# Vector Function Application Binary Interface Specification for AArch64

*Release 2Q2018*

**Arm Ltd**

**Jun 26, 2018**

# CONTENTS

CHAPTER

# ONE

# PREAMBLE

## 1.1 Abstract

The *Vector Function Application Binary Interface Specification for AArch64* describes the application binary interface for vector functions generated by a compiler.

This document uses the OpenMP[1] `declare simd` directive to classify the vector functions that can be associated to a scalar function.

The following rules apply to a compiler that implements the OpenMP directive `#pragma omp declare simd`:

1. The use of a `#pragma omp declare simd` construct for a *function definition* enables the creation of vector versions of the function from the scalar version of the function, that can be used to process multiple instances concurrently in a single invocation in a vector context (e.g. vectorized loops).
2. The use of a `#pragma omp declare simd` construct for a *function declaration* enables the compiler to know the exact list of available vector function implementations provided by a library that is based on the OpenMP pragmas found in the function's prototype of the library headers.

This Vector Function ABI defines a set of rules that the caller and the callee functions must obey.

## 1.2 Keywords

| | |
|---|---|
| **SVE** | Scalable Vector Extension |
| **A64** | Instruction set of the ARMv8-A architecture |
| **AArch64** | 64-bit execution mode of the ARMv8-A architecture |
| **Advanced SIMD** | SIMD and floating point instructions of the A64 instruction set |
| **Qn register** | Quad-word (128-bit) floating point register |
| **Dn register** | Double-word (64-bit) floating point register |
| **Sn register** | Single-word (32-bit) floating point register |
| **Hn register** | Half-word (16-bit) floating point register |
| **Bn register** | Byte (8-bit) floating point register |
| **Vn register** | Advanced SIMD vector register |
| **Zn register** | SVE vector register |
| **ACLE** | ARM C Language Extensions |
| **SVE ACLE** | ARM C Language Extensions for SVE |
| **Vector function** | A function processing vector data through the SIMD registers |

[1] http://www.openmp.org/

**Leaf function** Function at the end of a call tree

**AAPCS** ARM Architecture Procedure Call Standard

**OpenMP** Open Multi-Processing standard

**Uniform parameter** A function parameter marked with the OpenMP *uniform* clause

**Linear parameter** A function parameter marked with the OpenMP *linear* clause

**LP64** Data model in which Long and Pointers are 64-bit

**ILP32** Data model in which Integer, Long and Pointers are 32-bit

**VLA** Vector Length Agnostic

**VLS** Vector Length Specific

**`MTV(P)`** `P` Maps To Vector

**`PBV(P)`** `P` is Passed By Value

**`LS(P)`** Lane Size of `P`

**`MAP(P)`** Mapping of `P`

**`ADVSIMD_MAP(P)`** Mapping of `P` - Advanced SIMD specific rules.

**`SVE_MAP(P)`** Mapping of `P` - SVE specific rules.

**`NDS(f)`** Narrowest Data Size of `f`

**`WDS(f)`** Widest Data Size of `f`

## 1.3 Latest release and defects report

Please check the ARM Developer website[2] for the latest release of this document if this copy is more than one year old.

Please report defects in this specification to arm dot eabi at arm dot com.

## 1.4 Licence

## 1.5 Non-Confidential Proprietary Notice

[2] http://developer.arm.com/
[3] https://developer.arm.com/docs/ihi0037/latest/base-platform-abi-for-the-arm-architecture
[4] https://developer.arm.com/docs/ihi0037/latest/base-platform-abi-for-the-arm-architecture

CHAPTER

# TWO

# DEFINITIONS

## 2.1 Vector Procedure Call Standard

AArch64 functions use the calling convention described in section 5 of the *Procedure Call Standard for the ARM 64-bit Architecture (with SVE support)*, or AAPCS hereafter. The most recent version of the AAPCS can be found on developer.arm.com[5].

The AAPCS defines two conventions: one convention for functions that take SVE arguments in registers or return SVE results in registers (the *SVE convention*) and one convention for all other functions (the *base convention*).

**Note:** The SVE-specific rules of the AAPCS are in beta version. The list of SVE call-clobbered and call-preserved registers in table *Modified PCS for vector functions (AAVPCS)* (page 4) will be updated when the final version of the AAPCS is published.

The base convention requires that all 32 Advanced SIMD vector registers are treated as call-clobbered, thus requiring the caller to perform up to 32 vector stores before a call and up to 32 vector loads after it. Vector math functions are typically leaf functions which require a small number of registers. Thus, to efficiently support such vector math routines, we define a modified version of the base convention for Advanced SIMD vector routines, called the *Vector Procedure Call Standard for the Arm 64-bit Architecture (AAVPCS)*. The list of parameter, result, call-perserved and call-clobbered registers for the AAVPCS are presented in the following table:

Table 1: Modified PCS for vector functions (AAVPCS)

| Extension | Parameter and Result registers | Call-clobbered registers | Call-preserved registers |
|---|---|---|---|
| Advanced SIMD | V0-V7 | V0-V7, V24-V31 | V8-V23 |
| SVE | Z0-Z7 | See AAPCS | |

The AAVPCS is implicit when a `#pragma omp declare simd` clause is attached to a function definition or declaration. For user-defined Advanced SIMD or SVE vector functions, the same behavior can be obtained by adding the `aarch64_vector_pcs` function attribute to the function definition or declaration as in the following examples. Note that to ensure the compiler produces ABI consistent code, the attribute must be specified in every declaration and definition of the function.

```
/* function definition */
__attribute__((aarch64_vector_pcs))
uint64x2_t foo(uint32x2_t a, float32x2_t b) {
  /* function body */
}
/* function declaration */
__attribute__((aarch64_vector_pcs)) float64x2_t bar(float64x2_t a);
```

[5] https://developer.arm.com/docs/100986/latest/procedure-call-standard-for-the-arm-64-bit-architecture-aarch64-with-sve-support

## 2.2 Extended Vector Notation for Advanced SIMD

For the purposes of this specification, we define the following notational extensions for the Advanced SIMD vector types defined by the AAPCS64. These types are not made available to the user.

