Final Report
GTRI Project A-6193

Prototype Extension of VSIPL into C++ and Object Oriented Design

By:

Daniel P. Campbell, Georgia Tech Research Institute

Submitted to:

Bill Koenig
AFRL/IFSC
BLDG 620
2201 AVIONICS CIRCLE
WRIGHT PATTERSON AFB, OH 45433

Submitted by:

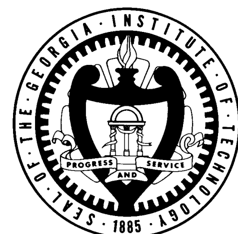
GEORGIA INSTITUTE OF TECHNOLOGY
A Unit of the University System of Georgia
Georgia Tech Research Institute
Atlanta, Georgia 30332-0800

Contracting through:

GEORGIA TECH APPLIED RESEARCH CORPORATION
Centennial Research Building
Georgia Institute of Technology
Atlanta, Georgia 30332

On USAF Contract No. F33615-98-C-1341

September 2000



1 Acknowledgements

This project was supported under On United States Air Force Contract Number F33615-98-C-1341. The Georgia Tech Research Institute would like to recognize and thank Dr. Jose Muñoz, Bob Graybill, William Koenig, DARPA, and Wright Patterson Air Force Base for their support for this effort.

2 Introduction

The Vector/Signal/Image Processing Library (VSIPL) Application Programming Interface (API) (more info is available at http://www.vsipl.org) is intended to facilitate cheaper lifetime ownership of application software, through decreased platform and hardware dependency. As a side effect of this development, a layer of abstraction was included in the API for handling diverse memory architectures and processors. This abstraction was a first step toward making an interface to low level embedded signal processing hardware which will uncouple application authors from the implementation details of a particular hardware/operating system combination.

The VSIPL Forum has borrowed very heavily from the concepts of Object Oriented Design in the construction of the API. The concepts of block and view objects, data hiding through partial headers, as well as the general methods of accessing and operating on views and blocks is consistent with Object Oriented Design. The VSIPL Forum, however, for various reasons, chose not to pursue "Object Oriented"-ness as a design goal for the API. While this allowed a more rapid growth and deployment of a usable API, the long term future of application development demands that the VSIPL Forum consider extending the API to include a specific Object Oriented structure in order to retain the gains that VSIPL stands to make over the short term.

A prototype adaptation of the Core Lite profile of the existing VSIPL API is presented which incorporates a more rigorous, though not complete application of Object Oriented Design principles, as well as a prototype implementation of that class hierarchy in C++ as a thin wrapper using an existing C library as a backend. While most Object Oriented Design principles can be applied to nearly all of the VSIPL API functionality, there are specific cases where the goals of VSIPL are contradictory to traditional Object Oriented Design principles. In these cases, the purpose of VSIPL should override a strict obedience to an abstract principle, worthwhile as that principle may be. Several instances of this are discussed and left as suggestions that the VSIPL Forum may wish to consider as evolutionary improvements on the proposed design, trading some of the original intent off for more flexibility for application authors in the future.

One criticism of an Object Oriented approach to the VSIPL API that is not valid is that the abstraction brought would cripple the performance of applications using such an API. While it is true that a properly designed abstract class hierarchy is very likely to carry the potential for some additional runtime overhead, it is almost always possible to construct an implementation that *allows* applications to chose paths which circumvent this overhead. In addition, the remaining unavoidable overhead is generally small for almost all applications – usually on the order of a table lookup for each service call. While it is the purpose of VSIPL to support embedded applications where every processor cycle is precious, as applications grow more complex, the per-low-level function call overhead will be easily outweighed at the application

level by the additional level of abstraction and flexibility, and the increased algorithmic efficiency that these can bring. Sample application code which makes use of the prototype C++ extension of VSIPL is provided, as well as code which achieves the equivalent application functionality using the C API. These simple applications are benchmarked and compared in order to illustrate the ways in which an Object Oriented version of VSIPL would and would not adversely affect performance. Since most vendors' implementation of VSIPL currently is a wrapper on a proprietary API, it is reasonable to believe that a C++ version of VSIPL for these vendors would also be a wrapper on the internal API. Therefore it is reasonable to believe that the benchmark comparisons presented represent a reasonable comparison between the performance of a vendor's potential C++ VSIPL and his internal API rather than between a C++ VSIPL and the C version.

3 Motivation

Object Oriented Design is a natural choice for the evolution of the VSIPL API. Version 1.0 of the VSIPL API borrows heavily from Object Oriented Design techniques, and does not require a major overhaul in order to apply a much more thorough Object Oriented approach. All structures specified in the API hide their internal implementation. Nearly all functions are operations on various structures that are analogous to objects (blocks, views, etc.), and all access to these objects must go through a strictly defined interface. However, applying a more thorough Object Oriented approach to the design of the API would bring several benefits.

Most importantly, Object Oriented Programming is the development paradigm of choice for a very large (and growing) number of application developers. This fact alone should carry tremendous weight with the VSIPL Forum, since the purpose of any API, ultimately, is to facilitate application development. If the VSIPL Forum can adapt the API in such a way that VSIPL becomes available for a larger body of developers, it should. There are many reasons that Object Oriented Programming is so popular, but it is important to consider the goals of Object Oriented Programming while designing an Object Oriented VSIPL.

An Object Oriented VSIPL will also facilitate more advanced applications which make use of other current APIs such as MPI. The Object Oriented approach allows a more abstract treatment of the data in question than functional paradigms, and this abstraction will be necessary if, for example, the VSIPL and MPI APIs are to merge into a single API in the future.

It is clear, also, that the High Performance Embedded Computing community regards signal and image processing problems in data centric terms, which is the hallmark of Object Oriented Design. Updating the VSIPL API to allow application writers to think more about actions performed on data, and less on the specific data or implementation details of a particular system will increase development efficiency, and portability. It will also allow the development of more general and abstract applications and systems which can grow with user demands more easily.

4 Design Goals

The purpose of this prototype design is to embrace the principles and advantages of Object Oriented Design as they apply to VSIPL. These principles are not a rigid recipe for modeling

systems, but are a collection of goals that should be striven toward. Some languages (such as Java, Eifel) require more strict application of these principles than others (such as C++), and the VSIPL Forum should consider the ramifications of the requirements of these languages as the API evolves to incorporate Object Oriented principles. The proposed design strives to use the principles of Object Oriented Design to meet and advance the goals of the VSIPL API. The proposed interface attempts to meet several goals that are specific to the problem of applying Object Oriented techniques to the VSIPL API. These goals are:

1. **Do not sabotage performance**

   Above all else, VSIPL is intended to be an API for creating high performance, embedded applications. Any change to the API for which it is impossible to create a high performance implementation is counter to that purpose and should be avoided. This does not mean that all proposed changes should guarantee a high performance path for all application code, but a high performance path should be available for any changes made. The proposed class structures and interfaces all have possible implementations that, while they may add some function overhead, do not make fast implementations impossible.

2. **Design interface and class structures that are applicable to a wide variety of languages**

   A C++ version of VSIPL is proposed, but C++ is only a stepping stone to a wider evolution of the API. Emphasis was placed on providing a C++ implementation that did not prevent a parallel interface structure in other Object Oriented languages and systems. While not every aspect of the proposed interface is directly applicable to Java, for example, nearly all of it is. The "Suggestions for the VSIPL Forum" section contains a discussion of the interface issues that departed from pure Object Oriented Design or are C++ specific.

3. **Design from the application writer's perspective**

   Object Oriented interfaces should be specified in a user-centric fashion. In all cases, method names, argument lists, and method behavior is specified to be as intuitive as possible to the user of an object. Also, methods are attached to the objects on which they operate.

4. **Make method interface and behavior as simple as possible**

   Interfaces in Object Oriented systems should be simple and intuitive. This proposal attempts to reduce the argument set for functions where possible, and reduces the number of functions which perform more than one action.

5. **Develop a class hierarchy that is extensible**

   The class and interface hierarchy specified should not tie application writers to currently available data types and precisions. This proposal attempts to specify an interface hierarchy which is abstract enough to accommodate future expansion in the API, including the inclusion of other block and view types. As an example of this goal, the

class hierarchy presented has room for as many additional precisions as the VSIPL Forum cares to specify in the future.

6. Simplify naming conventions

Version 1.0 of the VSIPL API contains families of functions that differ only by the type of data on which they operate. Placing the burden of selecting the correct function to be called on the application programmer is error prone, and is unnecessary in Object Oriented systems. The proposed interface strips the type suffixes from all operations where possible. In almost all cases, the type of object being acted on uniquely identifies the VSIPL function that would need to be called, so the proposal makes wide use of polymorphism and dynamic binding to avoid precision suffixes.

7. Increase the ability of applications to use abstract types

Current application developers must write functions based on specific precision selection. Since implementations of the API may contain different subsets of the various precisions, application portability is damaged. Also, application writers who wish to write subroutines that accept VSIPL objects as parameters must write a different subroutine for each precision of data on which the function may operate. By establishing an appropriate class hierarchy, functions may accept objects which are appropriately abstract for the operation at hand.

8. Maintain object opacity

The VSIPL API embodies the very important Object Oriented Design concept of information hiding. The hiding of implementation specific details of the operation of objects separates applications from the details of a particular VSIPL implementation, and enforces a portable interface. This proposal maintains the object opacity in version 1.0 of the VSIPL API.

9. Keep the class hierarchy as simple as possible

A common mistake in designing Object Oriented systems is to introduce excessive, unneeded layers of class and interface inheritance. This only leads to confusion on the part of application writers, and the possibility of excessive, unnecessary overhead. This proposal attempts to only include class heirs that introduce new features, or redeclare inherited features. There are some deferred classes in this proposal which violate this rule, but they are needed for the expansion of the class hierarchy to include the Core profile, as well as the rest of the VSIPL API.

5 Proposed Interface and Class Hierarchy

The standard interface proposed is that instantiated by the C++ header file listed in Appendix A. The Block and View interface hierarchy is graphically illustrated in Appendix B. Note that this is the proposed interface, and not the proposed class structure. Individual implementations should be allowed to overload existing services with those accepting more specific argument types, so long as they maintain at least services listed in the standard interface. In order to allow a fast implementation, it will likely be necessary, for example, to overload most of the view

services that accept a view as a parameter with services that accept specific types of views as a parameter. The services that accept more general parameters allow the greatest level of abstraction in application code, at the potential cost of extra overhead. By overloading more general services with versions that accept more specific parameter types, it is possible to construct an implementation that short circuits several levels of potential overhead. The prototype implementation contains many examples of this.

