metadata object is the SPIR function that represents an OpenCL kernel. 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:

```
!opencl.kernels = !{ !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 }
```

2.3.1 Optional attribute qualifiers

2.3.1.1 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.

2.3.1.2 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 2.11.1.

2.4 Kernel Arg Info

Kernel argument specific information is preserved using metadata objects. These objects are generated for every kernel except for `kernel_arg_name` and `kernel_arg_optional_qual` metadata which are optional. The metadata nodes describing the kernel argument info are in the form of a string tag, and then a list of the corresponding data for each one of the kernel's arguments.

The following table shows the valid kernel argument information types and values:

ARG Info	**Type**	**Values**
`"kernel_arg_addr_space"`	i32	0 – private 1 – global 2 – constant 3 – local
`"kernel_arg_access_qual"`	string metadata	"read_only" "write_only" "read_write" "none"
`"kernel_arg_optional_qual"`	string metadata	"nosvm" "none"
`"kernel_arg_type"`	string metadata	The type name specified for the argument. The type name 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.
`"kernel_arg_base_type"`	string metadata	The base type name of the argument. The type name will be identical to the kernel_arg_type metadata, except for types derived from a single OpenCL built-in type (typedef). In this case the name of the OpenCL built-in type will be used.
`"kernel_arg_type_qual"`	string metadata	"const" "restrict" "volatile" "pipe" or a single space separated combination of these.
`"kernel_arg_name"`	string metadata	the name specified for the argument. Generated only when the -cl-kernel-arg-info build option is specified for compilation.

Table 7: Kernel Arg Info metadata description

Note: Images data types reside in global memory and hence should be marked as such in the `"kernel_arg_addr_space"` metadata.

Example:

```
typedef sampler_t mySampler;
__kernel void helloworld(__global char* in, __attribute__((nosvm)) __global char* out,
                                    mySampler s);

!opencl.kernels = !{!0}

!0 = metadata !{void (i8 addrspace(1)*, i8 addrspace(1)*, i32)* @helloworld, metadata !1,
                        metadata !2, metadata !3, metadata !4, metadata !5, metadata !6,
                        metadata !7}
!1 = metadata !{metadata !"kernel_arg_addr_space", i32 1, i32 1, i32 0}
!2 = metadata !{metadata !"kernel_arg_access_qual", metadata !"none", metadata !"none",
```

```
                              metadata !"none"}
!3 = metadata !{metadata !"kernel_arg_type", metadata !"char*", metadata !"char*",
                              metadata !"mySampler"}
!4 = metadata !{metadata !"kernel_arg_base_type", metadata !"char*", metadata !"char*",
                              metadata !"sampler_t"}
!5 = metadata !{metadata !"kernel_arg_type_qual", metadata !"", metadata !"", metadata !""}
!6 = metadata !{metadata !"kernel_arg_name", metadata !"in", metadata !"out", metadata !"s"}
!7 = metadata !{metadata !"kernel_arg_optional_qual", metadata !"none", metadata !"nosvm", metadata
```

2.5 Storage class specifier

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

2.6 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.)
pipe	opaque* (read more about it in the pipes built-ins section)

Table 8: 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.

2.7 Attribute Qualifiers

2.7.1 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.

2.7.1.1 aligned attribute

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

2.7.1.2 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.

2.7.2 Variable Attributes

2.7.2.1 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.

2.7.3 Loop unroll hint attribute

This metadata is a hint that the loop can be unrolled. The first operand is the string `opencl.loop.unroll_hint` and the second operand is an i32 positive value specifying the unroll factor. This metadata is attached to the existing `llvm.loop` identifier.

Notes:

- Setting `opencl.loop.unroll_hint` to 0 leaves the unroll factor to be chosen automatically
- Setting `opencl.loop.unroll_hint` to 1 is a hint to disable unrolling

Example:

```
br i1 %exitcond, label %._crit_edge, label %.lr.ph, !llvm.loop !0
...
!0 = metadata !{ metadata !0, metadata !1 }
!1 = metadata !{ metadata !"llvm.vectorizer.unroll", i32 2 }
```

2.8 Compiler Options

Compiler options are represented in SPIR using a named metadata node opencl.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

Note: The `-cl-std` option is propagated to the `opencl.ocl.version` as defined in Section 2.13, OpenCL Version.

