

VS IPL 1.0 API Requirements Document

David A. Schwartz
HRL Laboratories, LLC

Randall R. Judd
Space and Naval Warfare Systems Center, San Diego

James M. Lebak
MIT Lincoln Laboratory

Randall S. Janka
Georgia Tech Research Institute



<http://www.vsipl.org>

March 14, 2000

©1999-2000 Georgia Tech Research Corporation, all rights reserved.

A non-exclusive, non-royalty bearing license is hereby granted to all persons to copy, modify, distribute and produce derivative works for any purpose, provided that this copyright notice and following disclaimer appear on all copies: THIS LICENSE INCLUDES NO WARRANTIES, EXPRESSED OR IMPLIED, WHETHER ORAL OR WRITTEN, WITH RESPECT TO THE SOFTWARE OR OTHER MATERIAL INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE, OR ARISING FROM A COURSE OF PERFORMANCE OR DEALING, OR FROM USAGE OR TRADE, OR OF NON-INFRINGEMENT OF ANY PATENTS OF THIRD PARTIES. THE INFORMATION IN THIS DOCUMENT SHOULD NOT BE CONSTRUED AS A COMMITMENT OF DEVELOPMENT BY ANY OF THE ABOVE PARTIES.

This material is based in part upon work supported by Ft. Huachuca/DARPA under contract No. DABT63-96-C-0060. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of Ft. Huachuca/DARPA.

The US Government has a license under these copyrights, and this material may be reproduced by or for the US Government.

Basic VSIPL Requirements.....	1
Introduction.....	1
Blocks and Views	1
Disclaimer	1
Language Binding Requirements	1
VSIPL Naming Convention and Functionality Requirements	2
Summary of VSIPL Types	2
Basic Data Types.....	6
Scalar Data Types	6
Block Data Types.....	7
View Data Types.....	8
Block Requirements.....	9
View Requirements.....	10
Additional Requirements for Complex Blocks and Views.....	11
User Data	12
Development Mode Requirements	12
Library Initialization and Finalization	13

Basic VSIPL Requirements

Introduction

This document contains basic requirements that must be met by a VSIPL compliant library. The basic requirements do not include the functionality, which is described in other documents. The basic requirements include the basic type definitions, *block* requirements, *view* requirements, and other basic VSIPL requirements. Other requirements in the functionality section pertain to a small subset of functions and are not covered here. This document covers the more general requirements that must be met by all VSIPL implementations.

It is important to note that this document is a summary of the basic requirements of the VSIPL API. If and when this requirements summary document disagrees with the API specification, the API will supercede this requirements document.

Blocks and Views

It is difficult to write about VSIPL without using the term *block* or *view*. Blocks and views are covered in detail later in the document. As a starting point this section is an introduction to the terms.

In VSIPL a *block* can be thought of as a contiguous area of memory for storage of data.¹ A *block object* associated with the block contains the information necessary for the VSIPL implementation to access this memory.

VSIPL allows the user to construct a *view* of the data in a block as a vector, matrix, or higher dimensional object. The information necessary to access the required information in the block in a manner consistent with the *view* is stored in the *view object*.

Disclaimer

The VSIPL library is *object-based*, **not** *object-oriented*. The reader should be careful not to bring to this document any object-oriented terms that may have specific meaning to them from another context, but which are used within VSIPL to mean something else.

Language Binding Requirements

The standard only defines an ANSI C application program interface.

¹ The memory area might not actually be contiguous, but must appear contiguous to the user.

VSIPL Naming Convention and Functionality Requirements

While there is nothing to prevent an implementor from writing VSIPL-compatible functions, only those functions that are approved and included in formal VSIPL documentation are a part of VSIPL. Functions outside the standard should not use the VSIPL naming conventions in order to avoid confusion and application porting problems. In particular, names outside of VSIPL should not start with “vsip” or “vsipl”, either in caps or lower case.

The exact names of VSIPL functions depend on the precision of the operation. The base names of VSIPL functions are explicitly specified in the functionality section. The VSIPL function name consists of the base name plus a precision affix which specifies the data precision. This is done to allow wide variation in precision to support diverse hardware. The allowed affixes are covered in the summary of VSIPL types below. Except for copy functions, the precision affix is usually a suffix. The precision affix for a copy function consists of two precision affixes corresponding to the data types of the source and destination data arrays.