The majority of services specified in the class interfaces correspond to their counterparts in the C version of the VSIPL API. Nearly all prefixes and suffixes have been removed, since, in most cases, class of the object being operated on, as well as the classes of objects being passed as parameters uniquely identifies the VSIPL function to which the service corresponds. In some cases, however, the prefixes and suffixes are needed in order to identify the class of object requested as a return type (e.g. blockcreate functions). In these cases, only the prefixes or suffixes required remain.

There are several specific aspects of the interface that are worth noting. The most immediately obvious change from the C version of the VSIPL API is that there are no free functions – all functionality of the API is encapsulated as services of objects. While many classes have been introduced, most correspond very closely to the object types used in the C VSIPL API. Most of the classes that do not are deferred, abstract classes which exist as placeholders that will allow services to make use of dynamic binding in order to operate on appropriately abstract objects. The most significant new class is the "vsip" class. Objects of this class form the basis of VSIPL operations in the Object Oriented version. The vsip object is responsible for all object creation (for Core Lite, this includes blocks, views, FFT objects, FIR objects, and randstate objects), as well as object binding, and any system-wide information retrieval that may be necessary. Encapsulating the basic VSIPL creation and information services within a central object that must be created prior to being used (and can be destroyed after use) has three immediate benefits. First, it simplifies the init/finalize process seen in the C version of the API. The questions of handling overlapping, nested, and repeated pairs of init and finalize calls is completely obviated. Any implementation for which the init and finalize pairs are not trivial operators will benefit by the automatic bookkeeping associated with class objects. It also enforces the rule that all VSIPL operations must be inside an init/finalize block at compile time, rather than depending on developers to remember to do so. Second, it allows completely independent VSIPL spaces to exist, should an implementation desire to keep them separate. For example, an implementation could automatically assign each vsip object and the objects it creates to a separate processor or memory space, and allow several parallel VSIPL applications to run independently. Third, it allows a completely extensible interface to the init/finalize process. Future versions of the API can overload the basic constructor with other creators which accept parameters of arbitrary type *without* breaking existing user code.

Another very significant aspect of the proposed API is the more complex linking of object types used, as well as the specified class hierarchy. Blocks and views have very similar hierarchies, because they represent the same basic types of data. All blocks and views follow the same basic hierarchy. Because the Core Lite Profile only specifies vector view (omitting matrix and tensor views), vector views are the only view classes specified in this proposal, but the matrix and tensor views should have the same hierarchy as blocks and vector views. Because there are some basic operations that are common to all views there is a deferred `vsip_view` class which specifies the interface to these operations. `vsip_vview` inherits from `vsip_view` (as would

vsip_mview and matrix_tview in the Core Profile). The next level of inheritance splits real (vsip_vview_real) and complex (vsip_vview_complex) views, since there are many operations for which real and complex views would require completely different parameter sets. All complex view precisions are classes which inherit from vsip_vview_complex. The implementation given with this proposal implements vsip_cvview_f. Among the real views, another level of inheritance is used to split integer views (vsip_vview_int) from floating point views (vsip_vview_float) since there are operations where both view types are not interchangeable. All integer view classes (vsip_vview_i in this implementation, vsip_vview_l, vsip_vview_ll, etc. in others), inherit from the vsip_vview_int class, and all floating point view classes (vsip_vview_f in this implementation, vsip_vview_d, vsip_vview_l, etc. in others) inherit from the vsip_vview_float class. Matrix and tensor views in the Core Lite Profile should have parallel inheritance structures to specify complex, real, integer, and floating point classes. The proposed interface includes functions to create views at any level of abstraction the equivalent to or more general than the block that is providing the view.

In order to meet the goal of keeping the interface as simple as possible, some services have changed from the C API. These changes all affected the redundant implied blockfind contained in several functions. The blockrebind and blockrelease functions no longer return pointers to the previously bound data.

The FFT and FIR operations are services of vsip_fft and vsip_fir class objects, respectively. Random element generation is a service of vsip_randstate class objects.

All C VSIPL functions which return values through the parameter list do so in the C++ proposal by means of references, rather than pointers.

Note that the atomic scalar types (vsip_scalar_i, vsip_scalar_f, etc.) are specified as typedefs and not as classes. In a pure Object Oriented Environment, every variable is an instance of a class. Unfortunately, the use of memory pointers as synonyms for arrays of values is nearly incompatible with having that array filled with class objects. Additionally, VSIPL implementations depend on tight memory packing of the basic scalar types, which is not possible if those types are implemented as classes. Since C++ supports implicit typecasting of basic numeric types, it is possible to simulate a class hierarchy among the scalar types by picking appropriate types to use as aliases. The scalar types available mirror the class hierarchy of objects and views, introducing vsip_scalar, vsip_scalar_real, vsip_scalar_int, and vsip_scalar_float. The vsip_scalar_complex and vsip_cscalar_f types are specified as classes since no binding is permitted to an array of either of those types. Implementers should remember that the more abstract scalar types will be used in situations where the type of the original data is unknown, and should therefore make choices about how those types will be implemented that weighs generality against performance.

6 Considerations for the VSIPL Forum

The proposed interface is not a completely Object Oriented interface. There are several areas where considerations of similarity to the original C VSIPL API, performance, and simplicity override strict adherence to Object Oriented Techniques. There are also various areas where the

VSIPL Forum should consider whether this proposal meets the needs of VSIPL users and application developers, and adjust accordingly.

6.1 Is the view and block hierarchy too complex?

The main use of inheritance is to allow operations on objects without having to know the exact class of the object. In the case of VSIPL, the base classes exist to support basic operations on abstract types. As an example, a function could take a `vsip_vview` pointer as a specified parameter and perform various operations on the object passed to it. The calling function can pass any type of vector view (`vsip_vview_f`, `vsip_vview_i`, `vsip_cvview_f`, etc) and the called function would function correctly. If there are no (or a very limited subset) of features common to all classes which inherit from a particular class, then the base class should not exist, and the inherited classes should be independent of each other. It should be noted that in this proposal there is a fairly small number of services which are common to all views, and even to all vector views. If, for example, it is unlikely that any application writer will ever want to make use of the limited set of operations available on `vsip_vview` objects without regard to their specific type, then this level of inheritance should be removed, and the relevant services should be duplicated in the formerly derived classes.

In addition, the only operations relevant to the deferred block classes are admit, release, and view binds. If the view inheritance hierarchy is simplified to remove the base view or base vector view classes, it is unlikely that the parallel block hierarchy would ever be used, and therefore should be simplified to remove the layers of inheritance equivalent to those removed from the view hierarchy.

6.2 The problem of memory binding and scalar values

Pointers, and the way that pointers are used in C and C++ present an interesting problem for making VSIPL applicable to other Object Oriented languages and environments. The notion of a pointer to an object being a synonym for an array of objects of that type does not translate into other languages. For example, Java has very limited basic arithmetic type support, and does not support pointers at all. Also, the need to bind blocks to arrays of values makes it very difficult to have a class hierarchy for scalar values that parallels blocks and views. This leaves scalar values as typedefs which are supported with implicit recasting in C++, but which do not translate well to other languages. Distributed environments such as CORBA also do not support pointers, and the entire concept of binding anything to a memory address loses meaning in such an environment. However, the notion of a buffer of contiguous data in system memory is a tremendous asset to high performance computing.

For the sake of future growth and applicability of the API, the VSIPL Forum should consider long term ways of removing memory binding from the API. Memory binding is clearly necessary as a part of the VSIPL API currently, because it is the only method for attaching VSIPL objects to external data that is system independent. Unfortunately, the external data itself is likely to be obtained in a system dependent manner, which reduces the overall portability of VSIPL applications. A solution that would circumvent this problem is to expand the blockcreate family of functions to include a wide array of methods for attaching to external data that do not involve an implicit assumption about the way that data is organized. This is

clearly a major task that will take considerable effort on the part of the VSIPL Forum, but would be quite advantageous over the long term.

6.3 Exceptions

Exceptions have been left out of this proposal, despite the fact that exceptions are a universally used mechanism within Object Oriented environments. They allow a much cleaner error checking and handling mechanism than the traditional method of signaling errors by means of return codes. The problems with adding exceptions to VSIPL are twofold. First, they represent a major philosophy departure from the C VSIPL API. C does not support exceptions, and as such, it would be difficult to maintain continuity between the C and Object Oriented versions of VSIPL. Also, VSIPL generally does not support much error checking and handling at all. The clear intent of production modeof the VSIPL API is that very little error checking is done, and that it is acceptable for any unchecked error to be catastrophic. Second, the disparity between the error checking done in production and development modes might lead to confusion among application developers regarding the promises made by the exception mechanism. For example, a developer may create a try/catch block assuming that if no exception is caught that the operations were successful, which may be true in development mode, but not in production mode.

Despite those two issues, the VSIPL Forum should give serious consideration to including exceptions as a part of the Object Oriented VSIPL specification. Exceptions are the standard means of handling errors in Object Oriented environments, and if the rules are made explicitly clear, application writers will be more efficient using a mechanism with which they are familiar. Despite what may look like a more cumbersome mechanism, exceptions can also be a considerably more efficient mechanism for handling errors than the traditional return value.

6.4 Pointers and References

The standard method used to refer to VSIPL objects in the C VSIPL API is pointers. While C++ supports pointers, many other Object Oriented languages and environments do not. The more common method for referring to objects is via the reference mechanism. Within C++ a reference is essentially a synonym for an object that is used just as the original object is. Unlike a pointer, however, a reference can not be changed to point at a different object, or be attached to a non-object (such as a NULL pointer in C), or be used after the object it refers to has been deleted. References also translate more directly into other Object Oriented environments, such as Java and CORBA. The use of pointers instead of references also makes operator overloading in C++ inoperable, since it requires the use of references.

There are two main reasons that this proposal uses pointers instead of references. First, references (like exceptions) represent a large shift in philosophy form the C VSIPL API, and may confuse application writers that use both the C and the C++ API. More importantly, however, using references instead of pointers would require that the view and block creation services return references. This creates the implicit expectation that the object which provides the creation service (the vsip class object in this case) retains responsibility for the created object, most importantly in terms of memory management and guaranteeing the proper destruction of the object. Returning object pointers leaves responsibility for ownership of the object in the

hands of the object whose service calls the creation service. Shifting the responsibility for created objects onto the vsip class object is very much against the philosophy of the current VSIPL API. The current specification is clearly designed to leave as little responsibility with the library, and as much with the application as is possible.

