TASP VSIPL Core Lite

Randall Judd

This document is a description of, and a manual for, an implementation of the VSIPL Core Lite Profile by the Tactical Advanced Signal Processing Common Operating Environmnet working group. This work is supported by PEO (USW) PMS411, Jim Broughton.

Space and Naval Warfare Systems Center
San Diego
D881

TABLE OF CONTENTS

CHAPTER 1 Introduction To TASP VSIPL and the Core Lite Profile

Introduction

This book describes the functionality of the Core Lite profile defined for the VSIP library. In particular this book describes an implementation of the Core Lite profile developed by the TASP (Tactical Advanced Signal Processing) COE (Common Operating Environment) effort..

This book is not a copy of, nor a replacement for, the VSIPL specification.

Code History

The original code basis for the library was a pre-alpha (incomplete) version of the VSIPL Reference library produced by Hughes Research Laboratory of Malibu, California in December of 1997. This library has been greatly reorganized and modified to fit a format more suitable to the author's vision of a VSIP library, and also the author's programing ability. The original December release was template based using m4 as a code generator. The author's method was to copy the generated C files and header files and modify them directly, instead of trying to maintain a template method he did not understand. In addition many changes have been made to the library to add performance, and to keep up with the changing VSIPL specification.

TASP and the TASP COE

The TASP group started out as an effort by NAVSEA PMS 428 (Now PMS 411) to do a procurement of COTS signal processing hardware for DOD use similar to the TAC program. One of the goals of the TASP group was to foster a Common Operating Environment (COE) for signal processing. Without a COE for signal processing the software upkeep cost of COTS signal processing hardware will be prohibitive.

In the last few years the methods used by DOD to procure hardware have changed, and eventually the TASP effort for hardware procurement was abandoned. However the COE effort is still important and has survived. One of the elements of a COE for Signal Processing is a common signal processing library supported by multiple vendors. The TASP group has decided to support the VSIP Library Forum effort to produce a de facto signal processing standard, and eventually an actual standard, for a signal processing library. If successful VSIPL will be used for the TASP COE signal processing library.

The VSIP Library Effort and the VSIPL Forum

The VSIP library (VSIPL) effort was initially funded by DARPA and headed by Hughes Research Lab (HRL) (David Schwartz). HRL has changed their name and are now called HRL Laboratories, LLC. HRL no longer stands for Hughes Research Lab.

The main goal of the VSIPL Forum is to produce a signal processing library specification suitable for a wide variety of embedded hardware. The specification will allow vendors to write an efficient and fast library implementation for their product, and at the same time will allow VSIPL application programers to write portable code which will run on a variety of VSIPL compliant hardware without major porting efforts.

Many groups have participated in the VSIPL Forum. For a more complete list of participants one should refer to the VSIPL specification Acknowledgment section.

The primary external funding for the forum was from DARPA, and from TASP; however most companies participated with no external funding.

The TASP VSIPL Demonstration Library

It should be made clear that this document is a product of the TASP effort, and the associated library is also a TASP effort. These products are considered to be separate from the VSIPL effort. We are not trying to do a separate specification, of course, but just as vendors will write their own VSIPL compliant library for their product as independent agents, without the desire for, or need of, the VSIPL Forum telling them how to go about their business, so too this effort is independent of the VSIPL forum.

Since the author is an active participant of the VSIPL Forum people may get the idea that this document, or the associated library effort have somehow been blessed as part of the specification. This is not the case. The author felt the need for a certain amount of independence in developing this library in order to produce a product for demonstration purposes at the earliest time. As such he felt it was necessary to make design decisions independent of any other agent.

For this reason people should be cautious and view this library as what it is. The advantage of being independent is the ability to get a lot of work done. The disadvantage is you may screw up the implementation, and depart from the specification. This implementation is not well tested. It attempts to be VSIP compliant, but it may have some problems. Many functions were completed with an eye toward getting something on the road versus writing a really good function. Participants who find departures from the VSIPL specification within this document, or within the library, or who find software errors, are encouraged to contact the author via email (judd@spawar.navy.mil).

As time goes by, if VSIPL is successful, the TASP VSIPL library may eventually become very good; or if another better public domain VSIPL library becomes available, this library may become unused. In any case view the results of every function with a certain amount of caution. No claims are made that the associated library, or this document, are good for any purpose.

The Core Lite Profile

The entire function list defined by the VSIPL specification is very large, and it would be prohibitively expensive to produce optimized code on embedded hardware for the entire library. Several of the signal processing hardware and software vendors proposed a profile of the library that was very small and was, they felt, usable and relatively inexpensive to produce. This profile has only 126 required functions and has been called Core Lite. There is another larger profile, which we won't discuss here, termed Core. Core is a subset of VSIPL, and Core Lite is a subset of the Core profile.

VSIPL Fundamentals

A little background into the VSIP methods used for defining data types and functions is needed before the function list and VSIPL programing are introduced. Some of this information is specific to the method used within the TASP VSIPL implementation, and it is not necessarily true that other implementation would use the same method. The VSIPL specification tries to abstract the method of achieving the end away from the end itself. As long as a vendor achieves the proper API and meets all the rules, the exact method for achieving the correct result is not of concern to VSIPL. However talking in abstract terms sometimes leads to confusion, so the author will be a little more direct in talking about how the TASP VSIPL implementation achieved the desired result. This should make it easier for the user to understand the implementation and how to use VSIPL. After one library is learned, any other compliant libraries used will be the same, no matter what internal (private) methods were used to define the VSIPL objects.

One should not bring excess programing definitions to this document. VSIPL is an object based method, but it is not object oriented in the strict sense. The author is not knowledgeable of object oriented terminology or programing. The author knows how the forum uses certain terms, and that there are regular discussions (or arguments) about using some terms improperly. For the purpose of this document the author will attempt to explain how he is using the term, and hopefully the reader will not be to critical.

VSIPL Objects and Data Types

Roughly speaking VSIPL has three basic types.

The first base type is a VSIPL *scalar.* Frequently these are just typedefs of ANSI C types to a VSIPL naming convention. For instance vsip_scalar_f is an ANSI C float (float), and vsip_scalar_i is an ANSI C integer (int). The author feels it is important that people use the VSIPL types in their programing. Because every function within VSIPL is strongly typed this will keep help keep you honest and will reduce errors. In addition if the need arises to port some code from say a float library to a double library it simplifies the porting. This will become more obvious as you learn more about the library. In VSIPL Core Light there are scalars of type vector index, boolean, float, integer, and complex float.

The next base type is a *block.* A block is equivalent to a memory storage area of a particular data type and some size. As far as the application programer is concerned the data is stored in sequential element locations. In TASP VSIPL a block object is actually an abstract data type (ADT) with variables to hold the block length, information about the block's state (more about

this later) and a pointer to some physical memory. For complex blocks (in TASP VSIPL) there are actually two pointers to two real blocks, and information concerning the data layout of those blocks. All you need to know about blocks to program portable code is that from the point of view of the VSIPL functions that work on blocks they are a chunk of sequential VSIPL elements of a particular type, the first element location being at 0 (zero) and the last element being at $N-1$ where N is the size of the block. In TASP VSIPL Core Light there are only blocks of type float, complex float, and integer.

The next, and final basic type, is a *view.* In TASP VSIPL Core Light, the same as for blocks, there are only vector views of type float, complex float, and integer. The view, similar to the block, is an abstract data type. The view holds all the information needed to access some particular portion of a blocks memory. For the TASP VSIPL implementation the view has a block pointer which is set equal to the block whose data it references. All data is referenced through this block pointer. In addition the view holds an offset from the beginning of the block (starting at zero), a length (of the vector) and a stride through the block. The stride indicates the distance between consecutive view elements within the block. A stride of 1 is every element, a stride of -1 is also every element, but the view goes through the block in the opposite direction. A stride of zero will select a particular element as a constant vector at the offset location.

In order to produce portable VSIPL application code the application programer must use only VSIPL function calls to use or modify *blocks* or *views*. To enforce this all the abstract data types used by VSIPL for its internal workings are created as incomplete data types. Because of this, unless the private header files where the blocks and view data types are completed are available, the user must use VSIPL function calls. The use of abstract data types, and incomplete type definitions requires the use of *view* and *block* objects.

For more information on VSIPL design requirements see Appendix A, or obtain the VSIPL specification (when it is completed).

A Simple First Example

Except for the list of functions in Chapter 2, and Appendices which may cover any topic, the rest of this document will be done in a tutorial fashion. In general the method used will be to produce a simple example with an exhaustive explanation. Within the explanation important principles for successful VSIPL programs will be explained. All examples in this document will be limited to code that will compile on VSIPL libraries that conform to the Core Lite Profile

Add two vectors example.

If this were being done in Matlab this example would look as follows:

```
>> A=[0:7]
A =
   0   1   2   3   4   5   6   7
>> B(A+1)=5
B =
   5   5   5   5   5   5   5   5
>> C=A+B
```

C =

5 6 7 8 9 10 11 12

Now let's do this in VSIPL Code:

Example 1

```
1     #include<stdio.h>
2     #include<vsip.h>
3
4     #define N 8 /* the length of the vector */
5
6     int main()
7     {
8        void VU_vprint_f(vsip_vview_f*);
9        vsip_vview_f   *A = vsip_vcreate_f(N,0),
10                      *B = vsip_vcreate_f(N,0),
11                      *C = vsip_vcreate_f(N,0);
12       vsip_vramp_f(0,1,A);
13       printf("A = \n");VU_vprint_f(A);
14
15       vsip_vfill_f(5,B);
16       printf("B = \n");VU_vprint_f(B);
17
18       vsip_vadd_f(A,B,C);
19       printf("C = \n");VU_vprint_f(C);
20
21       vsip_valldestroy_f(A);
22       vsip_valldestroy_f(B);
23       vsip_valldestroy_f(C);
24       return 1;
25    }
26
27    void VU_vprint_f(vsip_vview_f* a){
28       int i;
29       vsip_vattr_f attr;
30       vsip_vgetattrib_f(a,&attr);
31       for(i=0; i<attr.length; i++)
32       printf("%4.0f",vsip_vget_f(a,i));
33       printf("\n");
34       return;
35    }
```

The above program produces the following output:

```
A =
   0   1   2   3   4   5   6   7
B =
```

```
   5   5   5   5   5   5   5   5
C =
   5   6   7   8   9  10  11  12
```

Let's examine Example 1.

On **line 2** we include the **`vsip.h`** header file. This will be needed in every program using VSIPL code, and a compliant library is required to provide a header file called vsip.h.

On **line 9** we have our first VSIPL function call. Note that the VSIPL vector view objects A, B and C are type defined to a pointer of type **`vsip_vview_f`** and then assigned a value by the function **`vsip_vcreate_f`**.

The **`create`** function is a convenience function, subsuming the block create and vector bind jobs into one function. The **`vsip_vcreate_f`** function will create a block of type real float (with some state we talk about in chapter 3), create a data space of sufficient size to hold N real float values and then attaches this to the block, and creates a vector view of type real float and binds the block to this view. The vector view (or just vector) is created with a length of **`N`** elements, an offset of zero, and a stride of one, so that the vector is of an exact size to view the entire block.

An important item to note here is that VSIPL has allocated space in memory for three items. These are the space for the block object (block ADT), space for the data storage (the data array) and space for the vector object (vector ADT). All of this memory must be destroyed when no longer needed to prevent memory leaks.

Blocks and associated data arrays created by the VSIPL create functions are always created and destroyed together. Whenever you do a block create and a block destroy the actions to create the block, and its associated data, or destroy a block and its associated data happen together. Generally we will only say we create a block of length N, or destroy a block.

We have also created a vector view. There may be many vector views associated with a block. There are, of course, functions to destroy a vector view. These will generally not destroy the block also. It is important that the application programer keep track of what views are attached to a block and only destroy the block after all the views binding the block have been destroyed.

Note that all VSIPL functions which allocate memory return a null pointer if the memory allocation fails. We have not checked for an allocation failure in our example, but the check is recommended.

In our example above on **lines 21-23** we see where the object destruction takes place. We know, since this program is short and we kept track, that each block is bound by exactly one view. To destroy our objects we use a convenience function which will destroy the view, the block, and any VSIPL allocated data array associated with the block.

The other important items in our code reside on **lines 12**, **15**, and **18**. These are self explanatory as to function, but note that no stride, length or offset information are included in the arguments to these functions. All this information resides in the vector. So, for instance, the ramp function has a starting value of zero, an increment of one, but no stopping point. The

function just goes until the vector is full. Since we set the vector length to eight, we get a vector running from zero to seven.

In order to see our output we wrote a vector print function. For VSIPL user functions the author uses a prefix of VU_ for VSIPL User. The print function starts at **line 27**. We note the important functions **vsip_vgetattrib_f**, and **vsip_vget_f**, both of whose uses are obvious. The authors opinion is it was a mistake not to include VSIPL support functions getlength, getstride, and getoffset in the Core Lite profile, since the getattrib function is a bit of a pain to use; however using the getattrib function it is easy to write the convenience functions for ourselves if we so desire. In this manual we will always use getattrib since it is the one included in the Core Lite profile.

List of Acronyms

1.	TASP	Tactical Advanced Signal Processing
2.	COE	Common Operating Environment
3.	VSIPL	Vector/Signal/Image Processing Library
4.	COTS	Commercial Of the Shelf
5.	TAC	Tactical Advanced Computer
6.	API	Application Program Interface
7.	ADT	Abstract Data Type
8.	DOD	Department of Defense

CHAPTER 2 Core Lite Functions

Introduction

This section provides an alphabetical listing of all VSIPL functionality included with the TASP VISIPL Core Lite Library. Each function is listed with a functionality statement, the function prototype, and a description of each function argument. The purpose of the chapter is as a reference for VSIPL Core Lite Function calls.