Summary of VSIPL Types

All VSIPL type declarations and function names have the data type encoded into the name. Throughout the VSIPL documentation, a *generalized affix* of *_p* is used to denote a general precision of any type, and is a method to name functions or data types without spelling out every single prefix which might be needed for that function or data type. Other generalized affixes used are *_i* to denote any integer, or an *_f* to denote any float. The generalized affix is in an italic font style. To produce a valid VSIPL name use a specified name from the functionality section, and replace the generalized data type affix with the selected affix from the table below.

For example, the function `vsip_mag_p` listed in the functionality section takes the magnitude (absolute value) of its argument. A specific instance of this function which operates on single-precision floating-point data would be called `vsip_mag_f`, and a version that operates on integer data would be called `vsip_mag_i`. As a further example, consider the complex FFT function `vsip_ccfftop_f`. The “*_f*” suffix indicates that instances of the function that operate on single-precision, double-precision, and extra-precision floating-point data are part of the specification, while an instance that operates on integer data is not currently included.

The following table contains VSIPL affix notations for use in encoding type data in the names and type declarations, and a description of the data types supported in VSIPL. It is not expected that any implementation will support all possible VSIPL data types. The data types supported will depend in part on the hardware for which the library was developed, and the expected use of the hardware.

Standard Floating Point Data Types

Affix	Definition
<code>_f</code>	ANSI C single precision floating point
<code>_d</code>	ANSI C double precision floating point
<code>_l</code>	ANSI C extra precision floating point

Standard Integer Data Types	
Affix	Definition
<code>_c</code>	ANSI C char
<code>_uc</code>	ANSI C unsigned char
<code>_si</code>	ANSI C short integer
<code>_us</code>	ANSI C unsigned short integer
<code>_i</code>	ANSI C integer
<code>_u</code>	ANSI C unsigned integer
<code>_li</code>	ANSI C long integer
<code>_ul</code>	ANSI C unsigned long integer
<code>_ll</code>	Long, long integer, implementation dependent
<code>_ull</code>	Unsigned long, long integer, implementation dependent
Portable Precision Floating Point Data Types	
Affix	Definition
<code>_f6</code>	Floating point type with at least 6 decimal digits of accuracy. IEEE 754 single precision (32 bit) has 6 decimal digits of accuracy.
<code>_f15</code>	Floating point types with at least 15 decimal digits of accuracy. IEEE 754 double precision (64 bit) has 15 decimal digits of accuracy.
<code>_fn</code>	Floating point type with at least n decimal digits of accuracy. If the system supports such a precision, it resolves to the smallest C type based on the values of <code>FLT_MANT_DIG</code> , <code>DBL_MANT_DIG</code> , or <code>LDBL_MANT_DIG</code> (which are defined in <code>float.h</code>).

Portable Precision Integer Data Types	
Affix	Definition
<code>_i18</code>	<i>int</i> of at least 8 bits
<code>_i16</code>	<i>int</i> of at least 16 bits
<code>_i32</code>	<i>int</i> of at least 32 bits
<code>_i64</code>	<i>int</i> of at least 64 bits
<code>in</code>	<i>int</i> of at least n bits
<code>_u18</code>	unsigned <i>int</i> of at least 8 bits
<code>_u16</code>	unsigned <i>int</i> of at least 16 bits
<code>_u32</code>	unsigned <i>int</i> of at least 32 bits
<code>_u64</code>	unsigned <i>int</i> of at least 64 bits
<code>un</code>	unsigned <i>int</i> of at least n bits
<code>_ie8</code>	<i>int</i> of exactly 8 bits
<code>_ie16</code>	<i>int</i> of exactly 16 bits
<code>_ie32</code>	<i>int</i> of exactly 32 bits
<code>_ie64</code>	<i>int</i> of exactly 64 bits
<code>ien</code>	<i>int</i> of exactly n bits
<code>_ue8</code>	unsigned <i>int</i> of exactly 8 bits
<code>_ue16</code>	unsigned <i>int</i> of exactly 16 bits
<code>_ue32</code>	unsigned <i>int</i> of exactly 32 bits
<code>_ue64</code>	unsigned <i>int</i> of exactly 64 bits
<code>uen</code>	unsigned <i>int</i> of exactly n bits
<code>_if8</code>	fastest <i>int</i> of at least 8 bits
<code>_if16</code>	fastest <i>int</i> of at least 16 bits
<code>_if32</code>	fastest <i>int</i> of at least 32 bits
<code>-if64</code>	fastest <i>int</i> of at least 64 bits
<code>ifn</code>	fastest <i>int</i> of at least n bits
<code>_uf8</code>	unsigned fastest <i>int</i> of at least 8 bits
<code>_uf16</code>	unsigned fastest <i>int</i> of at least 16 bits
<code>_uf32</code>	unsigned fastest <i>int</i> of at least 32 bits
<code>_uf64</code>	unsigned fastest <i>int</i> of at least 64 bits
<code>ufn</code>	unsigned fastest <i>int</i> of at least n bits