While that philosophy leads to a simple, lean, and efficient library implementation in most cases, it is fraught with the danger. The most common errors in C programs (and often the most difficult to debug) are pointer errors. There are very few uses for a pointer to an object that can not be duplicated with a reference to that object. Also, moving bookkeeping responsibility to the vsip object will not measurably impact run-time performance, since the only additional load it imposes is at object creation and destruction, which will not be an inner-loop function in any real time application. This change would, however, impose an additional load on the implementation developer, and require some additional memory to handle bookkeeping at the VSIPL implementation level. The VSIPL Forum must weigh these costs against the benefits of using references in place of pointers.

7 Comparison of C and C++ VSIPL

The traditional view of comparing functional to Object Oriented programming is that Object Oriented programming decreases the overall cost of development by improving readability, maintainability, and at times bug-free development time at the cost of reduced run-time performance. At a general level, this is a reasonable comparison, however, Object Oriented languages do not always add crippling levels of performance overhead, and in most cases it is possible to implement Object Oriented programs that are as efficient as their functional equivalents. Another trade-off of note is that of executable size. Due to the abstract nature of Object Oriented programming, it is more difficult to determine at compile time which portions of a library may be used by a program than under functional languages. Compiled and linked executables are larger under many C++ environments than the equivalent C functionality.

Sample code is provided with this proposal that illustrate some of the tradeoffs between the C and C++ versions of VSIPL. For each of several different basic operations (add, multiply, subtract, real-to-complex and complex-to-complex FFT), an implementation is provided using the C VSIPL API, as well as one using the proposed C++ VSIPL API. Each of these programs measures CPU time required to repeat the operation a number of times using various sizes of vectors. In addition, two programs are included which attempt to illustrate the flexibility afforded by Object Oriented programming. The included programs are:

add: Adds two vectors of a variety of sizes, and reports CPU time required

cmul: Performs an element wise multiply on two complex vectors of a variety of sizes and reports CPU time required.

mul: Performs an element wise multiply on two real vectors of a variety of sizes and reports CPU time required.

div: Performs an element wise divide on two real vectors of a variety of sizes and reports CPU time required

fft: Performs a real to complex FFT on vectors of a variety of sizes and reports the CPU time required

fft2: Performs a forward complex to complex FFT on vectors of a variety of sizes and reports the CPU time required.

abstract: Provides a function which performs a simple compound function on a set of vectors passed to it. The operation corresponds to $C = 2A^2+B$. The C version can only accept float vectors, while the C++ version of the function can accept vectors of any type. Both programs perform the operation on vectors of a variety of sizes and report the CPU usage. The C version only uses _f vectors, while the C++ version uses several combinations of _f and _i vectors.

sigproc_model: Demonstrates the use of VSIPL to perform canonical signal conditioning and processing. There are two buffers for in-phase and quadrature (I & Q) input signal components. These buffers are filled with simulated 12-bit A/D samples in integer format. The following processing steps occur after these buffers are filled:

1. Convert input data from fixed-point to floating-point.
2. Window the data with a Blackman window.
3. Estimate the spectrum via the FFT of the windowed input data.
4. Compute the magnitude of the spectral estimate.
5. Find the max bin in the magnitude spectrum and its index.

These processing steps require user arrays to function as buffers, which are then copied into a complex vector, performing the fix-to-float conversion on the fly (step 1). VSIPL Core-Lite functions are invoked to accomplish the remaining processing (steps 2-5).

All sample code can be found in the sample-code/directory in the prototype package available on the VSIPL web site under the "Docs & Info" section (http://www.vsipl.org/documents). The C/subdirectory contains the programs written to use the C VSIPL API, and the C++/directory contains the programs written to use the proposed C++ VSIPL API. The software was compiled and executed on a Sun Sparc Ultra-60 running SunOS 5.7. The measured "user" CPU time required to perform each of the programs is listed in Appendix D. The bulk of the source code of some of the test cases is listed in appendix C. Utility functions that are not central to the functionality are omitted.

7.1 Performance

There are several differences between the C and C++ sample programs that are worth note. Of most interest to members of the VSIPL Forum, and perhaps least flattering to the Object Oriented approach is the difference in execution speed between the two versions. In nearly all cases, the C++ version of the program executed more slowly than the C version. In the most simple cases, there is a slight execution time overhead associated with the C++ version. For the 1024 vector-size, most operations suffered approximately a 2% time increase under the C++ version. However, it must be noted that since the prototype implementation provided is actually a wrapper which uses the TASP VSIPL library as a backend, some small performance loss is necessary. Since most embedded vendors that have implemented VSIPL have done so by writing a wrapper library around their own native API, it is natural to believe that if they were

to support a C++ version of VSIPL they would do so by writing a wrapper directly around the native API rather than around their C version of VSIPL. Therefore, it is likely that, for the simple operations illustrated by the add, mul, cmul, div, fft, and fft2 test cases, the performance overhead of the C++ version of VSIPL should be compared against the vendor's native API rather than the C version of VSIPL.

The *abstract1*, and *sigproc_model* test cases present situations where the Object Oriented nature of the proposed C++ version of VSIPL is put to use. In the *abstract1* test case, vectors of various data types are passed to a function which accepts real vectors of any type as inputs. This illustrates a case where the genericity of Object Oriented programming allows for a more streamlined and cheaper development cycle, potentially at the cost of large run-time performance penalties. In this case, one of the operations performed by the function (`vsip_vsquare`) is not a part of the Core-Lite profile for the *_i* vector types, and thus is not present in the library that this implementation uses as a backend. The function is implemented in the sample library by copying the vectors provided to *_f* vectors, performing the required operation, and copying result vectors back into the result vectors as needed. This is an exaggeratedly worst-case scenario, but it does illustrate one of the largest costs to an Object Oriented approach to VSIPL: abstraction comes at the price of either a larger development commitment by implementers, or the potential for slow paths to exist in a given implementation. Implementers can either implement every function required by the abstractions, for all data types included in their implementations, or can live with the possibility that some combinations of data types and functions may be too slow for real time use.

The *sigproc_model* test case illustrates a situation which is much more likely to be relevant in actual applications. In this test case, a real-world embedded DSP task is modeled, and since there is a copy operation internal to the main function, the abstraction of C++ allows the correct copy function to be called for the data type passed in at minimal runtime cost. In this case there was only a 2-3% performance cost, regardless of input data type, which is well in line with the simple operations above.

7.2 Simplicity

Programs written to use the proposed C++ version of the VSIPL API are considerably easier to read and write than their C counterparts. The need to prefix all function names with `vsip_` in order to avoid namespace collisions is gone, as is the need to decorate most function names with type specifiers. This leaves function names which are meaningful and simple representations of their intended actions, which leaves application writers with less work creating programs, and an easier time maintaining them.

7.3 Abstraction

The proposed C++ version of the VSIPL API allows a new level of flexibility in application development. Functions can be written that accept a variety of vector types without significant performance loss. This will allow greater portability (since applications will not be tied to whichever *_f* type is implemented on a given platform, for example), more reliable code, and cheaper overall software ownership. The ability to write software which is not tied to particular

data types allows a reduction of development time, and a concentration on algorithm rather than implementation which will improve overall software quality.

8 Object Oriented Design and the future of VSIPL

The proposed interface and C++ implementation of that interface are not intended to be a final declaration of what the Object Oriented and C++ versions of VSIPL should be. There is considerable room for discussion, improvements, and changes to what is presented. The design is primarily crafted from an application writer's perspective and there may be many pragmatic issues with the interface for implementers on embedded platforms. However, this proposal is a core class structure and interface hierarchy that the VSIPL Forum should strongly consider and use as a starting point for future discussions about the future of VSIPL and how it can be used in Object Oriented environments. It is shown that it is possible to implement a C++ version of VSIPL without excessive overhead or crippling performance drains. Other C++ versions of VSIPL are already in production which should also be considered seriously by the VSIPL Forum.