In the VSIPL specification many of the functions names have been generalized to include all precisions. The names in this document are not generalized and reflect the TASP VSIPL Core Lite implementation.

In order to have some reasonable ordering of the functions the alphabetical listing is based upon a root function name, not the actual VSIPL function. For instance the first function in the list is the “add” function. There are actually three add functions in the Core Lite profile. These are add two real float vectors (**`vsip_vadd_f`**), add two complex float vectors (**`vsip_cvadd_f`**) and add a real float scalar to a real float vector (**`vsip_svadd_f`**). All of these functions are placed together under add.

When a function requires a special object it needs support functions to create the object, and destroy it, and perhaps query it for its attributes. For instance to do a discrete fourier transform one needs a function to create an FFT object, a function to do the actual FFT using the FFT object, and a function to destroy the FFT object when it is no longer needed. The author calls function which are designed to work together to do a single job function sets. Function sets are placed together under a single heading. For instance all the functions involved with doing an FFT are placed under the FFT heading.

No attempt is made to be exhaustive in the function descriptions. All the functions included in TASP VSIPL Core Lite are briefly described. Those interested in more detail are directed to the VSIPL specification document available on the internet site.(www.vsipl.org)

Core Lite Function List

add

Scalar vector add

```
void vsip_svadd_f(
   vsip_scalar_f a1,
   vsip_vview_f* a2,
   vsip_vview_f* a3);
```

Argument a1 Input scalar.

Argument a2 Input vector.

Argument a3 Sum of scalar and vector elementwise.

Add two vectors element by element.

```
void vsip_vadd_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_f* a3);
void vsip_cvadd_f(
   const vsip_cvview_f* a1,
   const vsip_cvview_f* a2,
   const vsip_cvview_f* a3);
```

Argument a1 Input vector

Argument a2 Input vector

Argument a3 Sum of input vectors

alldestroy

Function to destroy a vector view and its associated block. If the block is bound to a user data array then the user data array is not destroyed.

```
void vsip_valldestroy_f(
   vsip_vview_f* a1);
void vsip_cvalldestroy_f(
   vsip_cvview_f* a1);
```

Argument a1 The pointer to the vector view to be destroyed.

atan

Elementwise arctangent of a vector. This performs elementwise the ANSI C math atan function.

```
void vsip_vatan_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2);
```

Argument a1 Input vector of tangent values.

Argument a2 Output vector of arctangent values.

atan2

Elementwise arctangent of two vectors. This is the same as the ANSI C math function atan2.

```
void vsip_vatan2_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_f* a3);
```

Argument a1 Input vector denominator

Argument a2 Input vector numerator

Argument a3 Output vector of arctangent values of the quotient in radians.

bind

Used to create a vector view object and bind it to a block.

```
vsip_vview_f* vsip_vbind_f(
   const vsip_block_f *a1,
   vsip_offset a2,
   vsip_stride a3,
   vsip_length a4);
vsip_cvview_f* vsip_cvbind_f(
   const vsip_cblock_f *a1,
   vsip_offset a2,
   vsip_stride a3,
   vsip_length a4);
vsip_vview_i* vsip_vbind_i(
   const vsip_block_i *a1,
   vsip_offset a2,
   vsip_stride a3,
   vsip_length a4);
```

Returns A pointer to the vector view object created. Returns null on creation failure.

Argument a1 The block bound.

Argument a2 The offset from the beginning of the block where the view starts. Offsets are zero based and positive so that an offset of zero is the first element of the block.

Argument a3 The stride through the block. This indicates the number of elements in the block between vector view elements. A stride of zero will access only the element indicated by the offset, and a stride of 1 will access consecutive elements. A stride of N will access every Nth element. Strides may be negative indicating a direction of movement through the block opposite to that of a positive stride.

Argument a4 The length of the vector view in terms of elements. The length is not zero based and a length of 1 indicates 1 element, and a length of N indicates N elements. The length is always greater than zero.

blockadmit

Admit a block connected to user data for use by VSIPL functions. This function is used to change the state of a VSIPL user block from released to admitted.

```
int vsip_blockadmit_f(
   vsip_block_f* a1,
   vsip_scalar_bl a2);
int vsip_cblockadmit_f(
   vsip_cblock_f* a1,
   vsip_scalar_bl a2);
int vsip_blockadmit_i(
   vsip_block_i* a1,
   vsip_scalar_bl a2);
```

Returns If the block admission succeeds a 0 (zero) is returned. A nonzero value indicates a failure.

Argument a1 A block pointer for an instantiated (valid) block. The admission will fail if the block is bound to a null data pointer.

Argument a2 A boolean flag. True indicates the value of the data must be maintained during the state change.

blockbind

This function creates a block and binds it to a user defined memory pointer. The pointer defines the beginning of some user defined data array. It is the responsibility of the user to ensure the memory pointer has enough data allocated with it for the desired number of block elements.

```
vsip_block_f* vsip_blockbind_f(
   const vsip_scalar_f a1,
   vsip_length a3,
   vsip_memory_hint a4);
vsip_cblock_f* vsip_cblockbind_f(
   const vsip_scalar_f a1,
   const vsip_scalar_f a2,
   vsip_length a3,
   vsip_memory_hint a4);
vsip_block_f* vsip_blockbind_i(
   const vsip_scalar_i a1,
   vsip_length a3,
   vsip_memory_hint a4,);
```

Returns Pointer to created block.

Argument a1 Pointer to user defined data array. For complex blocks this pointer will point to a single interleaved data array, or to the data array defined for real split data.

Argument a2 Pointer to user defined data array for imaginary complex data, if the split format is used, or to the null data pointer if interleaved complex is used.

Argument a3 Number of elements of the block type associated with the user data array(s).

Argument a4 This is ignored in TASP VSIPL implementation. Place a 0 (zero) here or use any enumerated memory hint defined in VSIPL.

blockcreate

This function creates a block object. The block creation includes allocating memory for the data associated with the block

```
vsip_block_f* vsip_blockcreate_f(
   vsip_length a1,
   vsip_memory_hint a2);

vsip_cblock_f* vsip_cblockcreate_f
   vsip_length a1,
   vsip_memory_hint a2);

vsip_block_f* vsip_blockcreate_i(
   vsip_length a1,
   vsip_memory_hint a2);
```

Returns Pointer to created block.

Argument a1 Number of elements of the block type to be created and attached to the block. This is the block size, or the length of the block.

Argument a2 This is ignored in TASP VSIPL implementation. Place a 0 (zero) here or use any enumerated memory hint defined in VSIPL.

blockdestroy

Destroy a block and any data bound to the block which was allocated by VSIPL. User data bound to the block is not destroyed.

```
void vsip_blockdestroy_f(
   vsip_block_f *a1);

void vsip_cblockdestroy_f(
   vsip_cblock_f *a1);

void vsip_blockdestroy_i(
   vsip_block_i *a1);
```

Argument a1 Block to be destroyed;

blockfind

Find the pointer to the user data bound to a VSIPL released block.

```
vsip_scalar_f* vsip_blockfind_f(
   const vsip_block_f* a1);

void vsip_cblockfind_f(
   const vsip_cblock_f* a1,
   vsip_scalar_f* *a2,
   vsip_scalar_f* *a3);

vsip_scalar_i* vsip_blockfind_i(
   const vsip_block_i* a1);
```

Returns Pointer to the user data array bound to the block, or void for complex blocks.

Argument a1 User released block

Argument a2 For complex, the data pointer to the user real data if split, or to the complex data if interleaved.

Argument a3 For complex, null if the user complex data is interleaved, and a pointer to the user imaginary data if split.

blockrebind

Bind an existing VSIPL user block to a new data array.

```
vsip_scalar_f* vsip_blockrebind_f(
   vsip_block_f* a1,
   const vsip_scalar_f* a2);

void vsip_cblockrebind_f(
   vsip_cblock_f* a1,
   const vsip_scalar_f* a2,
   const vsip_scalar_f* a3,
   vsip_scalar_f* *a4,
   vsip_scalar_f* *a5);

vsip_scalar_i* vsip_blockrebind_i(
   vsip_block_f* a1,
   const vsip_scalar_f* a2);
```

Returns Except for complex, returns a pointer to the user data array bound to the block before the rebind. Returns void if the block is complex.

Argument a1 Pointer to block to be rebound.

Argument a2 Pointer to new data array to be bound to the user block. If the block is complex this data array is the real part of the complex number if the layout to be bound is split.

Argument a3 A null pointer if the user complex data layout is interleaved, or a pointer to a data array encompassing the imaginary portion of the complex number if the data layout is split

Argument a4 A pointer to the previous real complex data array if the previous user complex data was split, or a pointer to the previous user interleaved complex data array.

Argument a5 A null pointer if the previous user data array was interleaved, or a pointer to the imaginary portion of the previous split complex user data array.

blockrelease

Release a user block. This function is used to change the state of a VSIPL user block from admitted to released.

```
vsip_scalar_f* vsip_blockrelease_f(
   vsip_block_f* a1,
   vsip_scalar_bl a2);

void vsip_cblockrelease_f(
   vsip_cblock_f * a1,
   vsip_scalar_bl a2,
   vsip_scalar_f* *a3
   vsip_scalar_f* *a4);

vsip_scalar_i* vsip_blockrelease_i(
   vsip_block_i* a1,
   vsip_scalar_bl a2);
```

Returns Pointer to public data array, or void for complex.

Argument a1 Pointer to block to be released.

Argument a2 A boolean flag. True indicates the value of the data must be maintained during the state change.

Argument a3 For complex user data a pointer to the interleaved user data, or to the real part of the complex user data for split representation.

Argument a4 For complex a null data pointer for the interleaved representation, and a pointer to the imaginary data array for split representation.

cloneview

Creates a new vector view object with all the attributes of the parent object.

```
vsip_vview_f* vsip_vcloneview_f(
   const vsip_vview_f* a1);
vsip_cvview_f* vsip_cvcloneview_f(
   const vsip_cvview_f* a1);
```

Returns A pointer to the new vector view.

Argument a1 The vector view to be cloned.

cmplx

Create a complex vector from two real vectors.

Scalar complex.

```
void vsip_CMPLX_f(
   vsip_scalar_f a1,
   vsip_scalar_f a2,
   vsip_scalar_f* a3);
```

Argument a1 An input scalar representing the real part.

Argument a2 An input scalar representing the imaginary part.

Argument a3 An7 output scalar representing the complex number.

```
vsip_cscalar_f vsip_cmplx_f(
   vsip_scalar_f a1,
   vsip_scalar_f a2);
```

Returns A complex scalar.

Argument a1 A scalar representing the real part

Argument a2 A scalar representing the imaginary part

Vector complex.

```
void vsip_vcmplx_f(
   vsip_vview_f* a1,
   vsip_vview_f* a2,
   vsip_cvview_f* a3);
```

Argument a1 Input vector representing the real part

Argument a2 Input vector representing the imaginary part.

Argument a3 The complex output vector.

copy

The copy function copies data from one vector to another vector. This function is used to convert data types.

```
void vsip_vcopy_f_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2);

void vsip_vcopy_f_i(
   const vsip_vview_f* a1,
   const vsip_vview_i* a2);

void vsip_vcopy_i_f(
   const vsip_vview_i* a1,
   const vsip_vview_f* a2);

void vsip_cvcopy_f_f(
   const vsip_cvview_f* a1,
   const vsip_cvview_f* a2);
```

Argument a1 Input vector to be copied.

Argument a2 Output vector, a copy of the input vector with possibly a data type conversion.

cos

Elementwise Cosine of a vector.

```
void vsip_vcos_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2);
```

Argument a1 Input vector of angles in radian format.

Argument a2 Output vector of Cosine values.

create

Convenience function to create the view, and the block and data associated with the block all at the same time. The created view accesses the entire block with an offset of zero, a stride of one, and a length equal to the block size.

```
vsip_vview_f* vsip_vcreate_f(
   vsip_length a1,
   vsip_memory_hint a2);

vsip_cvview_f* vsip_cvcreate_f(
   vsip_length a1,
   vsip_memory_hint a2);
```

Returns Pointer to vector view requested.

Argument a1 Length of the vector view

Argument a2 This is ignored in TASP VSIPL implementation. Place a 0 (zero) here or use any enumerated memory hint defined in VSIPL.

cstorage

Indicates the preferred method of complex storage for user data in a particular VSIPL implementation.

```
vsip_cmplx_mem vsip_cstorage(
   void);
```

Returns A value based on the enumerated typedef

```
typedef enum {
   VSIP_CMPLX_INTERLEAVED,
   VSIP_CMPLX_SPLIT,
   VSIP_CMPLX_NONE
   } vsip_cmplx_mem
```

cvconj

Conjugate a complex vector.

```
void vsip_cvconj_f(
   const vsip_cvview_f* a1,
   const vsip_cvview_f* a2);
```

Argument a1 Input vector.

Argument a2 Output vector.

destroy

Function to destroy a vector view

```
vsip_block_f* vsip_vdestroy_f(
   vsip_vview_f* a1);
vsip_cblock_f* vsip_cvdestroy_f(
   vsip_cvview_f* a1);
vsip_block_i* vsip_vdestroy_i(
   vsip_vview_i* a1);
```

Returns A pointer to the block the view was bound to.