### 2.2.1 Padded Short Vectors

Padded short vectors extend the definition of short vectors and are used as a notational convenience to describe vector types with a size of less than 64 bits. These can be formed where the `simdlen` clause specified in an OpenMP `declare simd` construct would force a smaller vector than would meet the AAPCS definition of a short vector. These have the form of a vector with <N> elements of type <T>:

```
<T>x<N>_t
```

Where

```
sizeof(<T>) * <N> < 8
```

A padded short vector is represented as an 8-byte short vector type with elements of type <T> in which lanes <N> and above have unspecified values. For example, a padded short vector `uint16x2_t` is represented as a `uint16x4_t` in which lanes 2 and 3 have unspecified values.

The contents of the 8-byte vector are arranged as though the whole padded short vector were a single lane. For example, a `uint16x2_t` is stored in the `uint16x4_t` as though it were lane 0 in a `uint32x2_t`.

**Note:** When a padded short vector is transferred between registers and memory it is treated as an opaque object of the notional type. That is, a padded short vector is stored in memory as if it were stored with a single STR of an object of the size of the notional type of the padded short vector; a padded short vector is loaded from memory using the corresponding LDR instruction. On a little-endian system this means that element 0 will always contain the lowest addressed element of a padded short vector; on a big-endian system element 0 will contain the highest-addressed element of a padded short vector.

This is shown in the following table,

Table 2: Big-endian vs Little-endian padded vectors.

| Padded short vector yype | Short vector type | Little-endian | Big-endian |
|---|---|---|---|
| `[u]int8x2_t` | `[u]int8x8_t` | `X|X|X|X|X|X|A[1]|A[0]` | `X|X|X|X|X|X|A[0]|A[1]` |
| `[u]int8x4_t` | `[u]int8x8_t` | `X|X|X|X|A[3]|...|A[0]` | `X|X|X|X|A[0]|...|A[3]` |
| `float16x2_t` | `float16x4_t` | `X|X|A[1]|A[0]` | `X|X|A[0]|A[1]` |

The set of padded short vector types, the short vector type they map to, and the appropriate store width for each type is given in the following table,

Table 3: Padded short vectors

| Padded short vector type | Short vector type | **LDR/STR** registers |
|---|---|---|
| `[u]int8x1_t` | `[u]int8x8_t` | Bn |
| `[u]int8x2_t` | `[u]int8x8_t` | Hn |
| `[u]int8x4_t` | `[u]int8x8_t` | Sn |
| `[u]int16x1_t` | `[u]int16x4_t` | Hn |
| `[u]int16x2_t` | `[u]int16x4_t` | Sn |
| `float16x1_t` | `float16x4_t` | Hn |
| `float16x2_t` | `float16x4_t` | Sn |
| `float32x1_t` | `float32x2_t` | Sn |

When using a padded short vector, the contents of the elements of the associated short vector that lie outside the padded short vector are undefined.

Where padded short vectors are used, this may cause the compiler to emit conservative, scalar code to process their content.

No language bindings are provided for padded short vectors. Padded short vectors are not generated for declare simd constructs with no simdlen clause.

### 2.2.2 Extended short vectors

Extended short vectors extend the AAPCS definition of short vectors and are used as a notational convenience to describe vector types with a size greater than 128 bits. These can be formed where the required vectorization factor would create a larger vector than would meet the AAPCS definition of a short vector. These have the form:

```
<T>x<N>_t
```

Where

```
sizeof(<T>) * <N> > 16
```

Extended short vectors are represented as a structure containing an array of short vectors of the appropriate type. These have the general form:

```
struct <T>x<NN>x<M>_t { <T>x<NN>_t val[<M>]; };
```

Where <NN> is such that <N>=<NN> * <M>.

A subset of the possible vector types are given in the following table,

Table 4: Extended short vector examples

| Notional type | Parameter/Return type |
|---|---|
| int32x16_t | struct int32x4x4_t { int32x4_t val[4]; }; |
| float64x4_t | struct float64x2x2_t { float64x2_t val[2]; }; |
| int32x16_t | struct int32x4x4_t { int32x4_t val[4]; }; |

No language bindings are provided for extended short vectors, though some of these types are also defined by `arm_neon.h`.

## 2.3 SVE unpacked vector

Let `sv<T>_t` be an SVE ACLE vector type with lanes of type <T>. The vector is said to be *unpacked* if only the logical lanes corresponding to the multiples of some power of 2 can be set active by a `svbool_t` predicate. Conversely, the vector is said to be *packed* if any lane can be active.

For example, 32-bit signed integers from a reference `int32_t * A` can be loaded into an unpacked `svint32_t` vector at lanes 0, 2, 4,... and so on, effectively using only half of the lanes available in the vector. In the following example, the resulting SVE packed vector is shown together with two unpacked versions (X is for undefined content):

```
lane idx 8    | 7    | 6    | 5    | 4    | 3    | 2    | 1    | 0
[msb]     ... | A[7] | A[6] | A[5] | A[4] | A[3] | A[2] | A[1] | A[0] [lsb]  // packed
[msb]     ... | X    | A[3] | X    | A[2] | X    | A[1] | X    | A[0] [lsb]  // unpacked 0, 2, 4, ...
[msb]     ... | X    | X    | X    | A[1] | X    | X    | X    | A[0] [lsb]  // unpacked 0, 4, 8, ...
```

CHAPTER

# THREE

# VECTOR FUNCTION SIGNATURE

This section describes how the scalar functions decorated with the OpenMP `declare simd` pragma are associated to vector function signatures.