Appendix A
Public Class Interface

```
typedef int     vsip_scalar_bl;

typedef double  vsip_scalar_real;
typedef double  vsip_scalar_float;
typedef int     vsip_scalar_int;

typedef float   vsip_scalar_f;
typedef double  vsip_scalar_d;
typedef int     vsip_scalar_i;

typedef unsigned long int vsip_scalar_vi;   /* vector index */
typedef struct { vsip_scalar_vi r,c; } vsip_scalar_mi;  /* matrix index */
typedef vsip_scalar_vi vsip_index;
#define VSIP_MAX_SCALAR_VI ULONG_MAX

/* Type for offset, stride and length */
typedef vsip_scalar_vi  vsip_offset;
typedef vsip_scalar_vi  vsip_length;
typedef signed long int vsip_stride;

/*    real scalar type definitions          */
typedef   float         vsip_scalar_f;
typedef   signed  int   vsip_scalar_i;
typedef   unsigned int  vsip_scalar_ue32;


typedef struct
{
       vsip_offset  offset;
       vsip_stride  stride;
       vsip_length  length;
} vsip_vattr;

typedef enum
{
    VSIP_MEM_NONE          = 0,            /* No Hint           */
    VSIP_MEM_RDONLY        = 1,            /* Read Only         */
    VSIP_MEM_CONST         = 2,            /* Constant          */
    VSIP_MEM_SHARED        = 3,            /* Shared            */
    VSIP_MEM_SHARED_RDONLY = 4,            /* Shared, Read Only */
    VSIP_MEM_SHARED_CONST  = 5             /* Shared, Constant  */
} vsip_memory_hint;

typedef enum
{
  VSIP_FFT_CC_OP    = 1,
  VSIP_FFT_CR_OP    = 2,
  VSIP_FFT_RC_OP    = 3,
  VSIP_FFT_CC_IP    = 4,
  VSIP_FFT_CR_IP    = 5,
  VSIP_FFT_RC_IP    = 6
} vsip_fft_type;

typedef enum
{
    VSIP_CMPLX_INTERLEAVED,
    VSIP_CMPLX_SPLIT,
    VSIP_CMPLX_NONE
} vsip_cmplx_mem;

#define VSIP_CMPLX_MEM VSIP_CMPLX_INTERLEAVED

/* Signal flags */
typedef enum
{
    VSIP_FFT_FWD = -1,
    VSIP_FFT_INV = +1
} vsip_fft_dir;

typedef enum
{
    VSIP_FFT_IP = 0,
    VSIP_FFT_OP = 1
} vsip_fft_place;

typedef enum
{
    VSIP_ALG_SPACE = 0,
    VSIP_ALG_TIME  = 1,
    VSIP_ALG_NOISE = 2
} vsip_alg_hint;

/* random number generator flags */
typedef enum
{
    VSIP_PRNG = 0,
    VSIP_NPRNG = 1
} vsip_rng;

typedef enum {
  VSIP_NONSYM = 0,
  VSIP_SYM_EVEN_LEN_ODD   = 1,
  VSIP_SYM_EVEN_LEN_EVEN  = 2
} vsip_symmetry;

typedef enum
{
  VSIP_STATE_NO_SAVE = 1,
  VSIP_STATE_SAVE = 2
} vsip_obj_state;
```

```
typedef enum
{
  VSIP_HIST_RESET = 1,
  VSIP_HIST_ACCUM = 2
} vsip_hist_opt;



class vsip
{
public:
  vsip();
  ~vsip();
  vsip_block_f *     blockcreate_f (vsip_length length, vsip_memory_hint hint);
  vsip_block_f *     blockcreate_f (vsip_length length);
  vsip_cblock_f *    cblockcreate_f (vsip_length length, vsip_memory_hint hint);
  vsip_cblock_f *    cblockcreate_f (vsip_length length);
  vsip_block_i *     blockcreate_i (vsip_length length, vsip_memory_hint hint);
  vsip_block_i *     blockcreate_i (vsip_length length);

  vsip_block_f *     blockbind_f (vsip_scalar_f *data, vsip_length length, vsip_memory_hint hint);
  vsip_block_f *     blockbind_f (vsip_scalar_f *data, vsip_length length);
  vsip_block_i *     blockbind_i (vsip_scalar_i *data, vsip_length length, vsip_memory_hint hint);
  vsip_block_i *     blockbind_i (vsip_scalar_i *data, vsip_length length);
  vsip_cblock_f *    cblockbind_f (vsip_scalar_f *data1, vsip_scalar_f *data2, vsip_length length, vsip_memory_hint hint);
  vsip_cblock_f *    cblockbind_f (vsip_scalar_f *data1, vsip_scalar_f *data2, vsip_length length);

  vsip_vview_f *     vcreate_f (vsip_length length, vsip_memory_hint hint);
  vsip_vview_f *     vcreate_f (vsip_length length);

  vsip_vview_i *     vcreate_i (vsip_length length, vsip_memory_hint hint);
  vsip_vview_i *     vcreate_i (vsip_length length);

  vsip_cvview_f *    cvcreate_f (vsip_length length, vsip_memory_hint hint);
  vsip_cvview_f *    cvcreate_f (vsip_length length);

  vsip_cmplx_mem     cstorage (void);

  vsip_randstate *   randcreate (vsip_index seed, vsip_index numprocs, vsip_index id, vsip_rng portable);

  vsip_fft_f *       ccfftop_f (vsip_length N, vsip_scalar_float scale, vsip_fft_dir dir, vsip_length ntimes, vsip_alg_hint hint);
  vsip_fft_f *       rcfftop_f (vsip_length N, vsip_scalar_float scale, vsip_fft_dir dir, vsip_length ntimes, vsip_alg_hint hint);
  vsip_fft_f *       crfftop_f (vsip_length N, vsip_scalar_float scale, vsip_fft_dir dir, vsip_length ntimes, vsip_alg_hint hint);

  vsip_cfir_f *      cfir_create_f (vsip_vview_complex *kernel, vsip_symmetry symm, vsip_length N, vsip_length D, vsip_obj_state state,
                     vsip_length ntimes, vsip_alg_hint hint);
  vsip_fir_f *       fir_create_f (vsip_vview_float *kernel, vsip_symmetry symm, vsip_length N, vsip_length D, vsip_obj_state state,
                     vsip_length ntimes, vsip_alg_hint hint);

  void               blockdestroy(vsip_block *block);
  void               vdestroy(vsip_vview *view);
  void               valldestroy (vsip_vview *view);
  void               fftdestroy (vsip_fft *fft);
  void               firdestroy (vsip_fir_complex *fir);
  void               firdestroy (vsip_fir_float *fir);
};

class vsip_block
{
public:
  virtual int        admit(vsip_scalar_bl update) = 0;
  virtual void       release (vsip_scalar_bl update) = 0;
  virtual vsip_vview *vbind (vsip_offset offset, vsip_stride stride, vsip_length length) = 0;
  inline int         is_valid(void) {return is_valid_;};
};

class vsip_block_real : public vsip_block
{
public:
  virtual vsip_vview_real *vbind_real (vsip_offset offset, vsip_stride stride, vsip_length length) = 0;
};

class vsip_block_complex : public vsip_block
{
public:
  virtual vsip_vview_complex *vbind_complex (vsip_offset offset, vsip_stride stride, vsip_length length) = 0;
};

class vsip_block_float : public vsip_block_real
{
public:
  virtual vsip_vview_float *  vbind_float (vsip_offset offset, vsip_stride stride, vsip_length length) = 0;
};

class vsip_block_int : public vsip_block_real
{
public:
  virtual vsip_vview_int *  vbind_int (vsip_offset offset, vsip_stride stride, vsip_length length) = 0;
};

class vsip_block_i : public vsip_block_int
{
public:
  void               find (vsip_scalar_i *&data1);
  void               release (vsip_scalar_bl update);
  void               rebind (vsip_scalar_i *data1);
  vsip_vview *       vbind (vsip_offset offset, vsip_stride stride, vsip_length length);
  vsip_vview_real *  vbind_real (vsip_offset offset, vsip_stride stride, vsip_length length);
  vsip_vview_int *   vbind_int (vsip_offset offset, vsip_stride stride, vsip_length length);
  vsip_vview_i *     vbind_i (vsip_offset offset, vsip_stride stride, vsip_length length);
  int                admit (vsip_scalar_bl update);

  ~vsip_block_i();
};

class vsip_block_f : public vsip_block_float
{
public:
```

```
  void              find (vsip_scalar_f *&data1);
  void              release (vsip_scalar_bl update);
  void              rebind (vsip_scalar_f *data1);
  vsip_vview *      vbind (vsip_offset offset, vsip_stride stride, vsip_length length);
  vsip_vview_real * vbind_real (vsip_offset offset, vsip_stride stride, vsip_length length);
  vsip_vview_float *vbind_float (vsip_offset offset, vsip_stride stride, vsip_length length);
  vsip_vview_f *    vbind_f (vsip_offset offset, vsip_stride stride, vsip_length length);
  int               admit (vsip_scalar_bl update);

  ~vsip_block_f();
};

class vsip_cblock_f : public vsip_block_complex
{
public:
  void                find (vsip_scalar_f *&data1, vsip_scalar_f *&data2);
  void                release (vsip_scalar_bl update);
  vsip_vview *        vbind (vsip_offset offset, vsip_stride stride, vsip_length length);
  vsip_vview_complex *vbind_complex (vsip_offset offset, vsip_stride stride, vsip_length length);
  vsip_cvview_f *     cvbind_f (vsip_offset offset, vsip_stride stride, vsip_length length);
  int                 admit (vsip_scalar_bl update);
  void                rebind (vsip_scalar_f *data1, vsip_scalar_f *data2);

  ~vsip_cblock_f();
};


class vsip_view
{
public:
  virtual vsip_view * cloneview(void) = 0;
  virtual vsip_block *getblock() = 0;
  int   is_valid(void) {return is_valid_;};
  virtual void print() = 0;
};

class vsip_vview : public vsip_view
{
public:
  virtual void          getattrib (vsip_vattr &attrib) = 0;
  virtual void          putattrib (vsip_vattr &attrib) = 0;
  virtual void          putstride (vsip_stride stride) = 0;
  virtual void          putlength (vsip_length length) = 0;
  virtual void          putoffset (vsip_offset offset) = 0;
  virtual vsip_vview *  vcloneview (void) = 0;
  virtual vsip_vview *  vsubview(vsip_index index, vsip_length length) = 0;
  virtual void          mag (vsip_vview_float *result) = 0;
  virtual void          fill (vsip_scalar_real value) = 0;
  virtual void          ramp (vsip_scalar_real alpha, vsip_scalar_real beta) = 0;
};

class vsip_vview_complex : public vsip_vview
{
public:
  virtual vsip_vview_complex *  vcloneview_complex(void) = 0;
  virtual vsip_vview_complex *  vsubview_complex(vsip_index index, vsip_length length) = 0;
  virtual vsip_vview_real *     imagview (void) = 0;
  virtual vsip_vview_real *     realview (void) = 0;
  virtual vsip_block_complex *  getblock_complex(void) = 0;
  virtual void                  put (vsip_index i, vsip_scalar_complex *j) = 0;
  virtual void                  get_complex (vsip_index i, vsip_scalar_complex &r) = 0;
  virtual void                  copy (vsip_vview_complex *source) = 0;
  virtual void                  magsq (vsip_vview_float *result) = 0;
  virtual void                  neg (vsip_vview_complex *result) = 0;
  virtual void                  recip (vsip_vview_complex *result) = 0;

  virtual void          add (vsip_vview_complex *operand, vsip_vview_complex *result) = 0;
  virtual void          add (vsip_scalar_complex *operand, vsip_vview_complex *result) = 0;
  virtual void          add (vsip_vview_real *operand, vsip_vview_complex *result) = 0;
  virtual void          add (vsip_scalar_real operand, vsip_vview_complex *result) = 0;

  virtual void          sub (vsip_vview_complex *operand, vsip_vview_complex *result) = 0;
  virtual void          sub (vsip_scalar_complex *operand, vsip_vview_complex *result) = 0;
  virtual void          sub (vsip_vview_real *operand, vsip_vview_complex *result) = 0;
  virtual void          sub (vsip_scalar_real operand, vsip_vview_complex *result) = 0;

  virtual void          mul (vsip_vview_real *operand, vsip_vview_complex *result) = 0;
  virtual void          mul (vsip_scalar_real operand, vsip_vview_complex *result) = 0;
  virtual void          mul (vsip_vview_complex *operand, vsip_vview_complex *result) = 0;
  virtual void          mul (vsip_scalar_complex *operand, vsip_vview_complex *result) = 0;

  virtual void          div (vsip_vview_real *operand, vsip_vview_complex *result) = 0;
  virtual void          div (vsip_scalar_real operand, vsip_vview_complex *result) = 0;
  virtual void          div (vsip_vview_complex *operand, vsip_vview_complex *result) = 0;
  virtual void          div (vsip_scalar_complex *operand, vsip_vview_complex *result) = 0;

  virtual void          jdot (vsip_vview_complex *operand, vsip_scalar_complex &ret) = 0;
  virtual void          dot (vsip_vview_complex *operand, vsip_scalar_complex &ret) = 0;
  virtual void          jmul (vsip_vview_complex *operand, vsip_vview_complex *result) = 0;
  virtual void          fill (vsip_scalar_complex *value) = 0;
  virtual void          ramp (vsip_scalar_complex *alpha, vsip_scalar_complex *beta) = 0;

  virtual void          real (vsip_vview_real *result) = 0;
  virtual void          imag (vsip_vview_real *result) = 0;

  virtual void          cmplx (vsip_vview_real *real_part, vsip_vview_real *imag_part) = 0;
  virtual void          conj  (vsip_vview_complex *result) = 0;
};

class vsip_vview_real : public vsip_vview
{
public:
  virtual vsip_vview_real * vcloneview_real(void) = 0;
  virtual vsip_vview_real * vsubview_real(vsip_index index, vsip_length length) = 0;
  virtual vsip_block_real * getblock_real(void) = 0;

  virtual void              put (vsip_index i, vsip_scalar_real j) = 0;
  virtual vsip_scalar_real  get_real (vsip_index i) = 0;
```