Argument a1 The pointer to the vector view to be destroyed

div

Divide two vectors element by element.

```
void vsip_vdiv_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_f* a3);
```

Argument a1 The numerator input vector.

Argument a2 The denominator input vector.

Argument a3 The quotient output vector.

Scalar vector divide.

```
void vsip_svdiv_f(
   vsip_scalar_f a1,
   vsip_vview_f* a2,
   vsip_vview_f* a3);
```

Argument a1 The numerator input scalar.

Argument a2 The denominator input vector.

Argument a3 The quotient output vector.

dot

Dot products. A dot product is an elementwise multiply of two vectors with a sum of the resulting vector.

Real Dot Product

```
vsip_scalar_f vsip_vdot_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2);
```

Returns Dot Product value

Argument a1 Real input vector

Argument a2 Real input vector

Complex Dot Product

```
vsip_cscalar_f vsip_cvdot_f(
   const vsip_cvview_f* a1,
   const vsip_cvview_f* a2);
```

Returns Dot Product value

Argument a1 Complex input vector

Argument a2 Complex input vector

Complex Conjugate Dot Product. The dot product here is done between the first input vector and the complex conjugate of the second input vector.

```
vsip_cscalar_f vsip_cvjdot_f(
   const vsip_cvview_f* a1,
   const vsip_cvview_f* a2);
```

Returns Dot Product value

Argument a1 Complex input vector

Argument a2 complex input vector

exp

Elementwise natural (base e) exponential of a vector.

```
void vsip_vexp_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2);
```

Argument a1 Input vector

Argument a2 Output vector

fft

Compute a Discrete Fourier Transform using FFT methods for radices of 2 and at least one factor of 3 as a minimum. The current TASP VSIPL FFT uses building block factors of 2, 4, 8, 5, 3, and 7.

Create an FFT object for doing a complex input vector to complex output vector FFT calculation.

```
vsip_fft_f* vsip_ccfftop_create_f(
   vsip_length a1,
   vsip_scalar_f a2,
   vsip_fft_dir a3,
   unsigned int a4,
   vsip_alg_hint a5);
```

Returns FFT object useful for creating a (user selected direction) forward or inverse FFT, or null on creation failure.

Argument a1 Length of input vector.

Argument a2 A scale factor. If a scale factor of 1 is used for a forward FFT then a scale factor of $1/(a1)$ in the inverse FFT will get back the original vector.

Argument a3 An enumerated type defining the direction of the FFT. You may use VSIP_FFT_FWD (-1) for the forward FFT and VSIP_FFT_INV (+1) for the inverse FFT.

Argument a4 This option is not supported in TASP VSIPL. Recommend placing zero here, although any number will work.

Argument a5 This option is not supported in TASP VSIPL, Recommend placing a zero here, although any valid algorithm hint will work.

Do a complex to complex FFT

```
void vsip_ccfftop_f(
   const vsip_fft_f* a1,
   const vsip_cvview_f* a2,
   const vsip_cvview_f* a3);
```

Argument a1 FFT object created using **`vsip_ccfftop_create_f`** for the desired FFT.

Argument a2 Complex input data

Argument a3 Complex output data

Create an FFT object for doing a real to complex FFT.

```
vsip_fft_f* vsip_rcfftop_create_f(
   vsip_length a1,
   vsip_scalar_f a2,
   unsigned int a3,
   vsip_alg_hint a4);
```

Returns FFT object useful for creating a real to complex FFT, or null on creation failure.

Argument a1 Length of the real vector input vector

Argument a2 Scale factor.

Argument a3 This option is not supported in TASP VSIPL. Recommend placing zero here, although any number will work.

Argument a4 This option is not supported in TASP VSIPL, Recommend placing a zero here, although any valid algorithm hint will work.

Do a real to complex FFT

```
void vsip_rcfftop_f(
   const vsip_fft_f* a1,
   const vsip_vview_f* a2,
   const vsip_cvview_f* a3);
```

Argument a1 FFT object created with **vsip_rcfftop_create_f**

Argument a2 Real input vector of length N, where N is even.

Argument a3 Complex output vector of length N/2 + 1.

Create an FFT object for doing a complex to real FFT.

```
vsip_fft_f* vsip_crfftop_create_f(
   vsip_length a1,
   vsip_scalar_f a2,
   unsigned int a3,
   vsip_alg_hint a4);
```

Returns FFT object useful for creating a complex to real FFT object, or null on creation failure.

Argument a1 Length of real output data.

Argument a2 Scale factor

Argument a3 This option is not supported in TASP VSIPL. Recommend placing zero here, although any number will work.

Argument a4 This option is not supported in TASP VSIPL, Recommend placing a zero here, although any valid algorithm hint will work.

Do a complex to real FFT

```
void vsip_crfftop_f(
   const vsip_fft_f* a1,
   const vsip_cvview_f* a2,
   const vsip_vview_f* a3);
```

Argument a1 FFT object created with **vsip_crfftop_create_f**.

Argument a2 Complex input vector of size N/2 + 1 where N is even.

Argument a3 Real output vector of size N.

Destroy a complex object.

```
int vsip_fft_destroy_f(
   vsip_fft_f* a1);
```

Returns On success a 0 (zero) is returned.

Argument a1 FFT object to be destroyed.

fill

Fill a vector with a constant value.

```
void vsip_vfill_f(
   vsip_scalar_f a1,
   const vsip_vview_f* a2);
```

Argument a1 Scalar value to fill output vector with.

Argument a2 Output vector

fir

Finite impulse response filter with decimation.

Finite impulse response filter object create.

```
vsip_fir_f* vsip_fir_create_f(
   const vsip_vview_f* a1,
   vsip_symmetry a2,
   vsip_length a3,
   vsip_length a4,
   vsip_obj_state a5,
   unsigned int a6,
   vsip_alg_hint a7);
vsip_cfir_f* vsip_cfir_create_f(
   const vsip_cvview_f* a1,
   vsip_symmetry a2,
   vsip_length a3,
   int a4,
   vsip_obj_state a5,
   unsigned int a6,
   vsip_alg_hint a7);
```

Returns Pointer to FIR object.

Argument a1 Vector view containing filter kernel.

Argument a2 Symmetry enumerated typedef associated with the selected kernel

Argument a3 Length of the data to be filtered at a time.

Argument a4 Decimation factor.

Argument a5 Save state (VSIP_STATE_SAVE) or don't save state (VSIP_STATE_NO_SAVE)

Argument a6 Not implemented in TASP VSIPL. Recommend placing a 0 (zero) in this spot.

Argument a7 Not implemented in TASP VSIPL. Recommend placing a 0 (zero) in this spot.

Finite impulse response filter function.

```
int vsip_firflt_f(
   vsip_fir_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_f* a3);
int vsip_cfirflt_f(
   vsip_cfir_f* a1,
   const vsip_cvview_f* a2,
   const vsip_vview_f* a3);
```

Returns The number of output samples placed in argument a3.

Argument a1 A FIR filter object

Argument a2 The input vector to be filtered

Argument a3 The output vector.

FIR filter object destruction function

```
int vsip_fir_destroy_f(
   vsip_fir_f* a1);
int vsip_cfir_destroy_f(
   vsip_cfir_f* a1);
```

Returns Returns 0 (zero) on success.

Argument a1 The FIR filter object to be destroyed.

get

Get an element from a vector

```
vsip_scalar_f vsip_vget_f(
   const vsip_vview_f* a1,
   vsip_scalar_vi a2);
vsip_cscalar_f vsip_cvget_f(
   const vsip_cvview_f* a1,
   vsip_scalar_vi a2);
```

Returns Value indexed by a2.

Argument a1 Vector view from which a value will be selected and returned.

Argument a2 Index value of desired element. The first element will have an index value of 0 (zero).

getattrib

Access function to retrieve a structure containing the attributes of a vecotor view object.

```
void vsip_vgetattrib_f(
   const vsip_vview_f* a1,
   vsip_vattr_f* a2);
void vsip_cvgetattrib_f(
   const vsip_vview_f* a1,
   vsip_cvattr_f* a2);
void vsip_vgetattrib_i(
   const vsip_vview_i* a1,
   vsip_vattr_i* a2);
```

Argument a1 Input vector whose attributes will be returned.

Argument a2 Attribute structure to be filled with attributes of input vector. The structure format is
a2->length, a2->stride, a2->offset, a2->block.

getblock

Access function to retrieve the block associated with a vector view object.

```
vsip_block_f* vsip_vgetblock_f(
   vsip_vview_f* a1);
```

```
vsip_cblock_f* vsip_cvgetblock_f(
   vsip_cvview_f* a1);
```

Returns A block object pointer.

Argument a1 The view bound to the block object being returned.

histo

Histogram function. This function uses a maximum value and a minimum value and the length of the output vector to calculate the bin size. Input values less than the minimum value are counted as belonging in the first element of the output vector and Input values greater than the maximum value are counted as belonging in the last element of the output vector.

```
void vsip_vhisto_f(
   const vsip_vview_f* a1,
   vsip_scalar_f* a2,
   vsip_scalar_f* a3,
   const vsip_vview_f* a4);
```

Argument a1 Input vector of values for which a histogram is desired.

Argument a2 Minimum value for which elements less than are counted in the first output element.

Argument a3 Maximum value for which elements greater than are counted in the last output element

Argument a4 Output vector of histogram counts.

imag

Copy the imaginary elements of a complex vector to a real vector.

Scalar imaginary part.

```
vsip_scalar_f vsip_imag_f(
   vsip_csclar_f a1);
```

Returns The imaginary part.

Argument a1 The input complex scalar.

Vector imaginary part.

```
void vsip_vimag_f(
   const vsip_cvview_f* a1,
   const vsip_vview_f* a2);
```

Argument a1 Input complex vector.

Argument a2 Output vector to contain the imaginary part of the complex input vector.

imagview

Create a real view of the imaginary portion of a complex view. This is not a copy. Modifying elements in either the real view or the complex view will modify the corresponding element in the other view.

```
vsip_vview_f* vsip_vimagview_f(
   const vsip_cvview_f* a1);
```

Returns Vector view of imaginary portion of the complex view a1.

Argument a1 Complex vector view from which the real view of the imaginary part will be derived.

log

Elementwise natural (base e) logarithm of a vector.

```
void vsip_vexp_log_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2);
```

Argument a1 Input vector.

Argument a2 Output vector.

log10

Elementwise base 10 logarithm of a vector.

```
void vsip_vlog10_f*(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2);
```

Argument a1 Input vector.

Argument a2 Output vector.

mag

Elementwise find the magnitude of a vectors elements and place them in an output vector.

```
void vsip_vmag_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2);
void vsip_cvmag_f(
   const vsip_cvview_f* a1,
   const vsip_vview_f* a2);
```

Argument a1 Input vector.

Argument a2 Output vector of magnitudes.

max

Compare two vectors element by element and place the maximum value of each element comparison in an output vector.

```
void vsip_vmax_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_f* a3);
```

Argument a1 Input vector.

Argument a2 Input vector.

Argument a3 Output vector.

maxval

Find the maximum value of a vector and return it and it's index.

```
vsip_scalar_f vsip_vmaxval_f(
   const vsip_vview_f* a1,
   vsip_scalar_vi* a2);
```

Returns Maximum value of the input vector

Argument a1 Input Vector.

Argument a2 If a2 is not a null value the index of the maximum value is returned.

min

Compare two vectors element by element and place the minimum value of each element comparison in an output vector.

```
void vsip_vmin_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_f* a3);
```

Argument a1 Input vector

Argument a2 Input vector

Argument a3 Output vector

minval

Find the minimum value of a vector and return it and it's index.

```
vsip_scalar_f vsip_vminval_f(
   const vsip_vview_f* a1,
   vsip_scalar_vi* a2);
```

Returns Minimum value of the input vector.

Argument a1 Input vector.

Argument a2 If a2 is not a null value then the index of the minimum value is returned.

mul

Multiply two vectors element by element.

Multiply two real vectors elementwise.

```
void vsip_vmul_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_f* a3);
```

Argument a1 Real input vector data.

Argument a2 Real input vector data.

Argument a3 Real output vector data.

Multiply two complex vectors elementwise.

```
void vsip_cvmul_f(
   const vsip_cvview_f* a1,
   const vsip_cvview_f* a2,
   const vsip_cvview_f* a3);
```

Argument a1 Complex input vector data.

Argument a2 Complex input vector data.

Argument a3 Complex output vector data

Multiply a real scalar times a real vector elementwise.

```
void vsip_svmul_f(
   vsip_scalar_f a1,
   const vsip_vview_f* a2,
   const vsip_vview_f* a3);
```

Argument a1 Real input scalar.

Argument a2 Real input vector data.

Argument a3 Real output vector data.

Complex scalar, complex vector elementwise multiply

```
void vsip_csvmul_f(
   vsip_cscalar_f a1,
   const vsip_cvview_f* a2,
   const vsip_cvview_f* a3);
```

Argument a1 Complex input scalar.

Argument a2 Complex input vector data.

Argument a3 Complex output vector data.

Real vector, complex vector elementwise multiply.

```
void vsip_rcvmul_f(
   const vsip_vview_f* a1,
   const vsip_cvview_f* a2,
   const vsip_cvview_f* a3);
```

Argument a1 Real input vector data.

Argument a2 Complex input vector data.

Argument a3 Complex output vector data.