Other Data Types	
Affix	Definition
_bl	Boolean Data Type. Logical <i>false</i> for 0, and Logical <i>true</i> for non-zero.
_vi	Vector Index. This is an unsigned integer of sufficient precision to index any VSIPL vector.
_mi	Matrix Index. This is a data type used for accessing matrix elements. The precision of the type is the same as _vi. The <i>matrix index</i> of the element $x_{i,j}$ is the 2-tuple $\{i, j\}$.
_ti	Tensor Index. This is a data type used for accessing tensor elements. The precision of the type is the same as _vi. The <i>tensor index</i> of the element $x_{i,j,k}$ is the 3-tuple $\{i, j, k\}$.

Basic Data Types

VSIPL has three basic data types, *scalars*, *blocks*, and *views*. This section discusses the requirements for these basic data types. VSIPL also has other special data types and structures, used for defining special objects in a single function or a small subset of functions. A summary of all VSIPL types, including these special structures and data types, is given in the functionality section and will not be repeated here.

Scalar Data Types

All supported VSIPL scalars have a type definition of `vsip_scalar_p` for real scalars, and a type definition of `vsip_cscalar_p` for complex scalars. Complex scalars are only supported for float and integer data types.

The following is an example of a VSIPL header definition for a scalar float and a scalar unsigned integer.

```
typedef float vsip_scalar_f;
typedef unsigned int vsip_scalar_u;
```

The following table gives the VSIPL header definitions for various types (assuming they are supported by an implementation). Some of the information is implementation dependent, and is indicated with a bracket (<?...?>) around the dependent section.

Complex	<code>typedef struct {vsip_scalar_p r, i;} vsip_cscalar_p;</code>
Boolean	<code>typedef <?char?> vsip_scalar_bl; typedef vsip_scalar_bl vsip_bool; #define VSIP_FALSE 0 #define VSIP_TRUE 1</code>
Vector index	<code>typedef unsigned <?long int?> vsip_scalar_vi typedef vsip_scalar_vi vsip_index;</code>
Matrix index	<code>typedef struct {vsip_scalar_vi r,c;} vsip_scalar_mi;</code>
Tensor index	<code>typedef struct {vsip_scalar_vi z,y,x;} vsip_scalar_ti;</code>
Offset	<code>typedef vsip_scalar_vi vsip_offset;</code>
Stride	<code>typedef signed <?long int?> vsip_stride;</code>
length	<code>typedef vsip_scalar_vi vsip_length;</code>

Note: The data type for the vector index (`vsip_scalar_vi`) is implementation dependent. It must be an *unsigned* integer of sufficient size to allow indexing any possible view element of the implementation.

Note: The stride data type must be a *signed* integer of the same number of bits

precision as the vector index.

Block Data Types

All supported VSIPL blocks have a type definition as described in the following table.

Type	VSIPL blocks
<code>vsip_block_p</code>	For real blocks, index blocks, and boolean blocks.
<code>vsip_cblock_p</code>	For complex blocks. Complex blocks are only supported for float and integer data.

The implementation must provide type definitions for these VSIPL block objects, so that they can be declared in the user’s application. The names of the type definitions are specified by VSIPL; however, the underlying structures are implementation dependent. The following examples show how the structures may be hidden using incomplete type definitions. The examples name the structures `vsip_blockobject_bl`, `vsip_blockobject_vi`, `vsip_blockobject_d`, and `vsip_cblockobject_d`. These structures and their names are all implementation dependent. The only required names are those for the type definitions of the block objects, shown in bold in each example.