When vectorizing the following loop, whatever vectorization factor we choose, we want to make sure that the compiler expects a vector version of `f` and `g` that operates on the same number of lanes.

```
float f(double);
double g(float);
float x[];
//...
for (int i = 0; i < 100; ++i)
  x[i] = f(g(x[i]));
```

The rules given in this chapter guarantee that any `#pragma omp declare simd` attached to a *function declaration* or *definition* generates a unique set of vector functions associated to the original scalar function. This is done to make sure that library vendors can provide a unique way to interface the routines of the library with a compiler, by means of the `declare simd` directive.

In all cases, the order of the vector function parameters reflects the ordering of the parameters of the original scalar function.

Throughout this chapter, `f` is a function declaration or definition decorated with an OpenMP *declare simd* directive, `<P>` is the return value or an input parameter of `f`, and `<T(P)>` is its associated type.

## 3.1 Common rules for parameter mapping

One or more vector functions `F` are associated to the original scalar function `f`. The return value and each function parameter is mapped to a unique return value or input parameter respectively, named *Mapping* of `P`, or `MAP(P)`. The type of these vector function return value and input parameters depends on the following rules. Their order is the same as in the original scalar function `f`.

### 3.1.1 Maps To Vector

To each `<P>`, a `true` / `false` predicate "`P` *Maps To Vector*", or `MTV(P)` hereafter, is associated as follows:

1. If `<P>` is an input parameter such that:

   (a) `<P>` is a `uniform` value, or

   (b) `<P>` is a `linear` value and not a reference marked with `val` or no linear modifiers,

   then `MTV(P)` is `false`.

2. If `P` is a `void` return value, then `MTV(P)` is `false`;
3. In all other cases, `MTV(P)` is `true`.

### 3.1.2 Pass By Value

When a scalar parameter maps to a vector, that vector sometimes contains the *values* of the scalar parameters and sometimes contains the *addresses* of the scalar parameters. The predicate *Pass by Value* `PBV(T)` is `true` if the former case applies for scalar parameters of type `T`; it is `false` if the latter case applies. The predicate is defined as follows:

1. `PBV(T)` is `true` if (a) `T` is an integer, floating-point or pointer type and (b) `sizeof(T)` is `1`,"2", `4` or `8`.
2. `PBV(T)` is `true` if `T` is a complex type with components of type `T'` and if `PBV(T')` is `true`.
3. Otherwise `PBV(T)` is false.

### 3.1.3 Paramater mapping

When mapping the return value or an input parameter <P> of the scalar function to the corresponding `MAP(P)` in the vector function, the following rules apply:

1. If `MTV(P)` is `false`, then `MAP(P)` is `P`.
2. Otherwise, if `MTV(P)` is `true`, then `MAP(P)` is target specific:
   (a) For Advanced SIMD, `MAP(P) = ADVSIMD_MAP(P)`, with `ADVSIMD_MAP(P)` defined in section *Advanced SIMD specific rules* (page 9).
   (b) For SVE, `MAP(P) = SVE_MAP(P)`, with `SVE_MAP(P)` defined in section *SVE specific rules* (page 9).

## 3.2 Vector length selection

A set of vector lengths `VLEN` is sometimes associated with the generated vector function `F`. When this is done, the algorithm for selecting the value(s) of `VLEN` is target dependent. The algorithm makes use of the definitions in this section.

### 3.2.1 Lane Size of a function parameter / return value

We then define the *Lane Size of P*, or `LS(P)`, as follows.

1. If `MTV(P)` is `false` and `P` is a pointer or reference to some type `T` for which `PBV(T)` is `true`, `LS(P) = sizeof(T)`.
2. If `PBV(T(P))` is `true`, `LS(P) = sizeof(P)`.
3. Otherwise `LS(P) = sizeof(uintptr_t)`.

### 3.2.2 Narrowest and Widest Data Size of a Function

For the function `f`, we define the following concepts:

1. The *Narrowest Data Size of f*, or `NDS(f)`, as the minumum of the lane size `LS(P)` among all input parameters and return value <P> of `f`.
2. The *Widest Data Size of f*, or `WDS(f)`, as the maximum of the lane size `LS(P)` among all input parameters and return value <P> of `f`.

Note that by definition the value of `NDS(f)` and `WDS(f)` can only be 1, 2, 4, 8, and 16.

## 3.3 Advanced SIMD specific rules

This section describes the Advanced SIMD specific rules for mapping <P> to its corresponding vector parameter `MAP(P)` when `MTV(P) = true`.

### 3.3.1 Vector Length

A `VLEN` is always associated with the vector function. The rules to generate the set of the available values are:

1. If `simdlen(len)` is specified, then the compiler generates only one version with `VLEN = len`. The value of `vlen` must be a power of 2.
2. If no `simdlen` is specified, the compiler generates multiple versions, according to the following rules:
   (a) if `NDS(f) = 1`, then `VLEN = 16, 8`;
   (b) if `NDS(f) = 2`, then `VLEN = 8, 4`;
   (c) if `NDS(f) = 4`, then `VLEN = 4, 2`;
   (d) if `NDS(f) = 8` or `NDS(f) = 16`, then `VLEN = 2`.

### 3.3.2 Parameter mapping

For a value of `VLEN`, the `ADVSIMD_MAP(P)` is build as follows:

1. If `PBV(T(P))` is `false`, `ADVSIMD_MAP(P)` is a vector of `VLEN` elements of type `uintptr_t`.
2. If `T(P)` is a complex type with components of type `T`, `MAP(P)` is a vector of `2*VLEN` elements of type `T`.
3. Otherwise `ADVSIMD_MAP(P)` is a vector of `VLEN` elements of type `T(P)`.
4. An optional `{not}inbrach` clause defines whether or not a vector mask parameter is added as the last input parameter of `F`, according to the rules in table 1 in chapter 4. The vector mask type is selected by building a vector of `VLEN` elements consisting of unsigned integers of `NDS(f)` bytes. The generation of the values in the mask parameter is described in section 4.1.