Real scalar, complex vector elementwise multiply.

```
void vsip_rscvmul_f(
   vsip_scalar_f a1,
   const vsip_cvview_f* a2,
   const vsip_cvview_f* a3);
```

Argument a1 Input scalar.

Argument a2 Complex input vector data.

Argument a3 Complex output vector data.

Complex vector conjugate multiply. Elementwise multiply the first input vector times the conjugate of the second input vector.

```
void vsip_cvjmul_f(
   vsip_cvview_f* a1,
   vsip_cvview_f* a2,
   vsip_cvview_f* a3);
```

Argument a1 The first input vector.

Argument a2 The second input vector.

Argument a3 The output vector.

neg

Elementwise perform an unary minus on a vector.

```
void vsip_vneg_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2);
void vsip_cvneg_f(
   const vsip_cvview_f* a1,
   const vsip_cvview_f* a2);
```

Argument a1 Input vector

Argument a2 Output vector

put

Put an element into a vector.

```
void vsip_vput_f(
   const vsip_vview_f* a1,
   vsip_scalar_vi a2,
   vsip_scalar_f a3);
void vsip_cvput_f(
   const vsip_vview_f* a1,
   vsip_scalar_vi a2,
   vsip_cscalar_f a3);
```

Argument a1 Vector view into which an element will be placed.

Argument a2 Index value of desired element. The first element will have an index value of 0 (zero).

Argument a3 Value to place in vector.

putattrib

Access function to set the attributes of a vector view object using an attributes structure.

```
vsip_vview_f* vsip_vputattrib_f(
   vsip_vview_f* a1,
   const vsip_vattr_f* a2);
```

```
vsip_cvview_f* vsip_cvputattrib_f(
   vsip_vview_f* a1,
   const vsip_cvattr_f* a2);
vsip_vview_i* vsip_vputattrib_i(
   vsip_vview_i* a1,
   const vsip_vattr_i* a2);
```

Returns Pointer to input vector as a convenience.

Argument a1 Input vector whose attributes will be modified.

Argument a2 Attribute structure to be filled with attributes of input vector. The structure format is
a2->length, a2->stride, a2->offset, a2->block.
Note that the block is ignored when putting an attribute.

putoffset

Access function to set the offset of a vector view object

```
vsip_vview_f* vsip_vputoffset_f(
   vsip_vview_f* a1,
   vsip_offset a2);
vsip_cvview_f* vsip_cvputoffset_f(
   vsip_cvview_f* a1,
   vsip_offset a2);
```

Returns Pointer to input vector as a convenience.

Argument a1 Vector whose offset is to be reset.

Argument a2 Offset value. An offset of 0 (zero) will place the offset at the first element of the block.

putstride

Access function to set the stride of a vector view object.

```
vsip_vview_f* vsip_vputstride_f(
   vsip_vview_f* a1,
   vsip_stride a2);
vsip_cvview_f* vsip_cvputstride_f(
   vsip_cvview_f* a1,
   vsip_stride a2);
```

Returns Pointer to input vector as a convenience.

Argument a1 Vector whose stride is to be reset.

Argument a2 Stride value. Strides may be positive, negative or zero.

putlength

Access function to set the length of a vector view object

```
vsip_vview_f* vsip_vputlength_f(
   vsip_vview_f* a1,
   vsip_length a2);
```

```
vsip_cvview_f* vsip_cvputlength_f(
   vsip_cvview_f* a1,
   vsip_length a2);
```

Returns Pointer to input vector as a convenience.

Argument a1 Vector whose length is to be reset.

Argument a2 Length value.

ramp

Fill a vector with an initial value plus some increment times the vector index.

```
void vsip_vramp_f(
   vsip_scalar_f a1,
   vsip_scalar_f a2,
   const vsip_vview_f* a3);
```

Argument a1 Starting value of ramp.

Argument a2 Ramp increment value.

Argument a3 Output vector containing ramp values.

rand

Generate a uniform random sequence. The protable random number generator is described in the VSIPL specification. The current non-portable sequence in the TASP VSIPL implementation is based on the congruential sequence

$$X_n = [(1664525)X_{n-1} + 1013904223] \bmod (2^{32}).$$

The number produced is normalized to a float value between zero and one.

Create a random state object.

```
vsip_randstate* vsip_randcreate(
   unsigned int a1,
   unsigned int a2,
   unsigned int a3,
   vsip_rng a4);
typedef enum {
   VSIP_PRNG = 0,
   VSIP_NPRNG = 1
   } vsip_rng;
```

Returns A random state object

Argument a1 The initial seed value for the random state object.

Argument a2 The total number of independent random state objects needed.

Argument a3 The particular random state object needed out of the number specified in argument a2.

Argument a4 A flag to indicate using either the VSIPL specified random number generator (PRNG) or an implementation random number generator (NPRNG).

Create the next random number from random state object.

```
vsip_scalar_f vsip_randu_f(
   vsip_randstate* a1);
```

Returns A random number

Argument a1 Random state object, created by randstate.

Create a vector of random numbers from the random state object.

```
void vsip_vrandu_f(
   vsip_randstate* a1,
   const vsip_vview_f* a2);
```

Argument a1 The random state operator

Argument a2 A vector view to be filed with sequential numbers from the generator specified by the randstate object.

Destroy the random state object

```
int vsip_randdestroy(
   vsip_randstate* a1);
```

Returns 0 (zero) on success

Argument a1 The random state object to be destroyed.

real

Copy the imaginary elements of a complex vector to a real vector.

Scalar real part.

```
vsip_scalar_f vsip_real_f(
   vsip_csclar_f a1);
```

Returns The real part.

Argument a1 The input complex scalar.

Vector real part.

```
void vsip_vreal_f(
   const vsip_cvview_f* a1,
   const vsip_vview_f* a2);
```

Argument a1 Input complex vector.

Argument a2 Output vector to conatain the real part of the input complex vector.

realview

Create a real view of the real portion of a complex view. This is not a copy. Modifying elements in either the real view or the complex view will modify the corresponding element in the other view.

```
vsip_vview_f* vsip_vrealview_f(
   const vsip_cvview_f* a1);
```

Returns Vector view of the real portion of the complex view a1.

Argument a1 Complex vector view from which the real view of the real part will be derived.

recip

Elementwise find the reciprocal of a vectors elements and place them in an output vector.

```
void vsip_vrecip_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2);
```

Argument a1 Input vector

Argument a2 Output vector

sin

Elementwise Sine of a vector

```
void vsip_vsin_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2);
```

Argument a1 Input vector in radian format.

Argument a2 output vector of Sine values.

sq

Elementwise find the square of a vectors elements

```
void vsip_vsq_f(
   const vsip_vview_f* a1,
   cons vsip_vview_f* a2);
```

Argument a1 Input vector.

Argument a2 Output vector of element squares from the input vector.

sqrt

Elementwise square root of a vector.

```
void vsip_vsqrt_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2);
```

Argument a1 Input vector.

Argument a2 Output vector.

sub

Subtract two vectors element by element

```
void vsip_vsub_f(
   const vsip_vview_f* a1,
   const vsip_vview_f* a2,
   const vsip_vview_f* a3);
void vsip_cvsub_f(
   const vsip_cvview_f* a1,
```

```
   const vsip_cvview_f* a2,
   const vsip_cvview_f* a3);
```

Argument a1 Input argument

Argument a2 Input argument, subtracted from a1.

Argument a3 Output vector.

subview

Creates a new vector view object from a parent object with a selected subset of the data of the parent object accessed by the new view.

```
vsip_vview_f* vsip_vsubview_f(
   const vsip_vview_f* a1,
   vsip_index a2,
   vsip_length a3);

vsip_cvview_f* vsip_cvsubview_f(
   const vsip_cvview_f* a1,
   vsip_index a2,
   vsip_length a3);
```

Returns Pointer to the new vector view object

Argument a1 Input vector.

Argument a2 Index of element in a1 starting the new vector view. The first element is index 0 (zero).

Argument a3 Length of new output vector view.

sumsqval

Sum all the squares of the elements of a vector and return the sum.

```
vsip_scalar_f* vsip_sumsqval_f(
   const vsip_vview_f* a1);
```

Returns Input vector elements squared and summed.

Argument a1 Input vector

sumval

Sum all the elements of a vector and return the sum.

```
vsip_scalar_f* vsip_vsumval_f(
   const vsip_vview_f* a1);
```

Returns Sum of input vector values.

Argument a1 Input Vector.

vcmagsq

Vector complex magnitude squared. For a complex vector find the magnitude squared value of each element.

CHAPTER 3 Introduction to VSIPL Programming using the Core Lite Profile

Introduction

This chapter introduces programing methods using VSIPL in general, and the VSIPL Core Lite function set in particular. The examples will be simple. Following each example will be some explanatory text.

Support Functions

The support functions are those functions used to make or destroy VSIPL objects, copy data, modify object properties (such as stride, length and offset) or do input and output from VSIPL. The input and output functionality of VSIPL will be handled in its own section since it is a complicated topic.

Block Creation

The base method for block creation is the function **`blockcreate`**. This function takes a size argument, which indicates how many elements of the block type to create, and a VSIPL hint indicating how the data will be used by the program. TASP VSIPL Core Lite does not support this hint.

Generally examples will have a zero for creation hints since they are not supported by the TASP version of the library; however if the programer expects to develop code using TASP VSIPL on a workstation and then compile the workstation code on an embedded product, then it is recommended the programer use a VSIPL hint if it is supported by the embedded product. All VSIPL implementations are required to ignore the hint if it is not supported, so using an unsupported hint is harmless.

Vector Creation

The base method for vector creation is the **`bind`** function. This function creates a vector object, binds a block to the vector, and sets the stride, offset, and length of the vector to view the required data within the block. Example 2 below is a code segment to create a real float block and a complex float block and attach a vector to each block.

Example 2

```
1   /* Create a block and bind a vector to it */
2   vsip_block_f *a = vsip_blockcreate_f(10,0);
3   vsip_cblock_f *b = vsip_cblockcreate_f(10,0);
4   vsip_vview_f *v_a = vsip_vbind_f(a,0,1,10);
5   vsip_cvview_f *cv_b = vsip_cvbind_f(b,0,1,10);
```

Notice that we create a block in **lines 2** and **3** each of size 10 elements, but the elements of block **a** are real and the elements of block **b** are complex. In **lines 4** and **5** we define a real view **v_a** and a complex view **cv_b**. The view is created with the **bind** function, and we set the offset to zero, the stride to one, and the length to ten. Notice that the vectors we create here encompass the entire block, and could just as well have been created with the convenience **create** function used in Example 1.

Examination of the type definitions used for the block and vector views, and the function name of the **blockcreate** functions is worthwhile in order to develop an understanding of the VSIPL naming convention. The *precision* of the data here is float and is indicated with an **_f** prefix. The depth of the data is either real (understood in the name) or complex, indicated with a **c** prefix on the root name.

Other methods of view creation and view modification.

There are several methods of view creation. We will cover some of these in this section, and also some methods for view modification. It is frequently preferable to modify a view since no create needs to take place.

A new view of a block may always be created using the bind function. Each time this is done memory allocation takes place and the new view must be destroyed when no longer needed. Example 3 is how to use a current view and vector bind to create a new view. Lets say we want a vector view of every other element of an available view.

Example 3

```
1   {
2      vsip_vattr_f attr;
3      vsip_vview_f *b;
4      vsip_vgetattrib_f(a,&attr);
5      b = vsip_vbind_f(attr.block,
6          attr.offset,2 * attr.stride,attr.length/2));
7      /* do something with b */
8      vsip_vdestroy_f(b);
9   }
```

We note that the vector view **a** resides outside the curly brackets. Since we don't know the stride and length of **a** we use the **getattrib** function in **line 4** to retrieve that information. In addition to offset, stride, and length getting the attribute structure also gets the block object. The **bind** function creates a view and binds the block to it. We set the offset to the same offset as **a**, however we only want every other point of the vector **a** so we set the stride to double the

stride of **a**. Note that if we just set the stride to two we would get every other point in the block **a** was attached to, and not every other point of **a** unless **a** happens to have a stride of one. Finally we set the length of the vector. We do not know if the length of **a** is even or odd, but length is some sort of unsigned integer, so division by 2 will result in an unsigned integer of the floor of the division, which is the number we need.

There are several ways to accomplish the same thing we accomplished in example 3. For instance in example 4 we demonstrate the same affect, but use **cloneview** instead of **bind** and for a change of pace we make the vector **a** complex.

Example 4

```
1   {
2      vsip_cvattr_f attr;
3      vsip_cvview_f *b = vsip_cvcloneview_f(a);
4      vsip_cvgetattrib_f(a,&attr);
5      attr.stride *= 2;
6      attr.length /=2;
7      vsip_cvputattrib_f(b,&attr);
8      /* do something with b */
10     vsip_cvdestroy_f(b);
11  }
```

Now we notice two things here. The first is VSIPL strides and offsets are in terms of the block element type. There is no difference in calculating the length and stride for this new complex view than there was in the real view of example 3. We also see a new function **putattrib** in **line 7**. Note that putting an attribute is the opposite of getting an attribute, except that the block value of the attribute is ignored. The block of the view object is set on view creation in **line 3**. The block attribute of a view is always set when the view is created, and it is not possible to reset the views block attribute. Also notice that the attribute is passed by reference (a pointer), both for getting, and putting, the attribute.