This example indicates that both `-cl-mad-enable` and `-cl-denorms-are-zero` standard compile options were used to compile the module:
!opencl.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 `opencl.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 `opencl.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:
!opencl.compiler.ext.options = !{!5}
!5 = metadata !{metadata !"-opt-arch-pdp11"}

2.9 Preprocessor Directives and Macros

It is the SPIR generator's responsibility to handle all preprocessor responsibilities including macro substitution.

2.9.1 Floating point contractions

The named metadata `opencl.enable.FP_CONTRACT` can be used to enable contractions at module level. If the named metadata node exists, contractions can be generated by a SPIR optimizer at module level.

Note: This is a case where OpenCL C allows finer grained optimisation than SPIR, since it allows the selective enabling of floating point contraction for only certain calculations within a compilation unit.

SPIR can nevertheless express these programs. Since FP_CONTRACT only relaxes precision requirements, OpenCL C programs that use FP_CONTRACT selectively can still be safely and legally represented as more precise SPIR programs without FP_CONTRACT. However, such a program will not necessarily have the same performance or identical rounding and precision as the original on any particular platform.

2.10 Built-ins

2.10.1 Name Mangling

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

2.10.2 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, CLK_GLOBAL_MEM_FENCE = 2 and CLK_IMAGE_MEM_FENCE=4.

2.10.3 Order and Consistency

`memory_order` enumeration constants are mapped to `i32` constants as follows:

Memory order	**Value**
memory_order_relaxed	0x0
memory_order_acquire	0x1
memory_order_release	0x2
memory_order_acq_rel	0x3
memory_order_seq_cst	0x4

Table 9: memory order values

2.10.4 Memory Scope

`memory_scope` enumeration constants are mapped to `i32` constants as follows:

Memory scope	**Value**
memory_scope_work_item	0x0
memory_scope_work_group	0x1
memory_scope_device	0x2
memory_scope_all_svm_devices	0x3
memory_scope_sub_group	0x4

Table 10: memory scope values

2.10.5 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`.

In SPIR the conversion specifiers e,E,g,G,a,A require a double type argument to be passed to the function printf. Thus a `float` or `half` argument that is a scalar type should be explicitly converted to a `double`. A device that doesn't support the `double` data type shall disregard this explicit conversion, or replace the conversion with a conversion to a `float` data type in the case of a `half` data type argument.

Note:

- The presence of this conversion alone is not enough to force the listing of `"cl_doubles"` as a "used optional core features" for this SPIR instance.
- The `format` argument is in constant address space and must be resolvable at compile time. Hence, it cannot be dynamically created by the executing program itself.

2.10.6 Enqueuing Kernels

2.10.6.1 Macros values

The following table describes the mapping between the device-side enqueue related Macros and SPIR `i32` literal values

Macro	**Value**
CLK_NULL_EVENT	llvm `null`
CLK_NULL_QUEUE	llvm `null`
CLK_SUCCESS	0x0
CLK_ENQUEUE_FAILURE	-101
CLK_INVALID_QUEUE	-102
CLK_INVALID_NDRANGE	-160
CLK_INVALID_EVENT_WAIT_LIST	-57
CLK_DEVICE_QUEUE_FULL	-161
CLK_INVALID_ARG_SIZE	-51
CLK_EVENT_ALLOCATION_FAILURE	-100
CLK_OUT_OF_RESOURCES	-5
CL_COMPLETE	0
CL_SUBMITTED	2

Table 11: Enqueuing Kernels Macros Values

2.10.6.2 Kernel enqueue flags

`kernel_enqueue_flags_t` enumeration constants are mapped to `i32` constants as follows:

Flag	**Value**
CLK_ENQUEUE_FLAGS_NO_WAIT	0x0
CLK_ENQUEUE_FLAGS_WAIT_KERNEL	0x1
CLK_ENQUEUE_FLAGS_WAIT_WORK_GROUP	0x2

Table 12: Kernel enqueue flags

Note: `kernel_enqueue_flags_t` arguments to built-in functions are treated as `i32` arguments for mangling purposes.

2.10.6.3 clk_profiling_info

`clk_profiling_info` enumeration constants are mapped to `i32` constants as follows:

Flag	**Value**
CLK_PROFILING_COMMAND_EXEC_TIME	1

Table 13: clk_profiling_info

Note: `clk_profiling_info` arguments to built-in functions are treated as `i32` arguments for mangling purposes.

