

CHAPTER 2

# Overview of the Wideband TMS320C3x/4x Optimized Math Library

---

## 2.1 Manual Contents

---

The Wideband Optimized Math Library is a high performance scalar math, vector, digital signal and image processing library for the TMS320C3x and TMS320C4x family of digital signal processor. The routines contained within the library are organized into 45 functional categories. Chapter 5 contains a detailed description of each function to includes the name, description, algorithm, call synopsis, domain, accuracy, execution times, and any applicable notes. The functional categories for the Optimized Math Library functions include:

- Scalar Power Functions
- Simple Scalar Trigonometric Functions
- Simple Scalar Functions
- Scalar Inverse Trigonometric Functions
- Scalar Hyperbolic Functions
- Scalar Inverse Hyperbolic Functions

- Scalar Fix, Float & Truncation Functions
- Simple Functions
- Power Functions
- Simple Trigonometric Functions
- Arc Trigonometric Function
- Hyperbolic Function
- Inverse Hyperbolic Functions
- Averaging and Summing Functions
- Comparison Functions
- Vector Fix, Float and Truncation Functions
- Limiting Functions
- Logical Functions
- 2-Vector & 1-Scalar Functions
- 2-Vector & 2-Scalar Math Functions
- 3-Vector Math Functions
- 3-Vector & 1-Scalar Functions
- 4-Vector Math Functions
- 5-Vector Math Functions
- Maximum/Minimum Functions
- Gather/Scatter Functions
- Comparison Functions
- Conversion Functions
- Other Functions
- Complex Functions
- Complex Vector Magnitude Functions
- Complex Conversion Functions
- Complex Vector Simple Functions
- Complex Vector Fundamental Functions
- Complex Vector Real Vector Functions
- Complex Vector Conjugation Functions
- Complex Vector Complex Scalar Functions
- Convolution Functions
- Correlation Functions
- Accumulating Spectrum Functions
- Filtering Functions (Both FIR & IIR)
- Windowing Functions
- FFT Functions Functions
- Integration Functions
- Distribution Generation Functions

## 2.2 Optimized Math Library

---

An optimized library is a collection of assembly language routines which are typically used in math-intensive applications. The Wideband Optimized Math Library is a collection of hand-optimized math, signal and image processing routines which are coded entirely in TMS320C3x or TMS320C4x assembly language. These routines are typically used in computationally intensive applications where the need for optimal execution speed is an important concern (in real-time applications, for instance). By carefully exploiting the TMS320C3x and TMS320C4x register set and utilizing TMS320 parallel instructions when possible, the Wideband optimized routines achieve execution times which are typically 50% to 200% faster (depending on the routine chosen) than comparable code written in the standard ANSI C language.

## 2.3 Definition of a Vector Library

---

Vectors are often defined as aggregates of data of the same type which typically relate to a phenomenon or problem you are attempting to analyse. If you compute the mean of a series of numbers, for example, you are performing a vector operation, as all data points for a defined array are being summed and divided by the number of total data points, **n**, to compute the average. For purposes of the Wideband Optimized Math Library, vectors are considered to be composed of floating-point elements, integer elements, or unsigned integer elements. From these elemental types Wideband has constructed a variety of efficient routines to perform most vector functions. For example, an aggregate of floating-point numbers is used to define complex data type which are extended to define complex vector operations.

With the exception of the scalar routines, all Optimized Math Library vector routine all numbers are held in memory as a vector, which we define as an array of numbers held in consecutive memory locations. Scalar routines could be considered to be a special form of a vector, where one memory location contains a one real number.

## 2.4 Vector Versus Scalar Operations

---

Vector operations are designed to perform repetitive tasks on a large grouping of data. This is opposed to performing a scalar operation where a routine is called once, supplied with one argument (typically) and produces one result.

For instance, the scalar cosine routine, called **cos**, accepts one input argument each time the routine is called. It will produce one answer for each argument supplied, but the routine must be repetitively called to compute multiple cosines. Alternately, the vector version of the cosine routine, called **vcos**, also accepts one argument and produces one answer. The difference is that the **vcos** vector routine can accept an array of input arguments which produces one answer each traversal through the code.

The two routines differ in that the **vcos** vector routine is specifically built to calculate multiple iterations of the cosine, as the code has an optimized loop built into its core. The scalar routine, **cos** is built to be called only once, and traversed only once per call.