Examples of incomplete type definitions for blocks
<code>struct vsip_blockobject_bl; /* boolean block structure */ typedef struct vsip_blockobject_bl vsip_block_bl;</code>
<code>struct vsip_blockobject_mi; /* matrix index block structure */ typedef struct vsip_blockobject_mi vsip_block_mi;</code>
<code>struct vsip_blockobject_d; /* double block structure */ typedef struct vsip_blockobject_d vsip_block_d;</code>
<code>struct vsip_cblockobject_d; /* complex double block structure */ typedef struct vsip_cblockobject_d vsip_cblock_d;</code>

VSIPL provides hints that specify how the memory referenced by a particular block will be used. These hints may provide the implementation with additional optimization opportunities. It is not required that any optimizations be performed when a hint is specified, but the structure must be available for portability.

```
typedef enum {
    VSIP_MEM_NONE = 0,
    VSIP_MEM_RDONLY = 1,
    VSIP_MEM_CONST = 2,
    VSIP_MEM_SHARED = 3,
    VSIP_MEM_SHARED_RDONLY = 4,
    VSIP_MEM_SHARED_CONST = 5
} vsip_memory_hint;
```

View Data Types

All supported VSIPL views have a type definition as follows:

<code>vsip_vview_p</code>	For real vector views, boolean vector views and index vector views.
<code>vsip_cvview_p</code>	For complex vector views.
<code>vsip_mview_p</code>	For real matrix views and boolean matrix views.
<code>vsip_cmview_p</code>	For complex matrix views.
<code>vsip_tview_p</code>	For real tensor views.
<code>vsip_ctview_p</code>	For complex tensor views.

Note that all index views are vectors. There are only types `vsip_vview_vi`, `vsip_vview_mi`, and types `vsip_vview_ti`.

The implementation must provide type definitions for these VSIPL view objects, so that they can be declared in the user's application. The names of the type definitions are specified by VSIPL; however, the underlying structures are implementation dependent. The following examples show how the structures may be hidden using incomplete type definitions. The examples show the structures `vsip_vviewobject_bl`, `vsip_vviewobject_vi`, `vsip_vviewobject_d`, and `vsip_cvviewobject_d`. These structures and their names are all implementation dependent. The only required names are those for the type definitions of the view objects, shown in bold in each example.

Examples of incomplete type definitions for views
<pre>struct vsip_vviewobject_bl; /* boolean vector view struct */ typedef struct vsip_vviewobject_bl vsip_vview_bl;</pre>
<pre>struct vsip_vviewobject_vi; /* vector index view struct */ typedef struct vsip_vviewobject_vi vsip_block_vi;</pre>
<pre>struct vsip_vviewobject_d; /* double vector view struct */ typedef struct vsip_blockobject_d vsip_block_d;</pre>
<pre>struct vsip_cvviewobject_d; /* complex double vector view struct */ typedef struct vsip_cvviewobject_d vsip_cblock_d;</pre>

Block Requirements

A *block* is a VSIPL type representing an object where data is stored. The *block* is conceptually a one-dimensional data array of elements of a single data type. The user supplies the size of the block on its creation. The block offset of the first element is at zero, and the block offset of the final element is at N-1, where N is the total length of the block.

Blocks are of two kinds, user blocks and VSIPL blocks. A *user block* is one which is created using a data array allocated directly by the application so that the application has a pointer to the data array. A *VSIPL block* is one which is created entirely by VSIPL functions, and the user has no proper method to retrieve a pointer to the data array.

A VSIPL block is created with a VSIPL creation function. When a VSIPL block is created, the data array bound to it is created at the same time. The details of the physical storage of the data array are implementation-dependent; however the data array appears as contiguous data elements for the purpose of assigning strides and offsets in views of the block.

A block may be in one of two states, *admitted* or *released*. The data array associated with an *admitted* block may not be manipulated directly by the user. VSIPL functions must be used to access the data array and perform operations. The data array associated with a block in the *released* state may be accessed directly by the user, but VSIPL implementations should not perform computation on it. A *VSIPL* block is created in the *admitted* state and can not be *released*. A *user* block is created in the *released* state, and can be *admitted* or *released* as required by the application.