2.10.7 Pipe functions

The signatures of all built-in functions described in Section 6.13.16.2.2-4 are modified in the following way:

- Pipe objects are represented using pointers to the opaque `%opencl.pipe` LLVM structure type which reside in the global address space.
- If a function accepts a pipe object, two additional `i32` arguments are appended to the end of the signature. The first argument represents the size in bytes of each packet in the pipe object. The second argument represents the alignment in bytes of each packet in the pipe object. These two argument must take a literal i32 value.
- The input/output `ptr` argument to the read_pipe and write_pipe built-ins is cast into a pointer to `i8` in the generic address space.

Example:

```
The OpenCL "C" example:
kernel void test_reserved_read_pipe(global int *Dst, read_only pipe int OutPipe) {
  read_pipe (OutPipe, Dst);
}

is represented in SPIR as follows:
%opencl.pipe_t = type opaque

define void @test_reserved_read_pipe(i32 addrspace(1)* %Dst,
                    %opencl.pipe_t addrspace(1)* %OutPipe) nounwind {
entry:
  %0 = addrspacecast i32 addrspace(1)* %Dst to i8 addrspace(4)*
  %1 = call i32 @_Z9read_pipePU3AS18ocl_pipePU3AS4vjj(
            %opencl.pipe_t addrspace(1)* %OutPipe,
            i8 addrspace(4)* %0, i32 4, i32 4)
  ret void
}

declare i32 @_Z9read_pipePU3AS18ocl_pipePU3AS4vjj(%opencl.pipe_t addrspace(1)*,
                                                  i8 addrspace(4)*, i32, i32)
```

Note:

- `CLK_NULL_RESERVE_ID` is mapped to llvm `null`
- Pipe built-ins are mangled based on the SPIR prototype.

2.10.8 Address Space Qualifier functions

The signatures of the address space qualifier built-in functions described in Section 6.13.9 are modified in the following way:

- The `gentype*` and `const gentype*` arguments and return values are replaced with `void*` and `const void*` respectively.
- The input `ptr` argument to these built-in functions is cast into a pointer to `i8` in the generic address space.

- The return value of the `to_global`, `to_local` and `to_private` built-in functions is cast back into a pointer to `gentype*` in the appropriate named address space.

Example:

```
The OpenCL "C" example:
kernel void foo(local int* lpi) {
  int *pi = lpi;
  local int *lpi2 = to_local(pi);
  ...
}

is represented in SPIR as follows:
define void @foo(i32 addrspace(3)* %lpi) #0 {
  ...
  %2 = load i32 addrspace(4)** %pi, align 4
  %3 = bitcast i32 addrspace(4)* %2 to i8 addrspace(4)*
  %call = call i8 addrspace(3)* @_Z8to_localPU3AS4v(i8 addrspace(4)* %3)
  %4 = bitcast i8 addrspace(3)* %call to i32 addrspace(3)*
  store i32 addrspace(3)* %4, i32 addrspace(3)** %lpi2, align 4
  ...
  ret void
}
```

2.11 KHR Extensions

2.11.1 Declaration of used optional core features

The named metadata object `opencl.used.optional.core.features` contains a single metadata object. The metadata object should contain a list of metadata 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.

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

2.11.2 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 `opencl.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
- cl_khr_gl_msaa_sharing
- cl_khr_depth_images
- cl_khr_gl_depth_images
- cl_khr_subgroups
- cl_khr_mipmap_image
- cl_khr_mipmap_image_writes
- cl_khr_egl_event
- cl_khr_srgb_image_writes

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

```
!opencl.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 `opencl.used.optional.core.features`.
- **cl_khr_fp64** doesn't exist in SPIR. Instead SPIR generators should use the `cl_doubles` optional core features.

2.12 SPIR Version

The SPIR version used by the module is stored in the `opencl.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 2.0:

```
!opencl.spir.version = !{!3}
!3 = metadata !{i32 2, i32 0}
```

2.13 OpenCL Version

The OpenCL version used by the module is stored in the `opencl.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 1.2:

```
!opencl.ocl.version = !{!4}
!4 = metadata !{i32 1, i32 2}
```

This example indicates the module is compiled for OpenCL 2.0:

```
!opencl.ocl.version = !{!4}
!4 = metadata !{i32 2, i32 0}
```

2.14 memcpy functions

The usage of LLVM memcpy and memset intrinsics is allowed in SPIR

2.15 Restrictions

Restrictions from OpenCL C also apply to programs represented in SPIR.

Also, recall that use of FP_CONTRACT is encoded at the module level. See Section 2.9.1 for a discussion of how this limits what OpenCL programs may be represented in SPIR.

3 SPIR and LLVM IR

3.1 LLVM Triple

SPIR introduces a couple of new LLVM triples called "`spir-unknown-unknown`" and "`spir64-unknown-unknown`"

```
target triple = "spir-unknown-unknown"
target triple = "spir64-unknown-unknown"
```

"`spir`" targets devices with address width of 32 bits. "`spir64`" targets devices with address width of 64 bits.

3.2 LLVM Target data layout

The `spir` triple datalayout is as follows:

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

The `spir64` triple datalayout is as follows:

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

3.3 LLVM Supported Identifiers

The name of named global identifiers must be in the form of the following regular expression:

`[%@][a-zA-Z$._][a-zA-Z$._0-9]*`

The following prefixes are reserved:

- llvm.*
- spir.*
- opencl.*

Note: The prefix `spir.internal.*` is reserved for internal use of SPIR consumers.

3.4 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
Terminator	invoke	no
Terminator	unwind	no
Terminator	resume	no
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 14: 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	yes
Conversion Operations	inttoptr .. to	yes
Conversion Operations	bitcast .. to	yes
Conversion Operations	addrspacecast .. to	yes
Other Operations	icmp	yes
Other Operations	fcmp	yes
Other Operations	phi	yes
Other Operations	select	yes
Other Operations	call	yes (including compile time resolvable pointers to functions)
Other Operations	va_arg	no, not supported by OpenCL
Other Operations	landingpad_arg	no

Table 15: Instructions, part 2

3.5 LLVM Supported Intrinsic Functions

None of the LLVM intrinsics are allowed in SPIR except the memcpy intrinsics.

3.6 SPIR ABI

In this section we define the application binary interface for OpenCL "C" programs in SPIR. The SPIR ABI defines the interfaces between the SPIR program and the OpenCL runtime, built-ins libraries and additional third party SPIR libraries.

Each function argument and return type is classified as follows:

- A return value which is an aggregate type is promoted to be the first argument of the function.
- Any aggregate type is passed as a pointer. Memory allocation (if needed) is the responsibility of the caller function.
- Enumeration types are handled as the underlying integer type.
- If the argument type is a promotable integer type, it will be extended according to the C99 integer promotion rules.
- Any other type, including floating point types, vectors, etc.. will be passed directly as the corresponding LLVM type.

Note: The ABI described in this section is implemented in Clang 3.xx and is called the "default" ABI.

3.7 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
internal	yes (maps to static)
available_externally	yes (describes C99 inline definition)
linkonce	no
weak	no
common	yes
appending	no
extern_weak	no
linkonce_odr	no
weak_odr	no
external	yes
dllimport	no
dllexport	no

Table 16: Linkage types

In addition, SPIR allows the usage of LLVM `unnamed_addr` optional attribute for both global variables and functions.

3.8 Calling Conventions

SPIR kernels should use `"spir_kernel"` calling convention. Non-kernel functions use `"spir_func"` calling convention. All other calling conventions are disallowed.

3.9 Visibility Styles

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

3.10 Parameter Attributes

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

Parameter Attribute	Supported
zeroext	yes
signext	yes
inreg	no
byval	yes
inalloca	no
sret	yes
noalias	yes
nocapture	yes
nest	no
returned	no

Table 17: Parameter attributes

3.11 Garbage Collection Names

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

3.12 Prefix Data

Prefix data is not part of SPIR

3.13 Attribute Groups

?

3.14 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
builtin	?
cold	? (yes)
inlinehint	yes
minsize	? (no)
naked	no
nobuiltin	?
noduplicate	yes, relevant for workgroup functions - how do we want the specifciation to address this