## 3.4 SVE specific rules

This section describes the SVE-specific rules for mapping <P> to its corresponding vector parameter `MAP(P)` when `MTV(P) = true`.

*One* vector function `F` is associated to `f` depending on its classification via the `declare simd` directive. The vector signatures that get generated are the same in all cases.

### 3.4.1 Vector Length

1. If no `simdlen` clause is specified, a VLA vector version is associated.
2. When using a `simdlen(len)` clause, the compiler expects a VLS vector version of the function that is tuned for a specific implementation of SVE. The size of the implementation is `WDS(f)* len * 8`.

### 3.4.2 Parameter mapping

Whether targeting VLA SVE or VLS SVE, the rules for mapping <P> to `SVE_MAP(P)` are:

1. If `PBV(T(P))` is `false`, `SVE_MAP(P)` is a scalable vector of `uintptr_t`.
2. If `T(P)` is a complex type with components of type `T`, `SVE_MAP(P)` is a scalable vector of `T`.
3. Otherwise `SVE_MAP(P)` is a scalable vector of `T(P)`.
4. An additional `svbool_t` mask parameter is added as the last parameter of `F`. The generation of the mask values is described in section 4.2.

### 3.4.3 Unpacked parameters / return value

The vectors of the signature of `F` are packed or unpacked according to the following rules:

1. if `LS(P) = WDS(f)`, then the vector is packed.
2. If `LS(P) < WDS(f)`, then the vector is unpacked.

Each element in the unpacked vector occupies the same number of bits as in the packed vector, and all elements are aligned to their least significant bits.

The following example shows the contents of an SVE vector consisting of 1-byte lanes, unpacked and aligned with the 4-byte lanes of a packed vector. The ?? characters indicate a byte whose value is undefined.

```
Zn.b [msb] ... 0x??????03 0x??????02 0x??????01 0x??????00 [lsb]
Zn.s [msb] ... 0x00000003 0x00000002 0x00000001 0x00000000 [lsb]
```

## 3.5 Vector function name mangling

The vector function name mangling for the *#pragma omp declare simd* directive is defined as follows:

Listing 1: Name mangling rules for vector functions.

```
<vector name> := <prefix>  "_" <name>

<name> := Assembly name of the function

<prefix> := "_ZGV" <isa> <mask> <len> <parameters>

<isa> := "n"   (Advanced SIMD)
       | "s"   (SVE)

<mask> := "N" (No Mask)
        | "M" (Mask)

<len> := VLEN (VLS SVE or Advanced SIMD)
        | "x" (VLA SVE)

<parameters> := <parameter> { <parameter> }

<parameter> := <parameter_kind> [ "a" <X> ]

                                       OpenMP version
                                       support (onwards)
<parameter_kind> := "v"                   4.0
                  | "l" | "l" <number>    4.0
                  | "R" | "R" <number>    4.5
```

(continues on next page)

(continued from previous page)

```
                  | "L" | "L" <number>    4.5
                  | "U" | "U" <number>    4.5
                  | "ls" <pos>            4.5
                  | "Rs" <pos>            4.5
                  | "Ls" <pos>            4.5
                  | "Us" <pos>            4.5
                  | "u"                   4.0

<number> := "n" <X>  // "n" means negative
          | <Y>

<pos> := <X>

<X> := decimal number greater than or equal to 1
<Y> := decimal number greater than or equal to 2
```

We generate as many parameters as are in the original function according to the following rules:

**"v"** Vector parameter - default for no linear/uniform clause.

**"l" | "l" <number>** Linear parameter <P> for which (a) the step is a compile-time constant, (b) `MTV(P)=false` and (c) the linear clause has either a *val* modifier or no modifier. `<number>` is the value of the constant linear step, or an empty string if the step is 1. For example, `linear(i:2)` gives `l2` and `linear(i:1)` gives `l`.

**"R" | "R" <number>** Linear parameter <P> for which (a) the step is a compile-time constant, and (b) the linear clause has a *ref* modifier. `<number>` is the value of the constant linear step, or an empty string if the step is 1. For example, `linear(ref(i):3)` gives `R3` and `linear(ref(i):1)` gives `R`.

**"L" | "L" <number>** Linear parameter <P> for which (a) the step is a compile-time constant, (b) `MTV(P)=true` and (c) the linear clause has either a *val* modifier or no modifier. `<number>` is the value of the constant linear step, or an empty string if the step is 1. For example, `linear(val(i):-3)` gives `Ln3`.

When the parameter <P> marked by the linear clause is a pointer or an OpenMP integral reference to a type `T`, the `step` of the linear clause must be multiplied by the size in bytes of the pointee, so that `<number>=sizeof(T) x step`.

**"U" | "U" <number>** Linear parameter <P> for which (a) the step is a compile-time constant, and (b) the linear clause has a *uval* modifier. `<number>` is the value of the constant linear step, or an empty string if the step is 1. For example, `linear(uval(i):2)` gives `U2`.

**"ls" <pos>** Linear parameter <P> for which (a) the step is a loop-independent runtime invariant, (b) `MTV(P)=false` and (c) the linear clause has either a *val* modifier or no modifier. `<pos>` is the position of the step parameter specified in the uniform clause (required by the OpenMP specs). For example, `linear(i:c) uniform(c)` with `c` being the third parameter gives `ls3`.

**"Rs" <pos>** Linear parameter <P> for which (a) the step is a loop-independent runtime invariant and (b) the linear clause has a *ref* modifier. `<pos>` is the position of the step parameter specified in the uniform clause (required by the OpenMP specs). For example, `linear(ref(i):c) uniform(c)` with `c` being the third parameter gives `Rs3`.