Access to a *released user* block's data must be through direct manipulation of the data array. It is an error for a *released user* block's data array to be accessed by any VSIPL function which will read or write elements in the data array. Access to an *admitted user* block must be through VSIPL functions. It is an error to directly manipulate or read a *user* block's data array while it is *admitted* to VSIPL; the behavior in such a

case is undefined.

It is possible to create a *user block* using a `NULL` data pointer. A *user block* bound to a `NULL` pointer can not be *admitted* until the *block* is rebound to a data pointer which is not `NULL`. A *user block* in the *released* state can be rebound to any valid pointer of the proper data type for the *block*. If, and only if, a block is in the released state can it be rebound to a different data array.

When a *released* block is admitted, the implementation is free to do whatever is necessary without concern for the *released* data layout. The *data* must be restored to the same location and layout in the *user data array* when the block associated with the user data array is *released*.

A *user block*, *admitted* or *released*, and a *VSIPL block*, which contain data of the same type, have a single block data type. Any information needed by the implementation developer regarding the state of the block (*admitted*, *released*, *user*, *VSIPL*, etc) is hidden from the application using some implementation dependent method.

View Requirements

A *view* is a *VSIPL type* representing some portion of the data in a block. A block can have many *views* bound to it; however, a *view* can only be bound to a single block.

When a *view* is created it is bound to a block. A *view* has an *attribute* which defines the *block* the *view* is bound to. The *block attribute* **may not** be modified by the user.

A *view* has an *offset attribute* which indicates the number of *elements* from the beginning of the *block* where the first *element* of the *view* is located within the block. The *offset attribute* is indexed starting at zero so that an offset of zero implies the first *element* of the *view* is the first *element* of the block. The offset attribute **may** be modified by the user.

A *view* has one or more stride attributes. The magnitude of the stride attribute defines the distance between two consecutive elements in some view axis direction. For example, the row stride indicates how many elements (through the block) from one element in the row to the next element in the row. The distance defined is through the block. The sign of the stride attribute defines what direction the view description moves through the block as the index value of the view description increases. A stride of zero must be supported. The stride attribute **may** be modified by the user.

A *view* has a length attribute for each stride attribute. A length attribute is a positive integer describing the number of elements in the view axis direction, such as the number of elements in the row of a matrix. The length attribute **may** be modified by the user.

For vectors there is only one axis direction so there is only one stride and one length. For matrices there are two view axis directions so there are two strides and two lengths. For tensors there are three view axis directions so there are three strides and three lengths. Additional information on offset, stride, and length attributes are available in the functionality section.

Additional Requirements for Complex Blocks and Views

It is possible to create a *derived view* of the real or imaginary portion of a complex *view*. Note that this is **not** a copy. Replacing an element in the *real* or *imaginary view* derived from the *complex view* replaces the corresponding element in the *complex view*. Similarly, replacing a complex element in the *complex view* replaces the corresponding elements in the *real view* and *imaginary view*.

The *real view* and *imaginary view* of the *complex view* are real and are not complex. They must be bound to a *block* of type `vsip_block_p`. The real block bound to the *real* or *imaginary view* of a *complex view* is a *derived block*. The method of instantiating a *derived block* is implementation-dependent. As a result of the implementation-dependent nature of *derived blocks*, the *stride* and *offset* of *derived views* are **not** determined until after the *view* is created.

The *required derived block* data space encompasses the entire real portion of the complex block if a real view is derived, or the entire imaginary portion of the complex block if an imaginary view is derived. The *derived block* can encompass other portions of the complex block outside the range of the required real or imaginary data space; however, the implementation is only required to maintain views bound to the *required derived block* data space. It is an error to bind new views to a *derived block* which will encompass both real and imaginary portions of the original complex block. The result is implementation dependent.

The *derived block* is destroyed when the complex block is destroyed. It is an error to attempt to destroy a derived block directly. The implementation must maintain sufficient state information in the *complex* block and the *derived* real block to support the proper behavior of the *derived* block.