noimplicitfloat	no
noinline	yes
nonlazybind	no
noredzone	no
noreturn	no
nounwind	yes, needs to be always set
optsize	no
readnone	yes
readonly	yes
returns_twice	no
sanitize_address	? (no)
sanitize_memory	? (no)
sanitize_thread	? (no)
ssp	no
sspreq	no
sspstrong	? (no)
uwtable	no

Table 18: Function attributes

3.15 Module Level Inline Assembly

LLVM module level inline assembly is not allowed in SPIR.

3.16 Pointer Aliasing Rules

SPIR follows the pointer aliasing rules of LLVM.

3.17 Volatile Memory Accesses

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

3.18 Memory Model for Concurrent Operations

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

3.19 Atomic Memory Ordering Constraints

The LLVM atomic orderings are disallowed in SPIR.

3.20 Fast-Math Flags

The LLVM IR floating-point binary ops fast-math flags are allowed in SPIR.

3.21 Poison Values

Poison values are disallowed in SPIR.

A SPIR name mangling

In order to support cross device compatibility of SPIR, the name mangling scheme must be standardized 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, OpenCL C built-in functions must use this mangling scheme.

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
pointer to private address space	P<*mangled-element-type-name*>
pointer to non private address space	PU3AS*N*<*mangled-element-type-name*> (where N is the address space number)
Vector types with N elements	Dv*N*_<*mangled-element-type-name*> (where N is one of 2, 3, 4, 8, 16)
image1d_t	11ocl_image1d
image1d_array_t	16ocl_image1darray
image1d_buffer_t	17ocl_image1dbuffer
image2d_t	11ocl_image2d
image2d_array_t	16ocl_image2darray
image3d_t	11ocl_image3d
image2d_msaa_t	15ocl_image2dmsaa
image2d_array_msaa_t	20ocl_image2darraymsaa
image2d_msaa_depth_t	20ocl_image2dmsaadepth
image2d_array_msaa_depth_t	25ocl_image2darraymsaadepth
image2d_depth_t	16ocl_image2ddepth
image2d_array_depth_t	21ocl_image2darraydepth
event_t	9ocl_event
sampler_t	11ocl_sampler
size_t, uintptr_t	treated as uint or ulong
ptrdiff_t, intptr_t	treated as int or long
atomic_*	U7_Atomic<*mangled-element-base-type-name*>
N/A	8ocl_pipe
reserve_id_t	13ocl_reserveid
queue_t	9ocl_queue
ndrange_t	9ndrange_t
clk_event_t	12ocl_clkevent
Block	U13block_pointerF<ret type><arg type1>...<arg typeN>E

Table 19: Mapping of OpenCL C builtin type names to mangled type names

A.2 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.3 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))
          ::= P<builtin-type> # A pointer to private address space.
          ::= PU3ASN<builtin-type> # A pointer to address space 'N' (non-private).
                                   # Only values of 1, 2, 3 and 4 are valid.
          ::= DvN_<builtin-type> # An OpenCL vector of length 'N' of the specified type.
                                   # Only values of 2, 3, 4, 8 and 16 are valid.
          ::= 11ocl_image1d # A 1d image type
          ::= 16ocl_image1darray # A 1d image array type
          ::= 17ocl_image1dbuffer # A 1d image buffer type
          ::= 11ocl_image2d # A 2d image type
          ::= 16ocl_image2darray # A 2d image array type
          ::= 11ocl_image3d # A 3d image type
          ::= 15ocl_image2dmsaa
          ::= 20ocl_image2darraymsaa
          ::= 20ocl_image2dmsaadepth
```

```
::= 25ocl_image2darraymsaadepth
::= 16ocl_image2ddepth
::= 21ocl_image2darraydepth
::= 9ocl_event # A event type
::= 11ocl_sampler # A sampler type
::= U7_Atomic<builtin-type> # An atomic type
::= 8ocl_pipe # A pipe object type
::= 13ocl_reserveid # A reserve_id_t object type
::= 8ocl_clkevent # A clk_event_t type
::= U13block_pointerF<ret type><arg type1>...<arg typeN>E # Block argument type
::= 9ocl_queue # A queue_t object type
::= u <source-name> # vendor extended type
```

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

```
<CV-qualifiers> ::= [r] [V] [K]   # restrict (C99), volatile, const
                                  # 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 2.0 — OpenCL C. http://www.khronos.org/registry/cl/specs/opencl-2.0-openclc.pdf, November 2013.

[3] LLVM Team. LLVM Bitcode File Format. http://www.llvm.org/releases/3.4/docs/BitCodeFormat.html, January 2014. Version 3.4.

[4] LLVM Team. LLVM Language Reference Manual. http://www.llvm.org/releases/3.4/docs/LangRef.html, January 2014. Version 3.4.