**"Ls" <pos>** Linear parameter <P> for which (a) the step is a loop-independent runtime invariant, (b) `MTV(P)=true` and (c) the linear clause has either a *val* modifier or no modifier. `<pos>` is the position of the step parameter specified in the uniform clause (required by the OpenMP specs). For example, `linear(val(i):c) uniform(c)` with `c` being the first parameter, gives `Ls1`.

**"Us" <pos>** Linear parameter <P> for which (a) the step is a loop-independent runtime invariant and (b) the linear clause has a *uval* modifier. `<pos>` is the position of the step parameter specified in the uniform clause (required by the OpenMP specs). For example, `linear(uval(i):c) uniform(c)` with `c` being the third parameter, gives `Us3`.

**"u"** Uniform parameter specified in the uniform clause. For example, `uniform(c)`.

**"a" <X>** Alignment value (in bytes) specified in the `aligned` clause. For example `aligned(c:a)`.

When targeting Advanced SIMD, if the value `a` is missing, the default alignment value is 16 (128 bits), so that an aligned clause with no alignment is mangled as `a16`.

When targeting SVE, the default value of an `aligned` clause is the alignment of the type pointed to by the corresponding parameter of the scalar signature. For example, `aligned(x)` for `T *x` defaults to the value `_Alignof(typeof(T))`.

## 3.6 Advanced SIMD examples

The following example shows which vector versions are provided when no `simdlen` clause is attached to the `declare simd` directive of a *function declaration*.

```
#pragma omp declare simd
float f(double x);
#pragma omp declare simd
double g(float x);
```

In this case, the vector versions of `f` and `g` operate on vectors consisting of 2 and 4 lanes, both with and without an additional lane masking parameter.

For the example, the available (unmasked) signatures associated to `f` and `g` are:

- `float32x2_t _ZGVnN2v_f(float64x2_t vx);` 2-lane `f`;
- `float64x2_t _ZGVnN2v_g(float32x2_t vx);` 2-lane `g`;
- `float32x4_t _ZGVnN4v_f(float64x4_t vx);` 4-lane `f`;
- `float64x4_t _ZGVnN4v_g(float32x4_t vx);` 4-lane `g`;

It is possible to tune the number of lanes using the `simdlen(N)` clause, where $N = 2^k$ for $k >= 0$. No other values of `simdlen` are allowed.

```
#pragma omp declare simd simdlen(2)
short foo(int64_t x, uint32_t y , int8_t z);
// 2-lane version.
int16x2_t _ZGVnN2vvv_foo(int64x2_t vx, uint32x2_t vy, int8x2_t vz);

#pragma omp declare simd simdlen(4)
short foo(int64_t x, uint32_t y, int8_t z);
// 4-lane version.
int16x4_t _ZGVnN4vvv_foo(int64x4_t vx, uint32x4_t vy, int8x4_t vz);
```

**Note:** Because AArch64 Advanced SIMD uses the first 8 SIMD registers for passing parameters and returning values, it is recommended that the value passed to `simdlen` is such that the signature of the vector function does not use more than 8 input registers, or more than 8 return registers.

## 3.7 SVE Examples

In case of the functions `float f(double)`, `double g(float)` and `short foo(int64_t, int32_t, int8_t)`, the use of `#pragma omp declare simd` will generate the following function signatures:

- `svfloat32_t _ZGVsMxv_f(svfloat64_t, svbool_t)` VLA signature for the vector version of `f`;
- `svfloat64_t _ZGVsMxv_g(svfloat32_t, svbool_t)` VLA signature for the vector version of `g`;
- `svint16_t _ZGVsMxvvv_foo(svint64_t, svint32_t, svint8_t, svbool_t)` VLA signature for the vector version of `foo`.

Note that the svbool_t parameter is described in *SVE masking* (page 16).

Listing 2: Examples with explicit simdlen for SVE.

```
#pragma omp declare simd simdlen(10) notinbranch
#pragma omp declare simd simdlen(16) notinbranch
int32_t foo(int32_t x);

// No 10-lane version generated because ten 4-byte lanes do not
// fit an SVE register.
// SVE 512-bit - widest type is  4 bytes -> 16 lanes
svint32_t _ZGVsM16v_foo(svint32_t vx, svbool_t vmask);

#pragma omp declare simd simdlen(8)
float bar(double x, double y);

// widest type is 8 bytes
// SVE 512-bit -> 8 lanes
svfloat32_t _ZGVsM8vv_bar(svfloat64_t vx, svfloat64_t vy,
                          svbool_t vmask);
```

## 3.8 Linear parameters examples

Input parameters marked with a linear clause need special handling. In particular, the linear clause specifies an implicit vector of values or addresses, depending on the type of the clause.

Table 1: Meaning of linear clause when x is an *integral parameter* or a *pointer*.

| Clause | MAP(x) | Constraints at lane i of the implicit vector |
|---|---|---|
| linear(x:s) | x | x_i = x + i * s |
| linear(val(x):s) | x | x_i = x + i * s |
| linear(uval(x):s) | n/a | n/a |
| linear(ref(x):s) | n/a | n/a |

Table 2: Meaning of linear clause when x is an *integral reference* (C++ and Fortran dummy parameters only).

| Clause | MAP(x) | Constraints at lane *i* of the implicit vector |
|---|---|---|
| linear(x:s) | [&x_0, &x_1, ..., &x_i, ...] | x_i = x + s * i |
| linear(val(x):s) | [&x_0, &x_1, ..., &x_i, ...] | x_i = x + s * i |
| linear(uval(x):s) | x | x_i = x + s * i and &x_i = &x |
| linear(ref(x):s) | x | &x_i = &x + s * i |