Another method to create a view is the **subview** function. The subview function takes an index into the parent view of the first element of the child view, and a length. Note that indexes are into vectors, not blocks. The stride is inherited from the original view, and there is no argument to allow resetting the stride in the **subview** function. In example 5 below we do example 4 again using a subview. This time we assume we know the stride and length of the vector **a**, and have stored them in variable **a_stride** and **a_length** respectively.

Example 5

```
1   {
2      vsip_cvview_f *b = vsip_vsubview_f(a,0,a_length/2);
3      vsip_vputstride_f(b,a_stride * 2);
4         /* do something with b */
5      vsip_cvdestroy_f(b);
6   }
```

Example 5 is shorter than example 4, but that is mostly because we already know the stride and length of the input vector. The new function to note here, besides **`subview`**, is the **`put-stride`** function on **line 3**. The Core Lite profile does not support any of the get attribute functions except **`getattrib`**, but it does support all the put attribute functions, including **`putlength`** and **`putoffset`**.

The final methods we will discuss to make views are **`realview`** and **`imagview`**. These two functions are so important that we will discuss them in their own section below.

Viewing the Real and Imaginary portions of a Complex Vector

In the elementwise function set there are two functions **`vsip_vreal_f`** and **`vsip_vimag_f`** which will copy the real or imaginary portion of a complex vector to a real vector, and another function **`vsip_vcmplx_f`** which will copy two real vector, one designated as real and one as imaginary, to a complex vector. Frequently it is desirable to operate on a complex vectors real or imaginary portions separately, but using the above function set is a lot of copying and requires extra memory allocation to allow room for the copies. What is really desirable is to be able to produce a real and imaginary view of the two parts of a complex vector in-place with no copies.

Functions in VSIPL allow one to create real and imaginary views of a complex vector. The functions are **`vsip_vimagview_f`** and **`vsip_vrealview_f`**. Producing a view of the real or imaginary part of a complex view is more involved than one might at first think. We will only look at one of the issues here. The problem is that these function create a vector view of type **`vsip_vview_f`**, a real vector view. This type view must be attached to a block of type **`vsip_block_f`**; however the complex view that the real views for the imaginary part and real part are *derived* from is bound to a complex block of type **`vsip_cblock_f`**. The first thing that must be done (by the implementation) is that a real block must be derived from the complex block which represents the data of the real or imaginary portion of the complex block. This block is termed a *derived block*.

A derived block is of the same data type as any other real block. Whether or not a block is derived from a complex block is a part of the state information kept by the block object. Derived blocks may not be destroyed. The derived block is destroyed when the complex block it is derived from is destroyed. The derived views are destroyed in the normal manner using **`vsip_vdestroy_f`**.

The only way to get a derived block is to derive a view (a *derived view*) using the **`imagview`** or **`realview`** functions. The method VSIPL uses to create the derived block is implementation dependent. These functions create a real view, bind the real view to the derived block, and set the offset, strides and lengths of the real view to view the required real or imaginary portion of the parent complex view. Although the length of the new view will be the same as the parent view, the stride and offset are implementation dependent. If these are needed for some reason the derived view must be queried.

Derived blocks may not be destroyed directly, they are destroyed when the parent complex block is destroyed. Derived blocks may be bound to new vector views. It is recommended that new views bound to derived blocks stay within the data space spanned by the original derived view.

There are some other subtle issues which we can ignore most of the time, and will ignore for this introduction. Lets do a couple of simple examples.

We may want to initialize a complex vector to zero. In the Core Lite profile there is no complex fill operation, only a real fill. Example 6 shows a method to fill a complex vector with a zero. Assume we have already produced the complex vector **a** outside the brackets.

Example 6

```
1   {  /* replacement for vsip_cvfill_f */
2      vsip_vview_f *b = vsip_vrealview_f(a);
3      vsip_vfill_f(b,0.0);
4      vsip_vdestroy_f(b);
5      b = vsip_vimagview_f(a);
6      vsip_vfill_f(b,0.0);
7      vsip_vdestroy_f(b);
8   }
```

It is important to notice that we destroy vector **b** twice, once on **line 4** and again on **line 7.** This is required. When we define the vector **b** on **line 2** we actually define a pointer of type real vector view. When we destroy the vector view in **line 4** we don't destroy the pointer, just what the pointer was pointed to which is the vector view created in **line 2**. We then create a new vector view in **line 5** and store the pointer in **b**. If we had not destroyed the object pointed to by **b** in **line 4** then in **line 5** we would have replaced the view object pointer and leaked the memory allocated for the real view object in **line 2**.

If this is clear, great. If not think about it this way. One wouldn't want to do

```
/* bad code */
float *b;
b = (float)malloc(N * sizeof(float));/* allocate memory */
b = (float)malloc(N * sizeof(float));/* leak above memory */
free((void *) b);
```

This is equivalent to what happens if you don't destroy a vector view before assigning a new vector view. Of course this is also true for blocks. With (VSIPL) blocks not destroyed properly you also leak the memory associated with the data array.

Another function that is not included with the Core Lite profile is the Euler function. Euler takes an input vector of angles (in radians) and outputs a complex vector of cosine values in the real part and sine values in the imaginary part. Example 7 shows us how to do that, and will also demonstrate some of the in-place functionality of VSIPL. In-place means to replace the input with the output. Most elementwise functions support in-place, but not all functions do. See the VSIPL specification for more details of in-place.

Example 7

```
1   vsip_cvview_f *v = vsip_cvcreate_f(N,0);
2   {/* do euler */
3      vsip_vview_f *v_r = vsip_vrealview_f(v);
4      vsip_vview_f *v_i = vsip_vimagview_f(v);
5      vsip_vramp_f(0,ft_f_2PI/(vsip_scalar_f)N,v_r);
6      vsip_vsin_f(v_r,v_i);
7      vsip_vcos_f(v_r,v_r);
8      vsip_vdestroy_f(v_r);
9      vsip_vdestroy_f(v_i)
10  }
```

In **line 1** of example 7 we create a complex vector of length **N**. We want to perform an Euler operation to fill this vector with sine and cosine values of angle arguments equally distributed between zero and 2π. In this example the input vector will stop just one increment short of 2π. In **lines 3** and **4** we produce views of the real and imaginary portion of the complex vector. In **line 5** we fill the real part of the vector **v** with angles starting at zero and incrementing by $(2\pi)/N$ until the last value of the vector **v_r** is $((N-1)/N)2\pi$. In **line 6** we place the sine of the real part values into the imaginary part of vector **v**. In **line 7** we replace the angles in the real part of the vector **v** with their cosine values. **Line 7** is the in-place operation. In **line 8** and **9** we destroy the views created in **lines 3** and **4** since they are no longer needed. Note that destroying these views does not destroy the data. The data is stored in the complex block and is still viewed by the complex vector **v.**

VSIPL Input and Output Methods

Since VSIPL blocks are created using incomplete type definitions it is not possible to manipulate the data array directly. There is no method within VSIPL to retrieve a pointer to any data memory which was created using **create** or **blockcreate.** Blocks created by these functions are termed VSIPL blocks.

It is important to get data into or out of VSIPL in order to communicate with other processes or to manipulate the data directly for some purpose. VSIPL has an understanding of owning the data it operates on. Now VSIPL blocks are always owned by VSIPL and are said to be in the *admitted* state. It is not possible to remove VSIPL blocks from the admitted state. It is possible to create a block and bind it to memory allocated by the application external to VSIPL. This type memory, and block, are termed *user data* arrays, and *user* blocks. A user block is created using the **blockbind** function. It is created in a *released* state. It is an error to use any VSIPL function which will read or write the data array of a released block.

When a user block is to be used by VSIPL It must first be admitted to VSIPL using the **blockadmit** function. When the application needs to access the data of an admitted user block directly the block must first be released using the **blockrelease** function.

Note the following. A user block and a VSIPL block of the same type and precision have the same type definition. All of the information as to whether a block is a *user* block, a *VSIPL* block, or is *released* or *admitted* is maintained by the block object as state information. A VSIPL block is always admitted, and may not be released. A user block is created as released and may be admitted or released as required.

Rebinding user data to a user block

In a situation where input and output (I/O) is continuous, sometimes called data streaming, it would be bad resource management to destroy and allocate new user blocks continuously where it is desirable to have multiple data buffers used for essentially the same thing. In addition the view setup on a user block would also need to be destroyed and reconstructed every time a new buffer is admitted to VSIPL and an old buffer is released. To get around this problem a function was defined in VSIPL that allows one to rebind a new buffer to a block. The new buffer has the same features as the old buffer. This way, for instance, while the second buffer is being filled VSIPL can be operating on data from the first buffer. When VSIPL is done the user block is released, and when the second buffer is filled it is bound to the user block using **`rebind`**, and the first buffer unbound from the block is free to accept new data from the user process while the user block is admitted to VSIPL for data manipulation.

I/O Example

In example 8 we assume we need an elementwise vector cosh function. In the VSIPL standard there is a general function for doing elementwise operations like this; however it is not included in the Core Lite profile. For this example we would like to manipulate the elements directly and then import them into VSIPL. We will do this in-place.

Now we examine Example 8 below. Note the use of the capital **`IP`** in the vector cosh function name to denote in-place is the author's notation, not VSIPL's.

Example 8

```
#include<vsip.h>
int VU_vcoshIP_f(vsip_vview_f *a)
{  vsip_vattr_f attr;
   vsip_vview_f *b = a;
   vsip_scalar_f *buff;
   vsip_length n;
   vsip_block_f *B;
   vsip_vgetattrib_f(a, &attr);
   B = attr.block; n = attr.length;
   if ((buff = vsip_blockrelease_f(B,1)) == NULL){
      buff=(vsip_scalar_f *)malloc(
           n * sizeof(vsip_scalar_f));
      if(buff != NULL){
           if((B = vsip_blockbind_f(buff,n,0)) == NULL){
              free((void*)buff);
              return 0;
           }
         }else{ return 0;
      }
      vsip_blockadmit_f(B,0);
      b = vsip_vbind_f(B,0,1,n);
      vsip_vcopy_f_f(a,b);
      vsip_blockrelease_f(B,1);
      }
   while(n-- >0){
      *buff = cosh(*buff); buff++;
   }buff = vsip_blockfind_f(B);vsip_blockadmit_f(B,1);
   if(a != b){
      vsip_vcopy_f_f(b,a);
      vsip_valldestroy_f(b);
      free((void *) buff);
   }return 1;
}
int main()
{     int VU_vcoshIP_f(vsip_vview_f *);
      void VU_vprint_f(vsip_vview_f*);
      vsip_vview_f *A = vsip_vcreate_f(8,0);
      vsip_vramp_f(0,.2,A);
      printf("A = \n");VU_vprint_f(A);
      VU_vcoshIP_f(A);
      printf("cosh(A) = \n");VU_vprint_f(A);
      vsip_valldestroy_f(A);
      return 1;
}
```

```
44  void VU_vprint_f(vsip_vview_f* a){
45        int i;
46        vsip_vattr_f attr;
47        vsip_vgetattrib_f(a,&attr);
48        for(i=0; i<attr.length; i++)
49           printf("%6.4f ",vsip_vget_f(a,i));
50        printf("\n");
51        return;
    }
```

In **line 10** we attempt to release the block attached to vector **a**. If the block releases we go to **lines 25-26** where the buffer returned from the blockrelease is used directly to calculate an in place cosh. We then readmit the block in **line 27** and return a 1 for successful in **line 32**. Note we set vector **b** equal to **a** in **line 4** so **line 28** is false and we fall through to the return statement.

If the **blockrelease** returns a null pointer at **line 10** then we know that the input vector **a** is not a user vector. In this case we create a buffer of the proper size to hold the number of **vsip_scalar_f** elements in vector **a**. We check to make sure we were successful at allocating memory for a block, and return zero if not successful. In **line 14** we create a block and bind the buffer to it. In **line 20** we admit the block to VSIPL. In **line 22** we copy the input vector **a** to the user vector **b**. Note that pointer **b** was set to **a** in **line 5**, but was reset in **line 21** to the newly created user vector **b**. We now release the block **B** and go into the loop at **line 23** to calculate the cosh vector.

Note that we had to admit the block **B** before copying the input vector to the user vector, and then we had to release the block **B** before using the buffer directly. We now reset the **buff** pointer to its original value, and then admit **B** at **line 27**. Note that when we reset the pointer with **blockfind** on **line 27** we must do it before the block is admitted. An admitted block will not return the public buffer. If the pointer **a** is not equal to the pointer **b** then it has been reset. We enter the **if** code (at **line 28**) which copies the result of the cosh to the vector **a**, where we want it, and then on **line 30**, destroy the vector **b**, the block **B**, and any memory allocated by VSIPL. Note that the buffer **buff** was not allocated by VSIPL, but by the application. The application is responsible for cleaning that up. On **line 31** we free this memory.

Note that the admission on **line 20** has a zero as the second argument. This equates to false in VSIPL and tells VSIPL that we are not interested in what the buffer contains. Note that the release in **lines 10** and **23** and the admit in **line 27** have 1 for this argument. This equates to a true in VSIPL and indicates that the buffer contains data that we are interested in maintaining during the change of state of the block.

To test our code we do a simple program starting on **line 34**. In Matlab our program (main) looks like.

```
a = [0:.2:1.4]
 a =
         0    0.2000    0.4000    0.6000    0.8000    1.0000    1.2000    1.4000
 a = cosh(a)
```

a =
1.0000 1.0201 1.0811 1.1855 1.3374 1.5431 1.8107 2.1509

The output of Example 8 is

```
A =
0.0000 0.2000 0.4000 0.6000 0.8000 1.0000 1.2000 1.4000
cosh(A) =
1.0000 1.0201 1.0811 1.1855 1.3374 1.5431 1.8107 2.1509
```

Here we elected to use a *VSIPL* vector for our input to our cosh function; however we could have used a *user* vector and avoided the creates and destroys in the function.