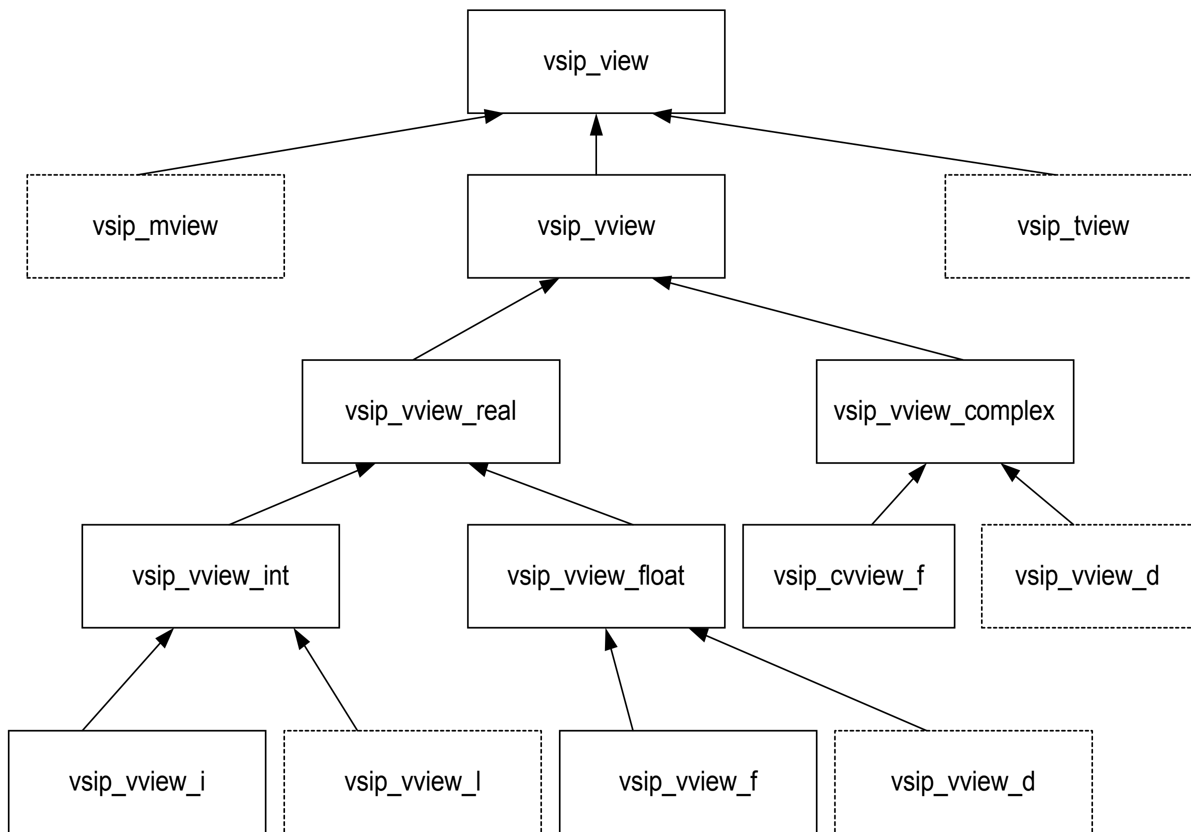
```
  virtual void              copy (vsip_vview_real *source) = 0;
  virtual void              square (vsip_vview_real *result) = 0;

  virtual void        add (vsip_vview_real *operand, vsip_vview_real *result) = 0;
  virtual void        add (vsip_scalar_real operand, vsip_vview_real *result) = 0;
  virtual void        add (vsip_vview_complex *operand, vsip_vview_complex *result) = 0;
  virtual void        add (vsip_scalar_complex *operand, vsip_vview_complex *result) = 0;

  virtual void        sub (vsip_vview_real *operand, vsip_vview_real *result) = 0;
  virtual void        sub (vsip_scalar_real operand, vsip_vview_real *result) = 0;
  virtual void        sub (vsip_vview_complex *operand, vsip_vview_complex *result) = 0;
  virtual void        sub (vsip_scalar_complex *operand, vsip_vview_complex *result) = 0;

  virtual void        mul (vsip_vview_real *operand, vsip_vview_real *result) = 0;
  virtual void        mul (vsip_scalar_real operand, vsip_vview_real *result) = 0;
  virtual void        mul (vsip_vview_complex *operand, vsip_vview_complex *result) = 0;
  virtual void        mul (vsip_scalar_complex *operand, vsip_vview_complex *result) = 0;

  virtual void        div (vsip_vview_real *operand, vsip_vview_real *result) = 0;
  virtual void        div (vsip_scalar_real operand, vsip_vview_real *result) = 0;
  virtual void        div (vsip_vview_complex *operand, vsip_vview_complex *result) = 0;
  virtual void        div (vsip_scalar_complex *operand, vsip_vview_complex *result) = 0;

  virtual void              max (vsip_vview_real *compare, vsip_vview_real *result) = 0;
  virtual vsip_scalar_real  maxval (vsip_index &index) = 0;
  virtual void              min (vsip_vview_real *compare, vsip_vview_real *result) = 0;
  virtual vsip_scalar_real  minval (vsip_index &index) = 0;

  virtual void        histo (vsip_scalar_real min_bin, vsip_scalar_real max_bin, vsip_hist_opt opt, vsip_vview_real *result) = 0;
  virtual void        neg (vsip_vview_real *result) = 0;
};

class vsip_vview_int : public vsip_vview_real
{
public:
  virtual vsip_vview_int *  vcloneview_int(void) = 0;
  virtual vsip_vview_int *  vsubview_int(vsip_index index, vsip_length length) = 0;
  virtual vsip_block_int *  getblock_int (void) = 0;
  virtual vsip_scalar_int   get_int (vsip_index i) = 0;
};

class vsip_vview_float : public vsip_vview_real
{
public:
  virtual vsip_vview_float *  vcloneview_float(void) = 0;
  virtual vsip_vview_float *  vsubview_float(vsip_index index, vsip_length length) = 0;
  virtual vsip_block_float *  getblock_float (void) = 0;
  virtual vsip_scalar_float   get_float (vsip_index index) = 0;

  virtual void        atan (vsip_vview_float *result) = 0;
  virtual void        atan2 (vsip_vview_float *denominators, vsip_vview_float *result) = 0;
  virtual void        cos(vsip_vview_float *result) = 0;
  virtual void        exp (vsip_vview_float *result) = 0;
  virtual void        log (vsip_vview_float *result) = 0;
  virtual void        log10 (vsip_vview_float *result) = 0;
  virtual void        sin (vsip_vview_float *result) = 0;
  virtual void        sqrt (vsip_vview_float *result) = 0;
  virtual void        recip (vsip_vview_float *result) = 0;
  virtual vsip_scalar_float   dot (vsip_vview_float *operand) = 0;
};

class vsip_cvview_f : public vsip_vview_complex
{
public:
  vsip_view *         cloneview(void);
  vsip_vview *        vcloneview(void);
  vsip_vview_complex *vcloneview_complex(void);
  vsip_cvview_f *     cvcloneview_f (void);

  vsip_vview *        vsubview(vsip_index index, vsip_length length);
  vsip_vview_complex *vsubview_complex(vsip_index index, vsip_length length);
  vsip_cvview_f *     cvsubview_f(vsip_index index, vsip_length length);

  vsip_vview_real *   imagview (void);
  vsip_vview_f *      imagview_f (void);

  vsip_vview_real *   realview (void);
  vsip_vview_f *      realview_f (void);

  vsip_block *        getblock();
  vsip_block_complex *getblock_complex();
  vsip_cblock_f *     cgetblock_f();

  void        getattrib (vsip_vattr &attrib);
  void        putattrib (vsip_vattr &attrib);
  void        putstride (vsip_stride stride);
  void        putlength (vsip_length length);
  void        putoffset (vsip_offset offset);

  void        put (vsip_index i, vsip_scalar_complex *j);
  void        get_complex (vsip_index i, vsip_scalar_complex &ret);

  void        copy (vsip_vview_complex *source);
  void        mag (vsip_vview_float *result);
  void        magsq (vsip_vview_float *result);
  void        neg (vsip_vview_complex *result);
  void        recip (vsip_vview_complex *result);

  void        add (vsip_vview_complex *operand, vsip_vview_complex *result);
  void        add (vsip_scalar_complex *operand, vsip_vview_complex *result);
  void        add (vsip_vview_real *operand, vsip_vview_complex *result);
  void        add (vsip_scalar_real operand, vsip_vview_complex *result);

  void        sub (vsip_vview_complex *operand, vsip_vview_complex *result);
  void        sub (vsip_scalar_complex *operand, vsip_vview_complex *result);
  void        sub (vsip_vview_real *operand, vsip_vview_complex *result);
  void        sub (vsip_scalar_real operand, vsip_vview_complex *result);
```