All vector routines in the Optimized Math Library are optimized to perform repetitive operations on large groups of data. However, note that due to the extensive use of parallel instructions within the code, we suggest that you run each routine with at least 2 data points to avoid problems associated with flushing the TMS320 parallel instruction pipeline. The scalar routines, on the other hand, are designed to be called with only one argument.

---

## 2.5 Optimization Principles

---

The key to the Optimized Math Library speed is based on four principles:

- A special assembly language instruction called RPTB or repeat block is implemented in all vector routines so that no overhead becomes associated with repetitive traversals of a main loop outside of 4 initial cycles necessary set up the zero overhead loop.
- Parallel instructions are used whenever possible. Current compiler technology has not yet reached the level of optimization inherent in a careful, hand assembled code. Part of this is due to the difficulty of building a compiler which takes full advantage of the TMS320 native parallel instruction set. Implementing vector operations in assembler code demands an implementation that avoids pipeline delays and illegal use of registers. Hand optimized implementations, as found in the Optimized Math Library library, results in code which contains sophisticated repeat block structures which even the most intelligent compiler would be hard pressed to match. This results in a performance increase of up to 20 to 200% over the same code written in fully optimized C.
- Careful coding techniques insure that the processor pipeline is always loaded with instructions necessary to perform a computation. Techniques are implemented which avoid the use of inefficient constructs and unnecessary branching.
- The inner loops are kept to a minimum size to by careful algorithm design techniques.

---

## 2.6 Basic Striding Concepts

---

Vector operations typically work on large aggregates of data. Most of the time, all of the data will be used. However, sometimes a user may wish to work with only selected portions of data.

In such cases, the user would instruct the computer to skip over, or ignore, a desired portion of data, and only read every *n* data points. This skipping of data is called *striding*. It is a controlled way to skip reading memory locations (data) without having to first process data before discarding it. In this manner, precious CPU cycles are saved as specific data is designated as not needing processing.

Most arrays (designated **a**, **b**, **c**, **d**, and **e**, etc.) in the Wideband Optimized Math Library have strides associated with them. For instance, the routine **vadd** adds the elements of array **a** and array **b** and places the results in array **c**. The **vadd** routine is called as follows:

**vadd (a, 1, b, 1, c, 1, 6)**

The integer following each array representation (**a**, **b**, **c**) represents the stride length assigned to the preceding array. In the case of the **vadd** example just shown, the stride length is 1. The preceding call states that the **vadd** routine is to add every corresponding element in input vector **a** to every corresponding input element in vector **b** and place the sum in the corresponding memory location defined by the index in array **c**. This is demonstrated as follows:

**TABLE 1**

**Vector Add - Vadd (a, 1, b, 1, c, 1, 6) - All Strides Equal**

Index Of Array	Vector a Value	Stride i Value	Vector b Value	Stride j Value	Vector C Value	Stride k Value
0	3	1	4	1	7	1
1	6	1	8	1	14	1
2	3	1	5	1	8	1
3	7	1	3	1	10	1
4	2	1	5	1	7	1
5	9	1	7	1	16	1

This, however, can be changed. If we wished to skip a portion of input data and only sample as 1/2 the rate, we would do this by skipping every other data point. In effect, we would leap-frog the odd samples. We could do this with the following call to **vadd**:

**vadd (a, 2, b, 2, c, 2, 6)**

The preceding call states that the **vadd** routine is to add every other element in input vector **a** to every other item in input vector **b** and place the sum in every other position in output array **c**. In effect, only half the data points will be sampled. Six represents the number of data samples to be considered by the routine. This can be demonstrated as follows (with the shaded boxes indicating which elements are used as both input and output values):

**TABLE 2**

**Vector Add - Vadd (a, 2, b, 2, c, 2, 6) - All Strides Equal But 1/2 Data Skipped**

Index Of Array	Vector a Value	Stride i Value	Vector b Value	Stride j Value	Vector C Value	Stride k Value
0	3	2	4	2	7	2
1	6	2	8	2	0	2
2	3	2	5	2	8	2
3	7	2	3	2	0	2
4	2	2	5	2	7	2
5	9	2	7	2	0	2

We could perform one final optimization on this routine by assigning the output vector **c** a stride of 1. This avoids filling half the memory of output array **c** with zeroes for those memory locations that are not assigned results. The call would be as follows:

`vadd (a, 2, b, 2, c, 1, 16)`

This would be demonstrated as follows:

**TABLE 3**

**Vector Add - Vadd (a, 2, b, 2, c, 1, 6) - All Strides Not Equal And 1/2 Data Skipped**

Index Of Array	Vector a Value	Stride i Value	Vector b Value	Stride j Value	Vector C Value	Stride k Value
0	3	2	4	2	7	1
1	6	2	8	2	8	1
2	3	2	5	2	7	1
3	7	2	3	2	0	1
4	2	2	5	2	0	1
5	9	2	7	2	0	1

In this case, the output vector **c** has results shown in array positions 0, 1 and 2 even though only values in positions 0, 2, and 4 of vectors **a** and **b** are used as the input arguments. In effect, every other input value of vector **a** (shown in shaded area) is added to a corresponding input value in vector **b** (shown in shaded area) and placed in contiguous output locations in vector **c** (shown as contiguous values 7, 8, 7).

You may stride through arrays in any stride increment you choose as long as the size of your stride is not larger than the size of its associated array. For instance, you may not specify that you expect to process 10 data points by setting **n** equal to 10 and then specify 11 as any one of the striding arguments. Also, the integer representing the stride value must be within the valid range of single precision integers acceptable to the TMS320 processor:  $2^{31} \leq x \leq 2^{31} - 1$ .

---

## 2.7 Register Contention Issues And The Use Of Index Registers

---

Special code is implemented in the Optimized Math Library to handle a register contention problem associated with implementing optimized striding. The TMS320 series processor has 2 special registers, called index registers which are designated as **IR0** and **IR1**, which are available as loop counters for integers which are commonly used to represent strides. The registers are intimately associated with the repeat block instruction (RPTB), where these registers are used to implement zero overhead looping. For instance, you can call the **vmov** routine as follows:

`vmov (a, 1, c, 1, 15)`

The elements within input vector **a** will be moved to output vector **c**. The actual implementation of the routine in assembly language will utilize index register **IR0** to update the stride of input vector **a**, while the stride of output vector **c** will be updated by index register **IR1**.

This works fine for math operations where only 2 vectors are involved. However, if more than 2 vectors are involved, then some mechanism must be implemented to

account for the strides of the arrays that were not assigned index registers **IR0** and **IR1**. For instance, the **vadd** routine adds the contents of input vector **a** and input vector **b** and stores the results in output vector **c**. It is called as follows:

```
vadd ( a, 1, b, 1, c, 1, 15)
```

The actual implementation of this routine in assembly language will utilize index register **IR0** to update the stride of input vector **a**, index register **IR1** to update the stride of input vector **b**, while the stride of output vector **c** will be updated by an extended precision register, say **R0**. What this means is that any routine which has more than two strides associated with it must have some mechanism to update the third unassigned stride that remains after index registers **IR0** and **IR1** are assigned the task of updating the first two stride arguments. This can be done by assigning other registers (such as **R0** or **R1**, etc.) the job of tracking and updating the leftover stride for vector **c** as each iteration of the inner-loop unfolds.

---

## 2.8 Performance Degradation Resulting From Unequal Striding

---

From a programmer's point of view, this is not desirable unless absolutely necessary. The reasons are as follows:

- You will incur a performance penalty. The inner-loop time is lengthened by 1 instruction (necessary to update the pointer to the next memory location) for each stride assigned an independent registers such as **R0**, **R1**, etc. The mechanism used to update pointer to next memory location has to be placed in the inner-loop, and therefore detracts from the execution of the time of the routine each time the loop is traversed.
- Inefficient code may be developed as register contention issues (too many variables seeking too few registers) may arise. When strides are assigned to registers such as **R0**, **R1**, etc., in addition to the standard 2 index registers, this limits the number of remaining registers available to compute other important processes with. Although this is not typical in the simpler types of vector operations, the more sophisticated operations such as the trigonometric functions can quickly become register limited. One solution is to use the PUSH and POP instructions to save and restore registers in the middle of an inner-loop, but this makes for crude and slow code.

---

## 2.9 Wideband's Enhanced Striding

---

Essentially, all of the previously discussed considerations are implemented to handle cases where the strides of called vector routines are unequal to one another. If all strides were equal every time a routine was called, the programmer could assign index registers **IR0** or **IR1** the stride length, as we could be done with all this complication. However, there are instances where a user wishes to implement stride arguments that vary for each array. These issues must be considered when writing an efficient algorithm.