Listing 3: C examples for the linear clause. The same rules apply to dummy arguments passed by value in Fortran. Note that the function signatures for the val modifier are the same as when no modifier is present.

```
// Advanced SIMD
#pragma omp declare simd linear(i)
float bar(int32_t i);
// 2-lane version
float32x2_t _ZGVnN2l_bar(int32_t);
// 4-lane version
float32x4_t _ZGVnN4l_bar(int32_t);
```

(continues on next page)

(continued from previous page)

```
#pragma omp declare simd linear(x)
float foo(double *x);
// 2-lane version
float32x2_t _ZGVnN2l8_foo(double *);
// 4-lane version
float32x4_t _ZGVnN4l8_foo(double *);

// SVE
#pragma omp declare simd linear(i)
float bax(int32_t i);
// VLA version
svfloat32_t _ZGVsMxl_bax(int32_t, svbool_t);

#pragma omp declare simd linear(x)
float bax(double *x);
// VLA version with signature
svfloat32_t _ZGVsMxl8_bax(double *, svbool_t);
```

Listing 4: C++ examples for `linear` clause when using reference parameters. The same function signature is generated for dummy arguments passed by reference in Fortran. For simplicity, the masked version for Advanced SIMD is not shown.

```
#pragma omp declare simd linear(ref(x))
int32_t g_ref(int32_t &x); // The vector version holds a pointer to x
// Advanced SIMD - 2-lane version
int32x2_t _ZGVnN2R4_g_ref(int32_t *);
// Advanced SIMD - 4-lane version
int32x4_t _ZGVnN4R4_g_ref(int32_t *);
// SVE - VLA version
svint32_t _ZGVsMxR4_g_ref(int32_t *, svbool_t);

#pragma omp declare simd linear(val(x))
int32_t g_val(int32_t &x); // vector of integral values
// Advanced SIMD - 2-lane version
int32x2_t _ZGVnN2L4_g_val(uint64x2_t vxp);
// Advanced SIMD - 4-lane version
int32x4_t _ZGVnN4L4_g_val(uint64x4_t vxp);
// SVE - VLA version
svint32_t _ZGVsMxL4_g_val(svuint64_t vxp , svbool_t);

#pragma omp declare simd linear(uval(x))
int32_t g_uval(int32_t &x); // scalar, used to produce a vector of integral values from x
// Advanced SIMD - 2-lane version
int32x2_t _ZGVsN2U4_g_uval(int32_t *);
// Advanced SIMD - 4-lane version
int32x4_t _ZGVsN4U4_g_uval(int32_t *);
// SVE - VLA version
svint32_t _ZGVsMxU4_g_uval(int32_t *, svbool_t);
```

CHAPTER

# FOUR

# MASKING

The inbranch and notinbranch clauses define whether or not a vector function should accept a masking parameter.

In all cases, the masking parameter is added to the vector function signature as the last parameter. The following table summarizes the behavior.

Notice that for SVE, masking is present regardless of whether inbranch or notinbranch is used.[6]

Table 1: Masked signature generation for [not]inbranch clause.

| | Advanced SIMD | | SVE | |
|---|---|---|---|---|
| | Masked | Unmasked | Masked | Unmasked |
| #pragma omp declare simd | Yes | Yes | Yes | No |
| #pragma omp declare simd inbranch | Yes | No | Yes | No |
| #pragma omp declare simd notinbranch | No | Yes | Yes | No |

## 4.1 Advanced SIMD masking

For Advanced SIMD, the type of the mask is generated using uint[NDS(f)*8]_t-based vectors.

All bits are set to one for active lanes, and all bits are set to zero for inactive lanes.

**Note:** The narrowest vector input parameter is chosen over the widest one because masking is often intended for lane masking, and not for bit masking of the vector lanes. Using the narrowest vector input parameter also limits the number of parameter registers needed to pass the mask.

**Note:** Because the masking is done using SIMD data registers, to avoid performance degradation it is recommended that the addition of the mask parameter does not overflow the maximum number of 8 vector input registers.

```
#pragma omp declare simd simdlen(2) inbranch
float f(double);
// 2-lane masked version
float32x2_t _ZGVnM2v_f(float64x2_t, uint32x2_t);

#pragma omp declare simd simdlen(2) inbranch
double g(float);
// 2-lane masked version
float64x2_t _ZGVnM2v_g(float32x2_t, uint32x2_t);
```

(continues on next page)

[6] The reason for using predication by default in SVE is to avoid a scalar tail loop when auto-vectorizing loops. The reason for using predication even for notinbranch is to avoid the performance degradation that would occur when porting code that uses functions not guarded by conditional branches that could have been marked as notinbranch.

(continued from previous page)

```
#pragma omp declare simd inbranch
float f(double); // -> float32x2_t(float64x2_t, uint32x2_t)
// 2 and 4-lane masked version
float32x2_t _ZGVnM2v_f(float64x2_t, uint32x2_t);
float32x4_t _ZGVnM4v_f(float64x4_t, uint32x4_t);

#pragma omp declare simd inbranch
double g(float);
// 2 and 4-lane masked version
float64x2_t _ZGVnM2v_g(float32x2_t, uint32x2_t);
float64x4_t _ZGVnM4v_g(float32x4_t, uint32x4_t);

#pragma omp declare simd simdlen(8) inbranch
float f(double);
// 8-lane masked version
float32x8_t _ZGVnM8v_f(float64x8_t, uint32x8_t);

#pragma omp declare simd simdlen(8) inbranch
double g(float);
// 8-lane masked version
float64x8_t _ZGVnM8v_g(float32x8_t, uint32x8_t);
```

**Note:** Using a mask parameter in AArch64 Advanced SIMD is not generally recommended for functions that operate on scalars of different widths, as widening of the input mask for wider types might require using call-preserved temporary registers (V8-V23).