Complex User Data

Most user data will be an array of vsip scalars of the type of the block the user array will be bound to. There are a few exceptions in VSIPL and only one exception in VSIPL Core Lite. This exception is an array of complex.

For user data, complex is **not** an array of type vsip_cscalar_f. For a complex user data array of size N the memory allocated is of type vsip_scalar_f, and is either two arrays of equal size N or a single array of size $2 \times N$. The first case of two arrays is termed split data, and the second case of a single array is termed interleaved data. For interleaved data the elements are organized consecutively as real, imaginary, real, imaginary, etcetera. For split data one array is real, the other imaginary. Examination of **`vsip_cblockbind_f`** will show how the two cases are handled in the function call.

Note that because of the possibility of split user data, requiring two data pointers, the I/O support functions for complex **`release`**, **`admit`**, and **`rebind`**, are slightly more complicated than their real counterparts. Examination of the prototypes in chapter two should make the differences clear.

There are many intricacies to I/O in VSIPL that were not covered in this section; however the author does not want to get bogged down in an introductory section with a lot of details. There is a great deal of information available in Appendix A, VSIPL fundamentals which deals with user blocks and data in some depth.

Scalar Functions

The Core Lite profile only defines 4 scalar functions. Three of these are **`vsip_real_f`**, **`vsip_imag_f`**, and **`vsip_cmplx_f`**. There is also a function **`vsip_CMPLX_f`** which is included to allow the vendors to include a macro for creating complex numbers. The TASP VSIPL Core Lite profile has not implemented **`CMPLX`** as a macro, and just calls the **`cmplx`** function within **`CMPLX`**.

The functions **`real`**, **`imag`** and **`cmplx`** are important for manipulating complex scalars. For instance **`vsip_cvget_f`** returns a scalar of type **`vsip_cscalar_f.`** To extract the real an imaginary portion of this scalar you would use **`vsip_real_f`** or **`vsip_imag_f`**. To make a complex number you would use **`vsip_cmplx_f`**. For instance

```
/* put a complex number (a,b) at */
/* element number 6 (index #5) in a vector */
vsip_vput_f(complex_vector,5,vsip_cmplx_f(a,b));
```

Or to print the real and imaginary portions of a complex number;

```
vsip_cscalar_f a;
/* some code */
a = vsip_cvget_f(complex_vector,5);
print("%f + %fi ",vsip_real_f(a),vsip_imag_f(a));
```

VSIPL Elementwise Functions

Most elementwise functions are straightforward. Generally these functions take one or more input vector views, or a scalar and a vector view, and do an element by element calculation outputting the result in an output vector. A few of the elementwise functions, such as **`sumval`** and **`dot`** calculate some value based on an elementwise operation and accumulate. Finally we have **`ramp`**, **`fill`**, and random numbers. These functions generate data and fill a vector based on some formula, element by element. The functions **`ramp`** and **`fill`** have been used in previous examples and are easy to understand based on their prototype definition; however random number generation is more complicated and requires some explanation.

Except for random number generators no specific example of elementwise functions will be included; however almost all of the examples include elementwise operations.

Random Number Generation

The VSIPL random number generator is a more complicated function, actually a set of functions, than the other elementwise functions. The function set includes a create function which is used to create a random number state object, a destroy function to destroy the random state object when we are done with it, and a vector random generator function. For VSIPL Core Lite only the uniform generator for real vectors is part of the profile. The function generates a uniform random number between zero and 1.

The vector random number is simple, just filling a vector with uniform random numbers from the sequence, and updating the state object each time.

The random number state creation is a little more complicated than the generator function. To understand it first one must understand the expectation that VSIPL programs will be run in multiple processor environments, and it is desirable to be able to calculate independent random number sequences on the different processes based on a single seed value. To this end the **`randcreate`** function has an argument which indicates the total number of processes that will be calculating random sequences, and an argument that indicates which process the state object is being created for. Having the value of the total number of processes allows the create function to subset the random number sequence into the proper number, and having the number of the process allows the create to initialize the state object for the process to the correct subset.

The random state creation also allows one to chose either the required portable random number generator (portable because all implementations use the same generator, defined by

VSIPL), or a non-portable generator of the implementors choosing. There is no requirement for an implementation to support its own generator, and the non-portable generator may default to the portable version.

Below find Example 9 demonstrating the **randu** function set. In this example create two separate vectors full of independent random numbers. The random number generator is also used in Examples 10 and 11 to make some data to work with.

Example 9

```
1     /* Create two independent random sequences */
2     #include<stdio.h>
3     #include<vsip.h>
4     #define TYPE VSIP_NPRNG /* non portable generator flag*/
5     #define N 1024  /* length of random vector */
6     #define init 17 /* random initialization */
7
8     int main()
9     {
10          vsip_vview_f *ran1 = vsip_vcreate_f(N,0),
11                       *ran2 = vsip_vcreate_f(N,0);
12          vsip_randstate *state1 =
13                   vsip_randcreate(init,2,1,NPRNG);
14          vsip_randstate *state2 =
15                   vsip_randcreate(init,2,2,NPRNG);
16          vsip_vrandu_f(state1,ran1);
17          vsip_vrandu_f(state2,ran2);
18    /* do something with ran1 and ran2 */
19          vsip_randdestroy(state1);
20          vsip_randdestroy(state2);
21          vsip_valldestroy_f(ran1);
22          vsip_valldestroy_f(ran2);
23          return 1;
24    }
```

Example 9 is very simple and does not do anything interesting. In **lines 10** and **11** we create two vectors to hold our random sequences. In **lines 12** and **14** we create two random state objects. Argument two of rand create tells the randcreate function we want a state object suitable for *two* independent random number generators. Argument 3 of **line 12** says to create the state object for the *first* process. Argument 3 of **line 14** says to create the state object for the *second* process. In **lines 16** and **17** we generate the random numbers and fill the two vectors created to hold them. **Lines 19** through **22** clean up all the objects created by VSIPL.

Signal Processing Functions

The Core Lite profile supports a histogram function, a complex to complex out of place Fourier transform, a real to complex, and a complex to real Fourier transform, and a finite impulse response (FIR) filter with desampling.

The Fourier Transform

The fourier transform function set for Core Lite includes three discrete fourier transforms (DFT), three corresponding create functions to create the fourier transform object, and a destroy function to destroy the fourier transform object. There is only one data type for the fourier transform object, and so only one destroy function is required. The type of fourier transform object created (for **`ccfftop`**, **`rcfftop`** or **`crfftop`**) is kept as state information by the fourier transform object.

The **`ccfftop_create`** function will create an fft for either a forward or inverse DFT. It is assumed that **`rcfftop`** and **`crfftop`** are done in the forward and inverse directions respectively, so there is no direction arguments in **`rcfftop_create`** and **`crfftop_create`**. Within the header file **`vsip.h`** resides an enumerated type definition which may be used for the direction argument.

```
typedef enum{
   VSIP_FFT_FWD = -1,
   VSIP_FFT_INV = 1
}   vsip_fft_dir;
```

VSIPL requires an FFT (Fast Fourier Transform) algorithm for a radix of two with one radix of 3 if necessary. Any length DFT is required to be supported, but a fast transform is only required for lengths of $N = 2^n 3^m$ where m is either 0 or 1. The TASP VSIPL Core Lite FFT base algorithm actually supports $N = 2^{m_0} 4^{m_1} 8^{n_0} 3^{n_1} 5^{n_2} 7^{n_3}$ where m_i is either 0 or 1 and n_i is some integer greater than or equal to zero. In the TASP VSIPL implementation if N is not factorable as above then the last factor (which is not factorable by 2, 4, 8, 3, 5, or 7) is used as a final factor and a DFT is done for that stage.

The functions **`rcfftop`** and **`crfftop`** require that the data be even, and only half the transform is returned. So for **`rcfftop`** the input real vector is of length N and the output complex vector is of length $N/2 + 1$. The inverse is true for **`crfftop`**.

So far we have done a few meaningless examples to illustrate VSIPL. This example will be just as meaningless. Lets find the FFT of a real vector of random numbers using rcfftop, extend the FFT to full length and find its inverse using **`ccfftop`**. We will then subtract the input vector from the output vector and find the mean square value of the result, which should be close to zero.

Example 10

```
#include<stdio.h>
#include<vsip.h>
#define RNL 1024  /* length of random vector */
#define RNS 17 /* random number seed */
#define RNT VSIP_PRNG /* random number type */
int main(){
   vsip_cvview_f *fft = vsip_cvcreate_f(RNL, 0),
                 *invfft = vsip_cvcreate_f(RNL, 0);
   vsip_randstate *state = vsip_randcreate(RNS,1,1,RNT);
   vsip_vview_f   *input = vsip_vcreate_f(RNL, 0);
   vsip_fft_f *rcfft = vsip_rcfftop_create_f(RNL,1,0,0);
   vsip_fft_f *ccfftI = vsip_ccfftop_create_f(
            RNL, 1.0/RNL, VSIP_FFT_INV,0,0);
   vsip_vrandu_f(state,input);
   vsip_cvputlength_f(fft,RNL/2+1);
   vsip_rcfftop_f(rcfft, input, fft);
   vsip_cvputoffset_f(fft, 1);
   vsip_cvputlength_f(fft, RNL/2-1);
   { /* fill out the forward fft to full length */
      vsip_cvview_f *temp = vsip_cvcloneview_f(fft);
      vsip_cvattr_f at; vsip_cvgetattrib_f(temp,&at);
      at.offset = RNL - 1; at.stride = - 1;
      vsip_cvputattrib_f(temp,&at);
      vsip_cvconj_f(fft,temp); vsip_cvdestroy_f(temp);
   }
   vsip_cvputoffset_f(fft,0);
   vsip_cvputlength_f(fft,RNL);
   vsip_ccfftop_f(ccfftI,fft,invfft);
   { /* compare results */
      vsip_vview_f *real = vsip_vrealview_f(invfft);
      vsip_vview_f *result = vsip_vimagview_f(invfft);
      vsip_vsub_f(input,real,result);
      printf("%f\n",vsip_vsumsqval_f(result)/RNL);
      vsip_vdestroy_f(real); vsip_vdestroy_f(result);
   }
   vsip_fft_destroy_f(rcfft);
   vsip_fft_destroy_f(ccfftI);
   vsip_randdestroy(state);
   vsip_cvalldestroy_f(fft);
   vsip_cvalldestroy_f(invfft);
   vsip_valldestroy_f(input);
   return 1;
}
```

In Example 10 we create the data space with **`create`** functions. We modify the stride, length and offset attributes with full knowledge of the initial attributes of the views. For this reason we don't need to get the attributes first. We do need to keep track as we move through the code however.

In **line 4** we define a constant for initializing the random number state created in **line 9.** In **line 14** we use the random number generator to initialize a real input vector of random values. In **line 7** and **8** we create two complex vectors of length RNL , the first (**`fft`**) to hold the transform of the random input vector, and the second (**`invfft`**) to hold the inverse transform of **`fft`**.

We want to do the forward transform using **`vsip_rcfftop_f`**. We made **`fft`** of length RNL since we plan on filling out the vector to a full length **`fft`** for use in **`vsip_ccfftop_f`** for doing the inverse transform; however **`rcfftop`** requires a vector of length $\mathrm{RNL}/2 + 1$ so we set this length in **line 25**. We create the fft object in **line 11** for a length of RNL and a scale factor of 1 . The last two arguments of the create are not used in TASP VSIPL so we set them to zero.

In **line 16** we do the Fourier transform on **`input`** placing the result in **`fft`**. Now we need to select the redundant portion of fft in preparation for filling out the fft vector to full length. In line 17 we set the offset of **`fft`** to 1 (the second element), since the first element is the DC value of the transform and is unique. The last value of the transform is also unique since the input real vector was even, so we set the length of **`fft`** to $\mathrm{RNL}/2 - 1$. Now we enter a section of code between **lines 19** and **25** where we copy the conjugate of the redundant section in reverse order to the end of the final transform vector. In **`line 20`** we create a clone of **`fft`** which sets the length of the vector properly. This is the vector we are going to copy into, and the first element of **`fft`** (as the view is currently defined) must copy to the first element of **`temp`**. We want this to be the last element of the block, so we set the offset of **`temp`** to the end of the block at offset $\mathrm{RNL} - 1$. We want **`temp`** to travel backward through the block so we set the stride of **`temp`** to -1 . We now do the copy and conjugation in one step using **`cvconj`** in **line 24**, and then destroy the **`temp`** vector in **line 24**. In **line 26** and **27** we restore the **`fft`** view to the entire block. What we have done between **lines 17** and **27** would look in Matlab, for Matlab vector *a* of length RNL , as

>> a(end:-1:RNL/2+2) = conj(a(2:RNL/2));

In **line 12** we create an fft object for use in **`ccfftop`** to do an inverse Fourier transform of length RNL and with a scale factor of $1/\mathrm{RNL}$. Notice that **`ccfftop_create`** has an argument for forward or inverse transform, but **`rcfftop_create`** does not have the argument and always goes in the forward direction. The last two arguments are not implemented in the TASP implementation of core lite and so are set to zero. We now find the inverse of **`fft`** in **line 28**.