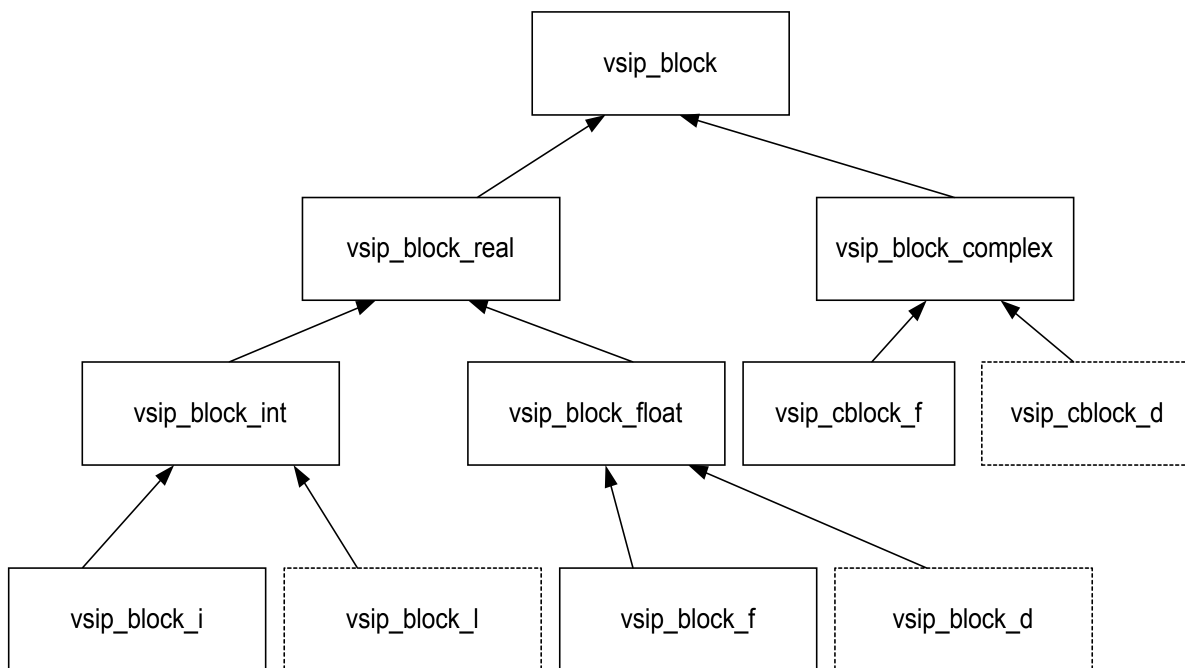
```
  void        mul (vsip_vview_real *operand, vsip_vview_complex *result);
  void        mul (vsip_scalar_real operand, vsip_vview_complex *result);
  void        mul (vsip_vview_complex *operand, vsip_vview_complex *result);
  void        mul (vsip_scalar_complex *operand, vsip_vview_complex *result);

  void        div (vsip_vview_real *operand, vsip_vview_complex *result);
  void        div (vsip_scalar_real operand, vsip_vview_complex *result);
  void        div (vsip_vview_complex *operand, vsip_vview_complex *result);
  void        div (vsip_scalar_complex *operand, vsip_vview_complex *result);

  void        jdot (vsip_vview_complex *operand, vsip_scalar_complex &ret);
  void        dot (vsip_vview_complex *operand, vsip_scalar_complex &ret);
  void        jmul (vsip_vview_complex *operand, vsip_vview_complex *result);

  void        fill (vsip_scalar_real value);
  void        fill (vsip_scalar_complex *value);

  void        ramp (vsip_scalar_real alpha, vsip_scalar_real beta);
  void        ramp (vsip_scalar_complex *alpha, vsip_scalar_complex *beta);

  void        real (vsip_vview_real *result);
  void        imag (vsip_vview_real *result);
  void        cmplx (vsip_vview_real *real_part, vsip_vview_real *imag_part);
  void        conj  (vsip_vview_complex *result);

  ~vsip_cvview_f(void);
};

class vsip_vview_f : public vsip_vview_float
{
public:
  vsip_view *       cloneview(void);
  vsip_vview *      vcloneview(void);
  vsip_vview_real * vcloneview_real(void);
  vsip_vview_float *vcloneview_float(void);
  vsip_vview_f *    vcloneview_f (void);

  vsip_vview *      vsubview(vsip_index index, vsip_length length);
  vsip_vview_real * vsubview_real(vsip_index index, vsip_length length);
  vsip_vview_float *vsubview_float(vsip_index index, vsip_length length);
  vsip_vview_f *    vsubview_f(vsip_index index, vsip_length length);

  vsip_block *      getblock();
  vsip_block_real * getblock_real();
  vsip_block_float *getblock_float();
  vsip_block_f *    getblock_f();

  void        print (void);

  void        getattrib (vsip_vattr &attrib);
  void        putattrib (vsip_vattr &attrib);
  void        putstride (vsip_stride stride);
  void        putlength (vsip_length length);
  void        putoffset (vsip_offset offset);

  void              put (vsip_index i, vsip_scalar_real j);
  void              put (vsip_index i, vsip_scalar_f j);
  vsip_scalar_real  get_real (vsip_index i);
  vsip_scalar_float get_float (vsip_index i);
  vsip_scalar_f     get_f (vsip_index i);

  void        copy (vsip_vview_real *source);

  void        atan(vsip_vview_float *result);
  void        atan2 (vsip_vview_float *denominators, vsip_vview_float *result);
  void        cos(vsip_vview_float *result);

  void        mag (vsip_vview_float *result);

  void        square (vsip_vview_real *result);

  vsip_scalar_f   sumval (void);
  vsip_scalar_f   sumsqval (void);

  void        add (vsip_vview_real *operand, vsip_vview_real *result);
  void        add (vsip_scalar_real operand, vsip_vview_real *result);
  void        add (vsip_vview_complex *operand, vsip_vview_complex *result);
  void        add (vsip_scalar_complex *operand, vsip_vview_complex *result);

  void        sub (vsip_vview_real *operand, vsip_vview_real *result);
  void        sub (vsip_scalar_real operand, vsip_vview_real *result);
  void        sub (vsip_vview_complex *operand, vsip_vview_complex *result);
  void        sub (vsip_scalar_complex *operand, vsip_vview_complex *result);

  void        mul (vsip_vview_real *operand, vsip_vview_real *result);
  void        mul (vsip_scalar_real operand, vsip_vview_real *result);
  void        mul (vsip_vview_complex *operand, vsip_vview_complex *result);
  void        mul (vsip_scalar_complex *operand, vsip_vview_complex *result);

  void        div (vsip_vview_real *operand, vsip_vview_real *result);
  void        div (vsip_scalar_real operand, vsip_vview_real *result);
  void        div (vsip_vview_complex *operand, vsip_vview_complex *result);
  void        div (vsip_scalar_complex *operand, vsip_vview_complex *result);

  void        exp (vsip_vview_float *result);

  void        log (vsip_vview_float *result);
  void        log10 (vsip_vview_float *result);

  void        sin (vsip_vview_float *result);

  void        sqrt (vsip_vview_float *result);

  void        neg (vsip_vview_real *result);

  void        recip (vsip_vview_float *result);

  vsip_scalar_float   dot (vsip_vview_float *operand);
```

```
  void        max (vsip_vview_real *compare, vsip_vview_real *result);

  vsip_scalar_real maxval(vsip_index &index);
  vsip_scalar_f   maxval_f (vsip_index &index);

  void        min (vsip_vview_real *compare, vsip_vview_real *result);

  vsip_scalar_real  minval (vsip_index &index);
  vsip_scalar_f     minval_f (vsip_index &index);

  void        fill (vsip_scalar_real value);
  void        fill (vsip_scalar_f value);
  void        ramp (vsip_scalar_real alpha, vsip_scalar_real beta);
  void        ramp (vsip_scalar_f alpha, vsip_scalar_f beta);

  void        histo (vsip_scalar_real min_bin, vsip_scalar_real max_bin, vsip_hist_opt opt, vsip_vview_real *result);
};

class vsip_vview_i : public vsip_vview_int
{
public:
  vsip_view *       cloneview(void);
  vsip_vview *      vcloneview(void);
  vsip_vview_real * vcloneview_real(void);
  vsip_vview_int *  vcloneview_int(void);
  vsip_vview_i *    vcloneview_i (void);

  vsip_vview *      vsubview(vsip_index index, vsip_length length);
  vsip_vview_real * vsubview_real(vsip_index index, vsip_length length);
  vsip_vview_int *  vsubview_int(vsip_index index, vsip_length length);
  vsip_vview_i *    vsubview_i(vsip_index index, vsip_length length);

  vsip_block *      getblock();
  vsip_block_real * getblock_real();
  vsip_block_int *  getblock_int();
  vsip_block_i *    getblock_i();

  void        print (void);

  void        getattrib (vsip_vattr &attrib);
  void        putattrib (vsip_vattr &attrib);
  void        putstride (vsip_stride stride);
  void        putlength (vsip_length length);
  void        putoffset (vsip_offset offset);

  void        put (vsip_index i, vsip_scalar_real j);

  void        copy (vsip_vview_real *source);

  void        neg (vsip_vview_real *result);

  vsip_scalar_real  get_real (vsip_index i);
  vsip_scalar_int   get_int (vsip_index i);
  vsip_scalar_i     get_i (vsip_index i);

  void        mag (vsip_vview_float *result);

  vsip_scalar_f   sumval (void);
  vsip_scalar_f   sumsqval (void);

  void        square (vsip_vview_real *result);

  void        add (vsip_vview_real *operand, vsip_vview_real *result);
  void        add (vsip_scalar_real operand, vsip_vview_real *result);
  void        add (vsip_vview_complex *operand, vsip_vview_complex *result);
  void        add (vsip_scalar_complex *operand, vsip_vview_complex *result);

  void        sub (vsip_vview_real *operand, vsip_vview_real *result);
  void        sub (vsip_scalar_real operand, vsip_vview_real *result);
  void        sub (vsip_vview_complex *operand, vsip_vview_complex *result);
  void        sub (vsip_scalar_complex *operand, vsip_vview_complex *result);

  void        mul (vsip_vview_real *operand, vsip_vview_real *result);
  void        mul (vsip_scalar_real operand, vsip_vview_real *result);
  void        mul (vsip_vview_complex *operand, vsip_vview_complex *result);
  void        mul (vsip_scalar_complex *operand, vsip_vview_complex *result);

  void        div (vsip_vview_real *operand, vsip_vview_real *result);
  void        div (vsip_scalar_real operand, vsip_vview_real *result);
  void        div (vsip_vview_complex *operand, vsip_vview_complex *result);
  void        div (vsip_scalar_complex *operand, vsip_vview_complex *result);

  void              max (vsip_vview_real *compare, vsip_vview_real *result);
  vsip_scalar_real  maxval (vsip_index &index);
  void              min (vsip_vview_real *compare, vsip_vview_real *result);
  vsip_scalar_real  minval (vsip_index &index);

  void        histo (vsip_scalar_real min_bin, vsip_scalar_real max_bin, vsip_hist_opt opt, vsip_vview_real *result);
  void        fill (vsip_scalar_real value);
  void        ramp (vsip_scalar_real alpha, vsip_scalar_real beta);
};

class vsip_randstate
{
public:
  void          randu (vsip_vview_real *output);
  int           is_valid() { return is_valid_;};
};

class vsip_scalar_complex
{
public:
  virtual vsip_scalar_real  real(void) = 0;
  virtual vsip_scalar_real  imag(void) = 0 ;
  virtual void              complex (vsip_scalar_real real_part, vsip_scalar_real imag_part) = 0;
};
```

```
class vsip_cscalar_f : public vsip_scalar_complex
{
public:
  vsip_scalar_real    real(void);
  vsip_scalar_f       real_f (void);
  vsip_scalar_real    imag(void);
  vsip_scalar_f       imag_f(void);
  void                complex (vsip_scalar_real real_part, vsip_scalar_real imag_part);
};

class vsip_fir_complex
{
public:
  virtual void    firflt (vsip_vview_complex *intput, vsip_vview_complex *output) = 0;
  inline int      is_valid(void) {return is_valid_;};
};

class vsip_cfir_f : public vsip_fir_complex
{
public:
  void            firflt (vsip_vview_complex *intput, vsip_vview_complex *output);
};

class vsip_fir_float
{
public:
  inline int      is_valid(void) {return is_valid_;};
  virtual void    firflt (vsip_vview_float *intput, vsip_vview_float *output) = 0;
};

class vsip_fir_f : public vsip_fir_float
{
public:
  void            firflt (vsip_vview_float *intput, vsip_vview_float *output);
};


class vsip_fft
{
public:
  virtual void  fft (vsip_vview_float *input, vsip_vview_complex *output) = 0;
  virtual void  fft (vsip_vview_complex *input, vsip_vview_float *output) = 0;
  virtual void  fft (vsip_vview_complex *input, vsip_vview_complex *output) = 0;
  inline  int   is_valid(void) {return is_valid_;};
};

class vsip_fft_f : public vsip_fft
{
public:
  void        fft (vsip_vview_float *input, vsip_vview_complex *output);
  void        fft (vsip_vview_complex *input, vsip_vview_float *output);
  void        fft (vsip_vview_complex *input, vsip_vview_complex *output);
};
```