Listing 1: Example of mask parameters for complex values.

```
#pragma omp declare simd inbranch
int32_t foo(_Complex double x);
// Advanced SIMD, 2-lane versions.
// Each logical lane of the mask is a 4 byte sequence,
// either 0x00000000 or 0xffffffff.
int32x2_t _ZGVnM2v_foo(float64x4_t vx, uint32x2_t vmask);

#pragma omp declare simd inbranch
float complex baz(double complex x);
// Double precision complex value -> 16 byte structure
// 2-lane Advanced SIMD.
// The narrowest type is an 8 byte structure, so mask
// is uint64x2_t
float32x4_t _ZGVnM2v_baz(float64x4_t vx, uint64x2_t vmask);

#pragma omp declare simd inbranch
double complex bar(float x, float y);
// Advanced SIMD, 2, and 4-lane.
float64x4_t _ZGVnM2v_bar(float32x2_t vx, float32x2_t vy, uint32x2_t vmask);
float64x8_t _ZGVnM4v_bar(float32x4_t vx, float32x4_t vy, uint32x4_t vmask);
```

## 4.2 SVE masking

For SVE vector functions, whether length-agnostic or length-specific, masked signatures are generated by adding a `svbool_t` mask (or *predicate* in SVE terms) as the last parameter.

```
#pragma omp declare simd
#pragma omp declare simd inbranch
#pragma omp declare simd notinbranch
float f(double);
// SVE - VLA
// Notice that the default behavior is not affected by `inbranch`
// or `notinbranch`.
svfloat32_t _ZGVsMxv_f(svfloat64_t, svbool_t);

#pragma omp declare simd
double g(float);
// SVE - VLA
svfloat64_t _ZGVsMxv_f(svfloat32_t, svbool_t);

#pragma omp declare simd simdlen(4)
float f(double);
// SVE - VLS - > implies a 256-bit implementation
svfloat32_t _ZGVsM4v_f(svfloat64_t, svbool_t);

#pragma omp declare simd simdlen(4)
double g(float);
// SVE - VLS - > implies a 256-bit implementation
svfloat64_t _ZGVsM4v_g(svfloat32_t, svbool_t);
```

### 4.2.1 Generating the predicate value of the mask parameter

The logical lane subdivision of the predicate corresponds to the lane subdivision of the vector data type generated for the widest data type, with one bit in the predicate lane for each byte of the data lane. Active logical lanes of the predicate have the least significant bit set to 1, and the rest set to zero. The bits of the inactive logical lanes of the predicate are set to zero. This method ensures that:

1. The inactive lanes of unpacked vectors do not get treated erroneously as active (see example `foo`).
2. The correct predicate can be generated programmatically from the input predicate for those types of the scalar signature whose layout requires more than 1 bit per active lane.

In the function `foo` of the following example, the widest data type subdivision selects 8-byte wide lanes. Therefore, the active lanes in the predicate will be represented by the 8-bit sequence `00000001`. The original input predicate works for all the types in the signature but not for the `vy` parameter. The callee must generate a new predicate for it, that carries the bit sequence `00010001` for the active lanes, so that the additional bytes of the logical lane associated to the complex type are correctly marked as active.

```
#pragma omp declare simd
double foo(double x, _Complex float y);

// VLA SVE
svfloat64_t _ZGVsMxv_foo(svfloat64_t vx, svfloat32_t vy,
              svbool_t vmask);
// vmask active lane value:  00000001
// vy active lane value:     00010001
```

CHAPTER

FIVE

# ADDITIONAL EXAMPLES

Throughout the following examples, for a given function f, we define *NDS(f)* and *WDS(f)* as the *Narrowest (and respectively, Widest) Size of f* as the size in bytes of the narrowest (and respectively, the widest) among the input parameter types and the return type of the function signature.

The *NDS* and *WDS* values are placed next to the vector signature to explain the choice of the vector length of the function. As a reminder, the former is used to select the vector length when targeting Advanced SIMD vectorization, the latter to select the vector length when targeting VLS SVE functions by using the simdlen clause.

Listing 1: Name mangling example for the SIMD directives with no decorations.

```
#pragma omp declare simd
int32_t foo(int32_t x);

// Advanced SIMD - NDS(foo) = 4 -> 2 and 4 lanes
int32x2_t _ZGVnN2v_foo(int32x2_t vx);
int32x2_t _ZGVnM2v_foo(int32x2_t vx, uint32x2_t vmask);
int32x4_t _ZGVnN4v_foo(int32x4_t vx);
int32x4_t _ZGVnM4v_foo(int32x4_t vx, uint32x4_t vmask);

// VLA SVE
svint32_t _ZGVsMxv_foo(svint32_t vx, svbool_t vmask);
```

Listing 2: Example mangling for a function with uniform and linear clause, with val modifier. The inbranch clause generates only the masked version for Advanced SIMD.

```
#pragma omp declare simd inbranch uniform(x) linear(val(i):4)
int32_t foo(int32_t *x, int32_t i);

// Advanced SIMD - NDS(foo) = 4 -> 2 and 4 lanes
int32x2_t _ZGVnM2ul4_foo(int32_t *x, int32_t i, uint32x2_t vmask);
int32x4_t _ZGVnM4ul4_foo(int32_t *x, int32_t i, uint32x4_t vmask);

// VLA SVE
svint32_t _ZGVsMxul4_foo(int32_t *x, int32_t i, svbool_t vmask);
```