In **lines 29** through **35** we subtract the **`input`** from the **`real`** output and examine the mean square value of the **`result`**. This should be very close to zero as the input and output should be the same.

The Finite Impulse Response Filter

The FIR (finite impulse response) function set is designed to allow for continuous filtering with desampling. The filter object saves state information from the previous filter operation allowing for continuous filtering of vector segments of a data stream. When desampling the number of returned filtered elements may vary depending upon the state of the filter object. For this reason the FIR filter function returns an integer which is equal to the number of elements in the output vector from the filter operation. Since a filter object is created there is a filter destroy function available for destroying the filter object.

The FIR filter create requires an input of a filter kernel, filter symmetry information, the length of the input vector to be filtered, and a desampling factor. The final two arguments are included to allow the vendor to optimize his routine for various common filter operations. Neither arguments are implemented in TASP VSIPL and so a zero will normally be placed here when using TASP VSIPL. For application programmers who are developing on TASP VSIPL for other hardware it is recommended to use the proper values for that hardware. TASP VSIPL ignores these values and will work fine for any information inserted.

The *filter kernel* and the symmetry argument vary depending upon the type of filter coefficients. For TASP VSIPL including all of the filter coefficients with a VSIP_NONSYM symmetry argument will always work. If the filter coefficients are symmetric and there are an even number then using the first half of the filter coefficients as the kernel and the VSIP_SYM_EVEN_LEN_EVEN argument will work. If the filter coefficients are symmetric and there are an odd number then using the first half of the filter coefficients plus the middle point $((N-1)/2+1)$ as the kernel and the VSIP_SYM_EVEN_LEN_ODD argument will work. TASP VSIPL always expands the filter coefficients to full length and does the same filter for all three cases, so the author recommends always using the VSIP_NONSYM version kernel, unless developing for another platform. Note that the *kernel* is simply a vector of filter coefficients as described above. Also be aware that the current FIR filter in TASP VSIPL is not a fast fir, and is not optimized in any way. For some operations it may be desirable to do the FIR filter directly using other VSIPL operations.

The FIR filter function requires a FIR filter object, an input vector of length N, and an output vector which has a length equal to the input vector length divided by the desampling factor rounded up to the nearest integer. This is commonly called the ceiling of N/D where N is the input vector length and D is the desampling factor.

The FIR is demonstrated in Example 11. The filter coefficients are in **lines 21** through **29** and are input to a user data array. In **line 31** we create a block and bind the coefficients to it using **`blockbind`**, and then **`bind`** the block to a vector view at **line 32**. Since this is a *user* block in **line 34** we **`admit`** it to VSIPL so we can use any vector views binding the block in our functions.

We then create the FIR object in **line 35**. Notice that we included the entire set of filter coefficients and so we create the object as VSIP_NONSYM. The actual filter coefficients are odd symmetric so we could have created the filter object as just the first 22 (of 43) coefficients and passed a symmetry argument of VSIP_SYM_EVEN_LEN_ODD. We then destroy the kernel since we no longer need it after the filter object is created. Notice we use **`alldestroy`** so that

both the vector view and the block is destroyed. The kernel data is not destroyed because the blocks *user* state is set, and the destroy function knows to not destroy the *user* data array.

For the example we set a decimation factor D, average **avg**, and base length N in **lines 3** through **5**. In **line 6** we set a constant to initialize the random number generator.

We create an input vector of length $D \times N$ in line 12. This ensures the output from the FIR filter will be of length N. In line 41 we fill the input vector with uniform random numbers between 0 and 1. In **line 42** we do a negative DC offset of our input vector by 0.5 and in **line 43** we filter the input vector into the output vector. In **line 44** and **45** we do an FFT estimate of the spectrum. Note power normalization is not done and is not important to the example. In **line 46** we do a running sum of the spectrums. In **line 4** we define **avg**, the number of sums, and in **line 48** we normalize our sum by the **avg** number. We then print out the result using a print subroutine VU_vprint_f contained at the end of the example. Note that **line 49** allows the output to be brought into Matlab and plotted by piping standard out into a *.m* file and executing the file in Matlab. (The author knows there are better ways to get data into Matlab, but he is to lazy too figure them out.)

In Figure 1 we note the frequency response of the filter coefficients as the bottom plot of the figure. Matlab code to generate this plot, and the filter coefficients for the example, were obtained from an internet site at Rice University
(*http://jazz.rice.edu/software/RU-FILTER/cpm/*)
There is a paper describing the method for calculating the coefficients by I. W. Selesnick and C. S. Burrus,
“Exchange Algorithms that Complement the Parks-McClellan Algorithm for Linear-Phase FIR Filter Design”.
I. W. Selesnick appears to be the author of the Matlab code.

Summary

In this chapter we have quickly covered the functionality of the Core Lite profile and given examples on its use. We cover methods for creating and destroying blocks and views, and for obtaining views of the real and imaginary portions of complex views. *User* data and methods to get data in and out of VSIPL are discussed. Finally we do examples illustrating the use of the random number generator, Fourier transform, and the FIR filter.

Example 11 (Page 1 of 2)

```
#include<stdio.h>
#include<vsip.h>
#define N   1024
#define avg 1000
#define D      2
#define RNS  17       /* Random Number Seed */
#define RNT VSIP_PRNG /* Random Number Type */
void VU_vprint_f(vsip_vview_f*);
int main()
{
   int i;
   vsip_vview_f *dataIn   = vsip_vcreate_f(D * N,0);
   vsip_cvview_f *dataFFT = vsip_cvcreate_f(N/2 + 1,0);
   vsip_vview_f *dataOut  = vsip_vcreate_f(N,0);
   vsip_vview_f *spect_avg = vsip_vcreate_f(N/2 + 1.0,0);
   vsip_vview_f *spect_new = vsip_vcreate_f(N/2 + 1.0,0);
   vsip_randstate *state  = vsip_randcreate(RNS,1,1,RNT);
   vsip_fir_f *fir;
   vsip_fft_f *fft = vsip_rcfftop_create_f(N,1,0,0);
   vsip_scalar_f b[] =
            {0.0234, -0.0094, -0.0180, -0.0129,  0.0037,
             0.0110, -0.0026, -0.0195, -0.0136,  0.0122,
             0.0232, -0.0007, -0.0314, -0.0223,  0.0250,
             0.0483, -0.0002, -0.0746, -0.0619,  0.0930,
             0.3023,  0.3999,  0.3023,  0.0930, -0.0619,
            -0.0746, -0.0002,  0.0483,  0.0250, -0.0223,
            -0.0314, -0.0007,  0.0232,  0.0122, -0.0136,
            -0.0195, -0.0026,  0.0110,  0.0037, -0.0129,
            -0.0180 ,-0.0094,  0.0234};
    {
        vsip_block_f *kblock = vsip_blockbind_f(b,43,0);
        vsip_vview_f *kernel =
            vsip_vbind_f(kblock,0,1,43);
        vsip_blockadmit_f(kblock,1);
        fir = vsip_fir_create_f( kernel, VSIP_NONSYM,
                  D * N, D, VSIP_SAVE_STATE, 0, 0);
        vsip_valldestroy_f(kernel);
     }
```

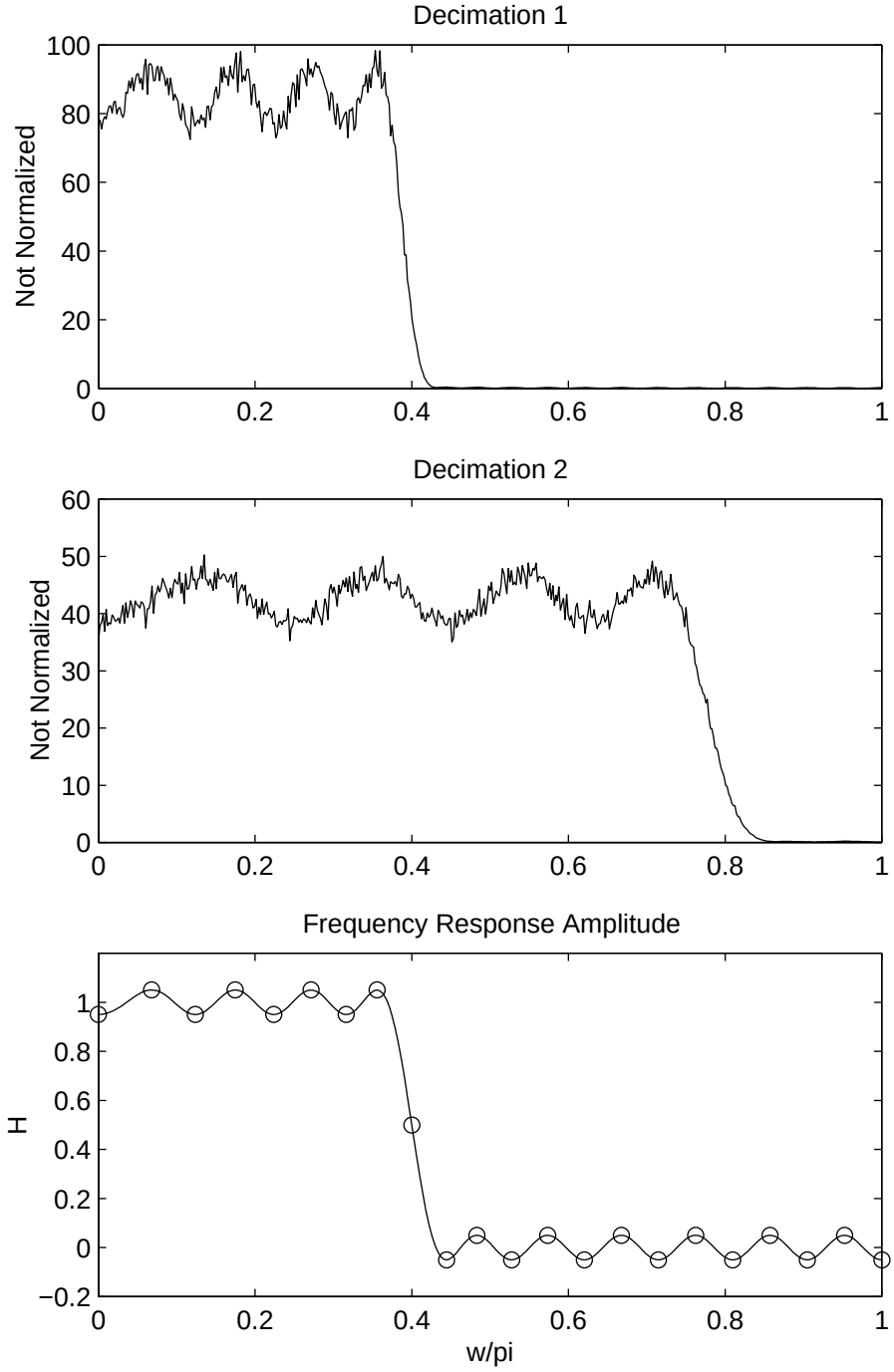
Example 11 (Page 2 of 2)

```
      vsip_vfill_f(0,spect_avg);
      for(i=0; i<avg; i++){
         vsip_vrandu_f(state,dataIn);
         vsip_svadd_f(-.5,dataIn,dataIn);
         vsip_firflt_f(fir,dataIn,dataOut);
         vsip_rcfftop_f(fft,dataOut,dataFFT);
         vsip_vcmagsq_f(dataFFT,spect_new);
         vsip_vadd_f(spect_new,spect_avg,spect_avg);
       }
       vsip_svmul_f(1.0/avg,spect_avg,spect_avg);
       printf("spect_avg =");VU_vprint_f(spect_avg);

       vsip_valldestroy_f(dataIn);
       vsip_valldestroy_f(spect_avg);
       vsip_valldestroy_f(spect_new);
       vsip_cvalldestroy_f(dataFFT);
       vsip_valldestroy_f(dataOut);
       vsip_randdestroy(state);
       vsip_fft_destroy_f(fft);
       vsip_fir_destroy_f(fir);
       return 1;
}

void VU_vprint_f(vsip_vview_f *a)
{
    vsip_vattr_f attr;
    vsip_index i;
    vsip_vgetattrib_f(a,&attr);
    printf("[");
    for(i=0; i<attr.length-1; i++)
         printf("%7.4f;\n",vsip_vget_f(a,i));
    printf("%7.4f];\n", vsip_vget_f(a,i));
    return;
}
```

Figure 1



The figures above are related to Example 11. The bottom plot is the frequency response related to the kernel in **lines 20** through **29** of the example. Matlab code available at internet site **jazz.rice.edu/software/RU-FILTER/cpm/** was used to generate the plot and the filter coefficients, and is directly from the first **Example** referenced on that web page. The plots Decimation 1 and 2 are the result of Example 11 for the stated decimation factors.

INDEX

H

I

K

O

Appendix A VSIPL Fundamentals

VSIPL Fundamentals

Introduction

This appendix contains fundamental information about a VSIPL compliant library including the basic type definitions, *block* requirements, *view* requirements, and basic VSIPL definitions and terminology.

Note that there are various requirements in the functionality document which pertain to some small subset of functions. These requirements are not covered here. This document covers the more general requirements that must be met by virtually all VSIPL implementations, no matter the *profile*.

Disclaimer

The VSIPL library is *object based*, **not** *object oriented*. The reader should be careful to not bring along to this document any object oriented terms which may have specific meaning to them from another context, but which are being used by the VSIPL forum to mean something else. The basic terminology used by VSIPL is described below.