Appendix B
Block and View Class Hierarchy


vsip_view
vsip_mview
vsip_vview
vsip_tview
vsip_vview_real
vsip_vview_complex
vsip_vview_int
vsip_vview_float
vsip_cvview_f
vsip_vview_d
vsip_vview_i
vsip_vview_l
vsip_vview_f
vsip_vview_d


vsip_block
vsip_block_real
vsip_block_complex
vsip_block_int
vsip_block_float
vsip_cblock_f
vsip_cblock_d
vsip_block_i
vsip_block_l
vsip_block_f
vsip_block_d

Appendix C
Sample Source Code

add.cpp:

```
int main (void)
{
  vsip v;
  vsip_vview_f *view1;
  vsip_vview_f *view2;
  vsip_vview_f *view3;
  int start, end;
  unsigned int i;

  printf ("VSIP C++ Add Benchmark Test program\n");

  for (i=0; i<elements(VECTOR_SIZE); i++)
  {
    int size = VECTOR_SIZE[i];
    view1 = v.vcreate_f (size);
    view2 = v.vcreate_f (size);
    view3 = v.vcreate_f (size);

    if (!view1 || !view2 || !view3)
    {
      if (view1) v.valldestroy (view1);
      if (view2) v.valldestroy (view2);
      if (view3) v.valldestroy (view3);
      printf ("Error creating vectors for size=%i\n", size);
      exit(1);
    }
    view1->ramp (1.0f, 2.0f);
    view2->ramp (3.0f, 4.0f);
    start = clock();
    for (int j=0; j<REPETITIONS; j++)
    {
      view1->add(view2, view3);
    }
    end = clock();
    float time = (float)(end-start)/(float)(CLOCKS_PER_SEC);
    printf ("For vector size=%i operation=add, repetitions=%i time=%f\n",
             size, REPETITIONS, time);
    v.valldestroy (view1);
    v.valldestroy (view2);
    v.valldestroy (view3);
  }
  return (0);
}
```

add.c:

```
int main (void)
{
  vsip_vview_f *view1;
  vsip_vview_f *view2;
  vsip_vview_f *view3;
  int start, end;
  unsigned int i, j;
  float time;

  printf ("VSIP C Add Benchmark Test program\n");
  vsip_init ((void *)0);

  for (i=0; i<elements(VECTOR_SIZE); i++)
  {
    int size = VECTOR_SIZE[i];
    view1 = vsip_vcreate_f (size, VSIP_MEM_NONE);
    view2 = vsip_vcreate_f (size, VSIP_MEM_NONE);
    view3 = vsip_vcreate_f (size, VSIP_MEM_NONE);

    if (!view1 || !view2 || !view3)
    {
      if (view1) vsip_valldestroy_f (view1);
      if (view2) vsip_valldestroy_f (view2);
      if (view3) vsip_valldestroy_f (view3);
      printf ("Error creating vectors for size=%i\n", size);
      exit(1);
    }
    vsip_vramp_f (1, 2, view1);
    vsip_vramp_f (3, 4, view2);
    start = clock();
    for (j=0; j<REPETITIONS; j++)
    {
      vsip_vadd_f (view1, view2, view3);
    }
    end = clock();
    time = (float)(end-start)/(float)(CLOCKS_PER_SEC);
    printf ("For vector size=%i operation=add, repetitions=%i time=%f\n",
             size, REPETITIONS, time);
    vsip_valldestroy_f (view1);
    vsip_valldestroy_f (view2);
    vsip_valldestroy_f (view3);
  }
  return 0;
}
```

sigproc_model.cpp:

```
const vsip_length LENGTH = 1024;            /* Length of data, blocks, and vectors */
const vsip_length N = LENGTH;
#define BITS 12               /* #Bits of quantization (2's complelement) */
#define FULLSCALE 4.8828125e-004        /* 1/(2^(BITS-1)) */

vsip *the_vsip;

class function_operator
{
public:
  function_operator();
  void do_function (vsip_vview_real *vec_I,
                    vsip_vview_real *vec_Q,
                    vsip_vview_real *vec_w,
                    vsip_scalar_f &det,
                    vsip_scalar_vi &det_index);
private:
  vsip_vview_f *rx_I, *rx_Q, *Rx_mag;
  vsip_cvview_f *rx_raw, *rx, *rx_w, *Rx;
  vsip_fft_f *fft_obj;
};

function_operator::function_operator (void)
{
  vsip_scalar_f scale = 1;
  vsip_length ntimes = 1;
  rx_raw = the_vsip->cvcreate_f(N);          /* Create complex vector for raw input I/Q signal */
  rx     = the_vsip->cvcreate_f(N);              /* Create complex vector for scaled/normalized I/Q
             signal */
  rx_w   = the_vsip->cvcreate_f(N);              /* Create complex vector for windowed input signal */
  rx_I   = rx_raw->realview_f();          /* Create vector view of Re[rx] */
  rx_Q   = rx_raw->imagview_f();          /* Create vector view of Im[rx] */
  Rx     = the_vsip->cvcreate_f(N);              /* Create FFT output vector */
  Rx_mag = the_vsip->vcreate_f(N);           /* Create |FFT| vector */
  fft_obj= the_vsip->ccfftop_f(LENGTH, scale,VSIP_FFT_FWD, ntimes,VSIP_ALG_TIME);  /* Create FFT object
             */
}

void function_operator::do_function (vsip_vview_real *vec_I,
                                     vsip_vview_real *vec_Q,
                                     vsip_vview_real *vec_w,
                                     vsip_scalar_f &det,
                                     vsip_scalar_vi &det_index)
{
  vsip_scalar_f full_scale=FULLSCALE;
  vsip_scalar_vi temp;

  rx_I->copy (vec_I);
  rx_Q->copy (vec_Q);
  rx_raw->mul (full_scale, rx);     /* Scale A/D values by full-scale */
  vec_w->mul (rx, rx_w);            /* Apply Blackman window to input signal */
  fft_obj->fft (rx_w, Rx);          /* Compute FFT of windowed input signal */
  Rx->mag (Rx_mag);                 /* Compute magnitude of FFT to get spectral estimate */

  det=Rx_mag->maxval (temp);  /* Find max bin value and bin index */
  det_index = temp;
}
int main(int argc, char *argv[])
{
  vsip_scalar_i I[N], /* In-phase (real) part of received A/D'd signal */
              Q[N]; /* Quadrature-phase (imaginary) part of received A/D'd signal */
  vsip_scalar_f w[N]; /* Window function to apply to received signal before FFT */
  vsip_scalar_vi det_index = 0, target_det_index =  0;
  vsip_scalar_f det = 0, target_det = 0;

  function_operator op;

  vsip_block_i *blk_I, *blk_Q;
  vsip_block_f *blk_w;
  vsip_vview_i *vec_I, *vec_Q;
  vsip_vview_f *vec_w;

  vsip_scalar_f I2[N], Q2[N], w2[N];
  vsip_block_f *blk_I2, *blk_Q2, *blk_w2;
  vsip_vview_f *vec_I2, *vec_Q2, *vec_w2;

  int i;
  int start, end;
  float time;
```

```
  the_vsip = new vsip();

  blk_I  = the_vsip->blockbind_i(&I[0],N);        /* Bind block to I user array */
  blk_Q  = the_vsip->blockbind_i(&Q[0],N);        /* Bind block to Q user array */
  blk_w  = the_vsip->blockbind_f(&w[0],N);        /* Bind block to w user array */
  vec_I  = blk_I->vbind_i((vsip_offset)0, (vsip_stride)1, N);     /* Bind view to blk_I block */
  vec_Q  = blk_Q->vbind_i((vsip_offset)0, (vsip_stride)1, N);     /* Bind view to blk_Q block */
  vec_w  = blk_w->vbind_f((vsip_offset)0, (vsip_stride)1, N);     /* Bind view to blk_w block */

  read_file ("../data/IQdata", &target_det, &target_det_index, w, I, Q);

  blk_w->admit (1);     /* Admit blk_w after populating window user data array */
  blk_I->admit (1);     /* Admit blk_I after populating input-I user data array */
  blk_Q->admit (1);     /* Admit blk_Q after populating input-Q user data array */


  // repeat all that, but do everything in floating point this time.

  blk_I2  = the_vsip->blockbind_f(&I2[0],N);        /* Bind block to I user array */
  blk_Q2  = the_vsip->blockbind_f(&Q2[0],N);        /* Bind block to Q user array */
  blk_w2  = the_vsip->blockbind_f(&w2[0],N);        /* Bind block to w user array */
  vec_I2  = blk_I2->vbind_f((vsip_offset)0, (vsip_stride)1, N); /* Bind view to blk_I block */
  vec_Q2  = blk_Q2->vbind_f((vsip_offset)0, (vsip_stride)1, N); /* Bind view to blk_Q block */
  vec_w2  = blk_w2->vbind_f((vsip_offset)0, (vsip_stride)1, N); /* Bind view to blk_w block */

  read_file2 ("../data/IQdata", &target_det, &target_det_index, w2, I2, Q2);

  blk_w2->admit (1);     /* Admit blk_w after populating window user data array */
  blk_I2->admit (1);     /* Admit blk_I after populating input-I user data array */
  blk_Q2->admit (1);     /* Admit blk_Q after populating input-Q user data array */

  printf ("loaded, starting %i repetitions\n", COMPLEX_REPETITIONS);
  start = clock();
  for (i=0; i<COMPLEX_REPETITIONS; i++)
  {
    op.do_function (vec_I, vec_Q, vec_w, det, det_index);
  }
  end = clock();
  time = (float)(end-start)/(float)(CLOCKS_PER_SEC);
  printf ("For repetitions=%i time=%f\n", COMPLEX_REPETITIONS, time);

  printf ("for fixed point input det=%f det_index=%i target is (%f, %i)\n",
           det, (int)det_index, target_det, (int)target_det_index);

  start = clock();
  for (i=0; i<COMPLEX_REPETITIONS; i++)
  {
    op.do_function (vec_I2, vec_Q2, vec_w2, det, det_index);
  }
  end = clock();
  time = (float)(end-start)/(float)(CLOCKS_PER_SEC);
  printf ("For repetitions=%i time=%f\n", COMPLEX_REPETITIONS, time);

  printf ("for float input det=%f det_index=%i target is (%f, %i)\n",
           det, (int)det_index, target_det, (int)target_det_index);

  start = clock();
  for (i=0; i<COMPLEX_REPETITIONS; i++)
  {
    op.do_function (vec_I, vec_Q2, vec_w, det, det_index);
  }
  end = clock();
  time = (float)(end-start)/(float)(CLOCKS_PER_SEC);
  printf ("For repetitions=%i time=%f\n", COMPLEX_REPETITIONS, time);

  printf ("for mixed input det=%f det_index=%i target is (%f, %i)\n",
           det, (int)det_index, target_det, (int)target_det_index);

  delete the_vsip;

  return 0;
}
```