Listing 3: Example of function name mangling when a runtime linear step is specified in the linear clause.

```
#pragma omp declare simd inbranch uniform(x,c) linear(i:c)
int32_t foo(int32_t *x, int32_t i, uint8_t c);

// Advanced SIMD - NDS(foo) = 1 -> 8 and 16 lanes
int32x4x2_t _ZGVnM8uls3u_foo(int32_t *x, int32_t i, uint8_t c, uint32x8_t vmask);
int32x4x4_t _ZGVnM16uls3u_foo(int32_t *x, int32_t i, uint8_t c, uint32x16_t vmask);
```

(continues on next page)

(continued from previous page)

```
// VLA SVE
svint32_t _ZGVsMxuls3u_foo(int32_t *x, int32_t i, uint8_t c, svbool_t vmask);
```

Listing 4: Example of vector function name generation from a fixed length simd declaration.

```
#pragma omp declare simd simdlen(4)
int32_t foo(int32_t x, float y);

// Advanced SIMD - NDS(foo) = 4 -> 4 lanes
int32x4_t _ZGVnN4vv_foo(int32x4_t vx, float32x4_t vy);
int32x4_t _ZGVnM4vv_foo(int32x4_t vx, float32x4_t vy, uint32x4_t vmask);

// SVE 128-bit - WDS(foo) = 4 -> 4 lanes
svint32_t _ZGVsM4vv_foo(svint32_t vx, svfloat32_t vy, svbool_t vmask);
```

Listing 5: Example with output size bigger than input size.

```
#pragma omp declare simd
double foo(float x)

// Advanced SIMD -  NDS(foo) = 4 -> 2 and 4 lanes
float64x2_t _ZGVnN2v_foo(float32x2_t vx);
float64x2_t _ZGVnM2v_foo(float32x2_t vx, uint32x2_t vmask);
float64x4_t _ZGVnN4v_foo(float32x4_t vx);
float64x4_t _ZGVnM4v_foo(float32x4_t vx, uint32x4_t vmask);

// VLA SVE - input in unpacked
svfloat64_t _ZGVsMxv_foo(svfloat32_t vx, svbool_t vmask);
```

Listing 6: Example with explicit alignment.

```
#pragma omp declare simd linear(x) aligned(x:16) simdlen(4)
int32_t foo(int32_t *x, float y);

// Advanced SIMD - NDS(foo) = 4 -> 4 lanes
int32x4_t _ZGVnN4la16v_foo(int32_t *x, float32x4_t vy);
int32x4_t _ZGVnM4la16v_foo(int32_t *x, float32x4_t vy, uint32x4_t vmask);

// SVE 128-bit - WDS(foo) = 4 -> 4 lanes
svint32_t _ZGVsM4la16v_foo(int32_t *x, svfloat32_t vy, svbool_t vmask);
```

The following example shows how to handle types that do not map directly to integers, floating point types or complex types. In this specific case, the rules give the following:

1. `MTV(P) = true` by rule 3 of 3.1.1.
2. `PBV(P) = false` by rule 3 pf 3.1.2.
3. Because `MTV(P)` is `true`, rule 2 of 3.1.3 applies.
4. Because `PBV(P)` is `false` and `MTV(P)` is `true`, rule 3 of 3.2.1 applies and therefore `LS(P)` is `sizeof(uintptr_t)`.

Listing 7: Example with generic types. In this case, the rules lead to mapping each concurrent object to pointers.

```
struct S { uint8_t R,G,B; };

#pragma omp declare simd notinbranch
```

(continues on next page)

(continued from previous page)

```
S DoRGB(S x);

// Advanced SIMD - NDS(DoRGB) = 8 (LP64 data model)
uint64x2_t  _ZGVnN2v_DoRGB( uint64x2_t vx); // 2-lane
// Advanced SIMD - NDS(DoRGB) = 4 (ILP32 data model)
uint32x2_t  _ZGVnN2v_DoRGB( uint32x2_t vx); // 2-lane
uint32x4_t  _ZGVnN4v_DoRGB( uint32x4_t vx); // 4-lane

// VLA SVE - WDS(DoRGB) = 8 (LP64 data model)
svint64_t _ZGVsMxv_DoRGB(svint64_t vx, svbool_t vmask);
// VLA SVE - WDS(DoRGB) = 4 (ILP32 data model)
svint32_t _ZGVsMxv_DoRGB(svint32_t vx, svbool_t vmask);
```

Listing 8: Example mangling for a function with `uniform` and `linear` clause, for corner case values.

```
#pragma omp declare simd linear(x:y) uniform(y) linear(z) linear(ref(k):-1) notinbranch
uint32_t foo(int32_t x, int32_t y, int32_t z, int32_t &k) {

// Advanced SIMD - NDS(foo) = 4 -> 2 and 4 lanes
uint32x2_t _ZGVnN2ls2ulRn4_foo(int32_t x, int32_t y, int32_t z, int32_t *k)
uint32x4_t _ZGVnN4ls2ulRn4_foo(int32_t x, int32_t y, int32_t z, int32_t *k)

// VLA SVE
svuint32_t _ZGVsMxls2ulRn4_foo(int32_t x, int32_t y, int32_t z, int32_t *k, svbool_t vmask);
```

Listing 9: Example mangling for default alignment values (assuming LP64).

```
typedef struct D { double a[2];} D_ty;

#pragma omp declare simd \
aligned(x) aligned(y) aligned(z) aligned(S) \
linear(x) linear(y) linear(z) linear(S) notinbranch
int32_t foo(int32_t *x, double *y, uint8_t *z, D_ty * S);

// Advanced SIMD - NDS(foo) = 4 -> 2 and 4 lanes (showing only the 2 lanes one)
int32x2_t _ZGVnN2l4a16l8a16la16l16a16_foo(int32_t *x, double *y, uint8_t *z, D_ty * S)

// VLA SVE (VLS would have the same aligment tokens)
svint32_t _ZGVsMxl4a4l8a8la1l16a16_foo(int32_t *x, double *y, uint8_t *z, D_ty * S, svbool_t)
```