Blocks and Views

VSIPL has a notion of data storage in a *block*. A *block* is an abstract notion of contiguous data elements available for storage of data. Associated with a block is a block object. The block object contains the information necessary for the VSIPL implementation to access and the memory needed by the block for data storage. The design of the block object is implementation dependent.

VSIPL has a notion of vectors, matrices and three tensors which are *views* of the block. The information necessary to access the block data as if it is a vector, matrix or tensor is stored in the view object.

User Data Arrays, VSIPL Data Arrays, Released and Admitted

Memory allocated by VSIPL for data storage is termed a VSIPL data array. There is no method for a user to directly access a VSIPL data array. This causes a problem when input or output of data is needed from VSIPL. To address this problem functions are available in VSIPL to associate a data array allocated by the user to a VSIPL block.

To insure that use of data stored in a user data array by the application does not conflict with use of the data by VSIPL functions the block object associated with the data array maintains state information which informs VSIPL whether or not the user block is *admitted* or *released*.

Note a *user block* is a block which is associated with a user data array. A *VSIPL block* is a block which is not associated with a user data array. These are states of a block. The block type of a user block and a VSIPL block are identical.

Functions are available to admit or release a user block. When a block is *admitted* it is an error to directly manipulate any data in the user data array. When a block is *released* it is an error to use any VSIPL function which will read or write data in the block.

VSIPL Naming Convention and Functionality Requirements

While there is nothing to prevent an implementor from writing VSIPL compatible functions, only those functions that have been approved and are included in formal VSIPL documentation are a part of VSIPL. Functions outside of the standard should not use the VSIPL naming conventions to avoid confusion of porting of applications. In particular, function names outside of VSIPL should not start with “vsip_”.

All VSIPL functionality is called out explicitly in the functionality document except for precision. The need to allow wide variation in precision to support diverse hardware precluded any attempt to specify every possible data precision. Specified methods must be followed for including data precision using a precision affix at the end of the specified VSIPL name. The approved affixes are covered in the summary of VSIPL types below. Except for copies the precision affix is usually a suffix. Copies require two precision affix’s to make up the precision suffix.

Summary of VSIPL Types

All VSIPL type declarations and function names have the data type encoded into the name. The following are required 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 type supported will depend in part on the hardware for which the library was developed, and the expected use of the hardware.

Note that throughout the VSIPL documentation an affix of *_p* is used to denote a general precision of any type, and is a method to name functions or data types without having to spell out every single prefix which might be needed for that function or data type. Also used are *_i* to denote any integer, or an *_f* to denote any float. Note that the generalized affix is in a different font style than the specific affix. To produce a valid VSIPL name, or data type, use a specified name, or data type, from the functionality document, and replace the generalized affix with the selected affix from the table below.

Standard Floating Point Data Types	
Affix	**Definition**
`_f`	ANSI C single precision floating point
`_d`	ANSI C double precision floating point
`_l`	ANSI C extra precision floating point
Standard Integer Data Types	
Affix	**Definition**
`_c`	ANSI C char
`_uc`	ANSI C unsigned char
`_si`	ANSI C short integer

Standard Integer Data Types	
Affix	**Definition**
`_us`	ANSI C unsigned short integer
`_i`	ANSI C integer
`_u`	ANSI C unsigned integer
`_li`	ANSI C long integer
`_ul`	ANSI C unsigned long integer
`_ll`	Long, long integer, implementation dependent
`_ull`	Unsigned long, long integer, implementation dependent

Portable Precision Floating Point Data Types	
Affix	**Definition**
`_f6`	Floating point type with at least 6 decimal digits of accuracy. IEE 754 single precision (32 bit) has 6 decimal digits of accuracy.
`_f15`	Floating point types with at least 15 decimal digits of accuracy. IEEE 754 double precision (64 bit) has 15 decimal digits of accuracy.
`_f`*n*	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 `FLT_MANT_DIG`, `DBL_MANT_DIG`, or `LDBL_MANT_DIG` (which are defined in `float.h`).

Portable Precision Integer Data Types	
Affix	**Definition**
`_il8`	*int* of at least 8 bits
`_il16`	*int* of at least 16 bits
`_il32`	*int* of at least 32 bits
`_il64`	*int* of at least 64 bits
`_il`*n*	*int* of at least *n* bits
`_ul8`	unsigned *int* of at least 8 bits
`_ul16`	unsigned *int* of at least 16 bits
`_ul32`	unsigned *int* of at least 32 bits
`_ul64`	unsigned *int* of at least 64 bits
`_ul`*n*	unsigned *int* of at least *n* bits

Portable Precision Integer Data Types	
Affix	**Definition**
`_ie8`	*int* of exactly 8 bits
`_ie16`	*int* of exactly 16 bits
`_ie32`	*int* of exactly 32 bits
`_ie64`	*int* of exactly 64 bits
`_ie`*n*	*int* of exactly *n* bits
`_ue8`	unsigned *int* of exactly 8 bits
`_ue16`	unsigned *int* of exactly 16 bits
`_ue32`	unsigned *int* of exactly 32 bits
`_ue64`	unsigned *int* of exactly 64 bits
`_ue`*n*	unsigned *int* of exactly *n* bits
`_if8`	fastest *int* of at least 8 bits
`_if16`	fastest *int* of at least 16 bits
`_if32`	fastest *int* of at least 32 bits
`-if64`	fastest *int* of at least 64 bits
`_if`*n*	fastest *int* of at least *n* bits
`_uf8`	unsigned fastest *int* of at least 8 bits
`_uf16`	unsigned fastest *int* of at least 16 bits
`_uf32`	unsigned fastest *int* of at least 32 bits
`_uf64`	unsigned fastest *int* of at least 64 bits
`_uf`*n*	unsigned fastest *int* of at least *n* bits
Other Data Types	
Affix	**Definition**
_bl	Boolean Data Type. Logical *false* for 0, and Logical *true* 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 *matrix index* 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 *tensor index* 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*. VSIPL also has other special data types and structures, used for defining special objects, which are used in a single function or a small subset of functions. These special structures and data types are defined in the functionality document, but not here. Structures required for VSIPL block creation are defined below. Also defined below are structures for complex scalars and scalar indices.

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.

Note: For VSIPL 1.0 there are support functions for complex integers, but no other functions which use complex integers are defined.

Below find 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 definitions (if the type is supported) must be included with the library. Some of the information is implementation dependent. Implementation dependent information will be indicated with a bracket (<?...?>) around the dependent section

Complex	`typedef struct {vsip_scalar_`*p* `r, i;} vsip_cscalar_`*p*`;`

Boolean	`typedef <?char?> vsip_scalar_bl;`
	`typedef vsip_scalar_bl vsip_bool;`
	`#define VSIP_FALSE 0`
	`#define VSIP_TRUE 1`

Vector index	`typedef unsigned <?long int?> vsip_scalar_vi` `typedef vsip_scalar_vi vsip_index;`
Matrix index	`typedef struct {vsip_scalar_vi r,c;} vsip_scalar_mi;`
Tensor index	`typedef struct {vsip_scalar_vi l,r,c;} vsip_scalar_ti;`
Offset	`typedef vsip_scalar_vi vsip_offset;`
Stride	`typedef signed <?long int?> vsip_stride;`
length	`typedef vsip_scalar_vi vsip_length;`

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
`vsip_block_`*p*	For real blocks, index blocks, and boolean blocks
`vsip_cblock_`*p*	For complex blocks. Complex blocks are only supported for float and integer data.

Examples of incomplete type definitions for blocks included in vsip.h
`struct vsip_blockobject_bl; /* boolean block structure */` `typedef struct vsip_blockobject_bl` **`vsip_block_bl`**`;`
`struct vsip_blockobject_vi; /* vector index block structure */` `typedef struct vsip_blockobject_vi` **`vsip_block_vi`**`;`
`struct vsip_blockobject_d; /* double block structure */` `typedef struct vsip_blockobject_d` **`vsip_block_d`**`;`
`struct vsip_cblockobject_d; /* complex double block structure */` `typedef struct vsip_cblockobject_d` **`vsip_cblock_d`**`;`

Note that the above structures `vsip_blockobject_bl, vsip_blockobject_vi, vsip_blockobject_d,` and `vsip_cblockobject_d` all may reside in a VSIPL header file which is private to the implementation developer. The names of all these structures are implementation dependent. The only required naming convention is the block type necessary for declaring VSIPL objects in the user application. For the examples above these are in bold.

The following hint structure must be included with an implementation. It is not required that the hints be supported (in the functions where they are required), but the structure must be available for portability reasons. Additional details of the hint are available on the functionality page where it is defined.

```
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

`vsip_vview_`*p*	For real vector views, boolean vector views and index vector views.
`vsip_cvview_`*p*	For complex vector views.
`vsip_mview_`*p*	For real matrix views and boolean matrix views.
`vsip_cmview_`*p*	For complex matrix views
`vsip_tview_`*p*	For real tensor views
`vsip_ctview_`*p*	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`.

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

Note that the above structures `vsip_vviewobject_bl`, `vsip_vviewobject_vi`, `vsip_vviewobject_d`, and `vsip_cvviewobject_d` may all reside in a VSIPL header file which is private to the implementation developer. The names of all these structures are implementation dependent. The only required naming convention is the view type needed by the user to declare view objects. For the examples above these are in bold.

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. Mixed data types are not supported. The user supplies the size of the block on its creation.

The data in a block is accessed using *views*. The attributes stored in a view describe a portion of a blocks data using offset from the beginning of the block, stride through the block, and number of elements of the block described by the view (the length attribute). The block location of the first element is at zero, and the block location of the final element is at N-1, where N is the total length of the block.

There are two kinds of blocks, *user* and *VSIPL*. A user block is one which is created using a data array which is allocated directly by the application so that the application has a pointer to

the data array. A VSIPL block is one which is not associated with a user data array, and the user has no (proper) method to retrieve a pointer to the blocks data array.

A *user* block is either in a released or an admitted state to VSIPL. It is an error for a released *user* blocks data array to be accessed by any VSIPL function which will read or write elements in the data array. Access to a released *user* blocks data must be through direct manipulation of 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 blocks data array after it has been admitted to VSIPL. After admission only VSIPL functions should be used to access the data. 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.

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 is implementation dependent; however the data array appears as contiguous data elements for the purpose of assigning strides and offsets in views of the block. *VSIPL* blocks are always admitted and the data array can only be manipulated with VSIPL function calls.

Except for a *released user block*, there is no function available to make a *block* and then at some later time attach a data array to it. 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 re-bound to a data pointer which is not `NULL`. A *user block* in the *released* state can be re-bound to any valid pointer of the proper data type for the *block*. There is no function available in VSIPL 1.0 to allow rebinding an admitted block to different data array.

For *released user blocks* the *user data* must be contiguous with the following exception. For *complex user blocks* the attached data can be contiguous, or it can consist of two separate contiguous *data arrays* of equal size.

When a *released* block is admitted the implementation is free to do whatever is necessary without concern for the *released* data layout. The *user array data* layout must be restored to the same location and layout when the block 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 such as *hidden attributes* of the block object.

Derived Blocks

There are functions defined to *derive real views* from *complex views*. These functions produce a *derived block* to bind the real views to. The derived block is of type `vsip_block_p`.

The *required derived block* data space must encompass the entire real portion of the complex block if a real view is derived, or the entire imaginary portion of the underlying 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 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 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 *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 admitted. The *derived block* is *admitted* when the *complex block* it is bound to is *admitted*.

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 There is no method in VSIPL 1.0 to create a *view* that is not bound to a *block*. A *view* must have a *user attribute* which defines the *block* the *view* is bound to. The *block attribute* is **not** setable.

A *view* must have 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 is setable by the application.

A view will have one or more stride attributes. The magnitude of the stride attribute defines the distance between two consecutive elements in some view dimension. 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 is settable

A view will have a length attribute for each stride attribute. A length attribute is a positive integer describing the number of elements in the view direction, such as the number of elements in the row of a matrix. The length attribute is settable.

For vectors there is only one dimension so there is only one stride, and one length. For matrices there are two view dimensions, so there are two strides, and two lengths. For three tensors there are three strides and three lengths.

Additional information on offset, stride, and length attributes are available in the functionality document.

Complex Views and Derived Real Views.

It is required that it be 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 element in

the *real* 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 termed a *derived block* since it is *derived* from the complex block.

The method of instantiating a *derived* block is implementation dependent.

Note: Because of the implementation dependent nature of *derived blocks* the *stride* and *offset* of *derived views* are **not** determined until after the *view* is created.

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 a contiguous memory segment of type `vsip_scalar_f`.

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

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

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

For **matrix index** the user data array is a contiguous memory segment 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 index, the second is the column index. Note that the matrix index element in a user data array is not the same as `vsip_scalar_mi`. This corresponds to the *interleaved* method described below for complex.

For **tensor index** the user data array is a contiguous memory segment 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 leg index, the second is the row index, and the third is the column 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 float *user data* arrays is not of type `vsip_cscalar_p`

> **Interleaved**: The user data array is a contiguous memory segment of type `vsip_scalar_p`. The complex element is two consecutive elements of type `vsip_scalar_p`. The first element is real, the second imaginary.
>
> **Split**: The user data array consists of two contiguous memory segments of equal length, each of type `vsip_scalar_p`. The order of the arguments when the data is bound to a block determines the real and imaginary portions.

Development mode requirements

The functionality portion of VSIP 1.0 has required error checks for *development* mode. The basic requirement is that the implementation developer of a library supporting development

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