In response to these trade-offs, the Wideband Optimized Math Library has been coded using *enhanced striding*. The routines are coded in such a way that the user is not inordinately penalized (in terms of routine execution time) for supplying striding arguments

in the call to a routine which not equal to one another. The routines analyze the stride arguments supplied from the calling sequence and determine the equality of number assigned to the stride sequences. The routine will then branch, depending on the results of this comparison, to a section of code optimized a particular series of striding arguments. In this manner, the inner-loop of a **vadd** routine called as **vadd (a, 1, b, 1, c, 1, 15)** will execute in the same amount of time as **vadd (a, 1, b, 2, c, 1, 15)** even though the strides arguments supplied to the routine are different.

---

## 2.10 Other Implementations

---

Other vector libraries offered by competitors also offer a striding mechanism. However, the typical solution offered is an implementation which checks for total equality of strides within the argument list. If all strides are equal, the logic will dictate a branch to an optimized section of code which will implement efficient code using one or two index registers. If one of the stride arguments is not equal to the others, the code will branch to a worst case code section where each stride is handled by a register who's pointer must be updated at the start or end of the loop to indicate the next memory location.

The problem with this approach is that the user is given only two solution: a best cases scenario where all strides are equal and a worst case which covers a scenario where all strides are dissimilar. The user will pay a performance penalty for using arguments which are not equal, as the worst case solution is not optimized for only slightly degraded (not perfectly symmetrical) cases. For instance, in the vector routine **vaam**, two vectors are added together, another two vectors are added together, with the sums being multiplied by one another to produce the final product, vector **e**. Since 5 vectors are involved in this routine, 2 of them will use index registers **IR0** and **IR1** to handle their stride values. The other 3 vectors will have to use a register such as **R0**, **R1**, **R2**, etc. to hold each of the remaining stride values provided by the arguments supplied to the routine. Each of these registers will contain a value which will be added to the current memory locations (at the end of the inner- loop of a routine) in order to point to the next memory location (value) which will be operated upon. In the case of the **vaam** routine, this updating costs 3 instructions per loop for a worst case scenario This is the penalty the user pays for supplying any argument stride series where only one value is dissimilar to any of the others.

Our solution to this problem differs in that we have implemented code which takes into account multiple stride scenarios which results in optimized solutions. We have implemented cases where all stride are equal, and case where all strides are unequal (the worst case in performance terms), and many cases in between. The result is that you may supply stride arguments to the routines which are unequal (in all kinds of configurations) and be assured that you are not being penalized for doing so.

The trade-off made in this case is that the code slightly larger than a routine which would handle only a best and worst case. Also, the user will note that the entry overhead associated with the execution timing is slightly increased (at most 10 to 15 cycles in some of the larger routines) due to the necessity of checking stride values as they are supplied from the calling routine. Note that this is done only once at the start, during the

entire execution of the routine. In both cases this is a very minor price to pay considering the flexibility of the software and benefits of improving execution time.

---

## 2.11 Introduction To The Trigonometric Functions

---

Trigonometric functions are used throughout signal processing, engineering and scientific applications. Indeed, it is often impossible to solve a scientific or engineering problems without utilizing a trigonometric relationship of some type. However, due to the need for accuracy and speed, what once began thousands of years ago as the simple calculation of the ratios of the sides of a right triangle has evolved in the modern era of computing into a fine art form.

Trigonometric functions are historically defined as mathematical expressions representing the ratios of the sides of a right triangle. From the definition of the sides of a right triangle, the trigonometric functions representing the acute angles were defined: sine, cosine, tangent, cosecant, secant, and cotangent. With the advent of the concept of the inverse function, the hyperbolic functions and the power functions (log, power, log2, log 10,  $\exp^{10}$ ,  $\exp^2$ , etc.) the number of functions expressing the periodic functions quickly expanded.

There are multiple ways to compute these relationships. Most lay persons are used to conceiving of these computations as little more than function buttons on a scientific calculator. The more mathematically sophisticated users, such as an average engineer, has had enough exposure to calculus to know that the trigonometric functions can be computed using a power series. For instance, in calculus, we learn that the sin function can be computed as a Maclaurin series expansion as follows:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots \text{ where } (|x| < \infty)$$

For a computer algorithm, this is a poor method to calculate the sine function as the algorithm will converge slowly on the answer. Also, the accuracy of the function is dependent on the number of terms in the polynomial, with a greater number of terms resulting in a more accurate answer.

---

## 2.12 Introduction To The Approximation Process

---

Fortunately, the advent of digital computers in the later 1940s stimulated research into the problem of computing these function using numerical approximations. In the late 1950s and the early 1960s much research was conducted at the Los Alamos and Argonne National Laboratories, and at the National Bureau of Standards into the methodologies for numerical approximation of these functions. The results of these research studies was the development of a series of approximations, which are, in reality, simple n degree polynomial equations with coefficients for each factor which are used to compute, to a given degree of accuracy, an approximation of a particular function.