sigproc_model.c:

```
#define LENGTH 1024              /* Length of data, blocks, and vectors */
#define N LENGTH
#define BITS 12                  /* #Bits of quantization (2's complelement) */
#define FULLSCALE 4.8828125e-004        /* 1/(2^(BITS-1)) */
vsip_vview_f *rx_I, *rx_Q, *Rx_mag;
vsip_cvview_f *rx_raw, *rx, *rx_w, *Rx;
vsip_fft_f *fft_obj;

void init_function (void)
{
  rx_raw = vsip_cvcreate_f(N,0);          /* Create complex vector for raw input I/Q signal */
  rx     = vsip_cvcreate_f(N,0);              /* Create complex vector for scaled/normalized I/Q signal
             */
  rx_w   = vsip_cvcreate_f(N,0);             /* Create complex vector for windowed input signal */
  rx_I   = vsip_vrealview_f(rx_raw);          /* Create vector view of Re[rx] */
  rx_Q   = vsip_vimagview_f(rx_raw);          /* Create vector view of Im[rx] */
  Rx     = vsip_cvcreate_f(N,0);              /* Create FFT output vector */
  Rx_mag = vsip_vcreate_f(N,0);           /* Create |FFT| vector */
  fft_obj= vsip_ccfftop_create_f(LENGTH,1,VSIP_FFT_FWD,1,VSIP_ALG_TIME);  /* Create FFT object */
}

void do_function (vsip_vview_i *vec_I,
                  vsip_vview_i *vec_Q,
                  vsip_vview_f *vec_w,
                  vsip_scalar_f *det,
                  vsip_scalar_vi *det_index)
{
  vsip_scalar_f full_scale=FULLSCALE,
                fft_scale=1.0;

  vsip_vcopy_i_f(vec_I,rx_I);           /* Copy/convert (int)I input to (float)Re[rx] */
  vsip_vcopy_i_f(vec_Q,rx_Q);           /* Copy/convert (int)Q input to (float)Im[rx] */
  vsip_rscvmul_f (full_scale,rx_raw,rx);   /* Scale A/D values by full-scale */
  vsip_rcvmul_f (vec_w,rx,rx_w);           /* Apply Blackman window to input signal */
  vsip_ccfftop_f(fft_obj,rx_w,Rx);      /* Compute FFT of windowed input signal */
  vsip_cvmag_f (Rx, Rx_mag);                 /* Compute magnitude of FFT to get spectral estimate */

  *det=vsip_vmaxval_f(Rx_mag, det_index); /* Find max bin value and bin index */
}

int main(int argc, char *argv[])
{
  vsip_scalar_i I[N], /* In-phase (real) part of received A/D'd signal */
                Q[N]; /* Quadrature-phase (imaginary) part of received A/D'd signal */
  vsip_scalar_f w[N]; /* Window function to apply to received signal before FFT */
  vsip_scalar_vi det_index, target_det_index;
  vsip_scalar_f det, target_det;

  vsip_block_i *blk_I, *blk_Q;
  vsip_block_f *blk_w;
  vsip_vview_i *vec_I, *vec_Q;
  vsip_vview_f *vec_w;

  int i;
  int start, end;
  float time;

  vsip_init((void *)0);

  blk_I  = vsip_blockbind_i(I,N,0);        /* Bind block to I user array */
  blk_Q  = vsip_blockbind_i(Q,N,0);        /* Bind block to Q user array */
  blk_w  = vsip_blockbind_f(w,N,0);        /* Bind block to w user array */
  vec_I  = vsip_vbind_i(blk_I, 0, 1, N);            /* Bind vector view to blk_I block */
  vec_Q  = vsip_vbind_i(blk_Q, 0, 1, N);            /* Bind vector view to blk_Q block */
  vec_w  = vsip_vbind_f(blk_w, 0, 1, N);            /* Bind vector view to blk_w block */

  read_file ("../data/IQdata", &target_det, &target_det_index, w, I, Q);
  vsip_blockadmit_f(blk_w, 1);     /* Admit blk_w after populating window user data array */
  vsip_blockadmit_i(blk_I, 1);     /* Admit blk_I after populating input-I user data array */
  vsip_blockadmit_i(blk_Q, 1);     /* Admit blk_Q after populating input-Q user data array */

  init_function();

  start = clock();
  for (i=0; i<COMPLEX_REPETITIONS; i++)
  {
    do_function (vec_I, vec_Q, vec_w, &det, &det_index);
  }
```

```
  end = clock();
  time = (float)(end-start)/(float)(CLOCKS_PER_SEC);
  printf ("For repetitions=%i time=%f\n", COMPLEX_REPETITIONS, time);

  printf ("det=%f det_index=%i target is (%f, %i)\n", det, (int)det_index, target_det,
              (int)target_det_index);

  vsip_finalize ((void *)0);
  return 0;
}
```

Appendix D
Performance Benchmarks

All values are seconds for total run of given number of repetitions.

add, 1048576 repetitions

Vector Size:	C	C++
16	0.74	0.82
64	1.42	1.52
256	4.26	4.42
512	7.98	8.17
1024	15.52	16.2

mul, 1048576 repetitions

Vector Size:	C	C++
16	0.75	0.79
64	1.44	1.56
256	4.23	4.35
512	7.98	8.08
1024	15.47	16.16

cmul, 1048576 repetitions

Vector Size:	C	C++
16	1.28	1.37
64	2.99	3.02
256	9.72	9.85
512	18.69	19.76
1024	37.50	37.97

div, 1048576 repetitions

Vector Size:	C	C++
16	1.22	1.27
64	3.46	3.51
256	12.44	12.60
512	24.44	24.67
1024	48.38	49.18

fft, 524288 repetitions

Vector Size:	C	C++
16	4.12	4.58
64	11.88	12.57
256	57.23	59.25
512	106.01	104.52
1024	214.08	209.13

fft2, 524288 repetitions

Vector Size:	C	C++
16	5.81	5.88
64	16.25	16.42
256	88.37	89.24
512	178.21	178.83
1024	514.72	514.31

abstract1, 524288 repetitions

Matched float:

Vector Size:	C	C++
16	0.95	1.34
64	1.94	2.33
256	5.84	6.25
512	15.8	17.36
1024	38.43	28.58

Matched Int

Vector Size:	C	C++
16		19.94
64		23.42
256		38.49
512		56.57
1024		106.93

Float, Int to Float

Vector Size:	C	C++
16		4.13
64		5.48
256		11.2
512		18.77
1024		41.58

Float, Float to Int

Vector Size:	C	C++
16		14.75
64		17.54
256		29.83
512		55.16
1024		90.73

sigproc_model, 16384 repetitions

Vector Size = 1024	C	C++
Fixed point input:	22.11	22.61
Floating point input:		22.48
mixed input:		22.56