The internal format of any *admitted* block is implementation dependent, so the underlying memory layout of the complex block is unknown. The following conditions must be met by an implementation for derived blocks.

1. A *derived block* bound to a *user complex block* can not be directly *released*. The *derived block* is *released* when the *complex block* it is bound to is released.
2. A *derived block* bound to a *user complex block* can not be directly *admitted*. The *derived block* is *admitted* when the *complex block* it is bound to is *admitted*.

User Data

This section covers the required data layout of *user data arrays*. The implementation developer must support, and the application developer must use, the required data array formats for *user data*. These formats allow for portable input of user data into VSIPL, and portable output of VSIPL results to the application.

For **float** the user data array is contiguous memory of type `vsip_scalar_f`.

For **integer** the user data array is contiguous memory of type `vsip_scalar_i`.

For **boolean** the user data array is contiguous memory of type `vsip_scalar_bl`.

For **vector index** the user data array is contiguous memory of type `vsip_scalar_vi`.

For **matrix index** the user data array is contiguous memory of type `vsip_scalar_vi`. The matrix index element in a user data array is two consecutive elements of type `vsip_scalar_vi`. The first element is the row, the second is the column. Note that the matrix index element in a user data array is not the same as `vsip_scalar_mi`. (This storage method corresponds to the *interleaved* storage method described below for complex data.)

For **tensor index** the user data array is contiguous memory of type `vsip_scalar_vi`. The tensor index element in a user data array is three consecutive elements of type `vsip_scalar_vi`. The first value in the element is the z index, the second is the y index, and the third is the x index. Note that the tensor index element in a user data array is not the same as `vsip_scalar_ti`.

For **complex float** or **complex integer** the user data array is either *interleaved* or *split* as described below. Both the interleaved and split formats must be supported for user data. Note that the data format for complex *user data* arrays is not of type `vsip_cscalar_p`.

Interleaved: The user data array is contiguous memory of type `vsip_scalar_p`. The complex element is two consecutive elements of type `vsip_scalar_p`. The first element is the real component and the second is the imaginary component.

Split: The user data array consists of two contiguous memory regions of equal length, each of type `vsip_scalar_p`. The real and the imaginary region are determined when the memory is bound to the block. A complex element consists of corresponding elements from the real and imaginary regions.

Development Mode Requirements

The functionality section has required error checks for *development* mode. The basic requirement is that the implementation developer of a library supporting development

mode maintains sufficient information within the implementation to support the required error checks for every function supported by the implementation.

Library Initialization and Finalization

Two functions, `vsip_init` and `vsip_finalize`, control the initialization and finalization of the VSIPL library. The use of initialization and finalization is required for all VSIPL programs; hence, a library must include these functions in any VSIPL library regardless of the profile. It must be possible to call the initialize and finalize functions an arbitrary number of times during the lifetime of a program that uses VSIPL. To support third party libraries that use VSIPL without the knowledge of the application programmer, calls to `vsip_init` and `vsip_finalize` functions may be nested. In addition, sequences of `vsip_init` and `vsip_finalize` pairs may occur in a given program. The following program is required to be executable:

```
/* example of nesting and sequences of init/finalize */
#include "vsip.h"

int main()
{
    /* Nested vsip_init and vsip_finalize */
    vsip_init((void *)0);
    vsip_init((void *)0);
    vsip_finalize((void *)0);
    vsip_finalize((void *)0);

    /* No VSIPL calls permitted here */

    vsip_init((void *)0);
    /* A second sequence of VSIPL calls */
    vsip_finalize((void *)0);

    return 0;
}
```

In each implementation, `vsip_init` and `vsip_finalize` are allowed to do as much or as little internally as is needed in order to support VSIPL services. Some implementations may do little or nothing inside these functions, while others may do quite a bit of resource management, allocating resources when `vsip_init` is called and returning them after the last call to `vsip_finalize`. The user must explicitly destroy all VSIPL objects before calling this function if this is an “outermost” `vsip_finalize`. When nesting initializations, there is no need to destroy all objects prior to calling this function, but the user is obliged to keep track of the nesting depth if programs are written in such a manner. When the “outermost” `vsip_finalize` is called, the implementation is required to return any resources that were allocated previously, and should strive to restore the state of special system resources (e.g., floating point state).