These approximations are polynomials which mimic, if you will, the mathematical behavior of the function over a defined input range, called the **domain**. Each approxi-

mation has a number of coefficients associated with it. Generally, the greater the degree of accuracy required by the user, the larger the number of coefficients, and the longer the algorithm will take to execute. For example, the sine function can be approximated using the following rational approximation created by Carlson and Goldstein at Los Almos Laboratories in 1955:

$$\sin x = x \left( 1 + a_2 x^2 + a_4 x^4 + a_6 x^6 + a_8 x^8 + a_{10} x^{10} \right) \quad \text{where } 0 \leq x \leq \frac{\pi}{2}$$

**TABLE 4**

**Sine Function Approximation Coefficients**

Sine Approximation Coefficients
$a_2 = -0.49999999963$
$a_4 = 0.0416666418$
$a_6 = -0.0013888397$
$a_8 = 0.0000027526$
$a_{10} = -0.000000239$

where the answer is accurate to  $\pm 2 \times 10^{-9}$

Note that the function has a limited domain (e.g., a range of valid arguments you can supply to the function) and a limited accuracy ( $\pm 2 \times 10^{-9}$ ).

### 2.13 Important Relationships When Computing Trigonometric Approximations

---

Although this exact approximation is not utilized in the Wideband library, it is useful for demonstrating a number of salient points regarding the computation of trigonometric function:

- The accuracy of the final answer produced by an approximation is always dependent upon the approximation chosen. Since most libraries never disclose the exact polynomial used to compute the function, you should pay special attention to the benchmark accuracy claims provided by the developers. In our case, the benchmarks are conservative. The Wideband library maintains 7.75 decimal digits of precision throughout its calculations.
- The speed at which the function computes the final answer is a function of the number of coefficients and the number of degrees of the polynomial used to approximate the function. Generally speaking, the greater the degree of the polynomial used and the greater the number of coefficients, the slower the routine will execute, but the more accurate the answer will be. The converse also holds true: generally speaking, the smaller the number of coefficients the faster the routine will execute and the less accurate the routine's answer will be. Beware of vendors who promise both speed and accuracy without suggesting that one affects the other. In the case of the Wideband routines, when presented with a scenario where speed must be balanced against accuracy, we have opted for accuracy.
- All trigonometric approximations must be chosen such that the approximation used to implement the function is fitted to the processor architecture. For instance, in the

case of the Wideband trigonometric functions, we have chosen approximations whose coefficient ranges do not exceed the ability of the processor to represent them. This is analogous to saying that we have not chosen approximations which are designed for 64-bit machines (like those running at National Laboratories) which might use coefficients whose fine accuracy might not be representable in a 32-bit machine such as the TMS320 series machines. Further, we have chosen approximations which return the maximum amount of accuracy for a 32-bit processor and yet do not waste precious inner-loop cycles that would be necessary to support 48 or 64-bit precision. Great research efforts went into choosing an algorithm that matches the processor's precision capabilities.

- This concept was further extended in choosing approximation algorithms that avoid the weaknesses of a specific class of processors. For example, CISC processors such as the x86 or 68000 series may easily implement microcoded algorithms which implement a division, square root, or reciprocal process.

The TMS320, however, is a RISC processor which lacks a native microcoded instruction to compute a reciprocal. Instead, the TMS320 uses either a lookup table to approximate a reciprocal or a square root to 8-bit precision and later process it through a Newton-Raphson algorithm to gain full 32-bit accuracy (the TMS320C4x); or in the case of the TMS320C3x family, depend on the programmer to provide the initial 8-bit approximation.

This translates into expensive processor timing that should be avoided if possible. The Wideband library attempts to avoid this through the judicious use of approximations that avoid the division process (when possible) and instead substitutes multiplication by reciprocal values. Further, when cases utilizing the reciprocal operation have to be implemented within code, special techniques are used to implement ensure maximum speed and accuracy. In effect, the library is coded in such a manner as to play into the strong suit of the TMS320 processor, which is multiplication, addition and accumulation.

## 2.14 Reduction of Arguments Supplied To Trigonometric Functions

---

The largest single source of error typically encountered when implementing a trigonometric function in code is the reduction process used to reduce the original argument supplied to the routine. For instance, the sine routine mentioned earlier uses a reduction process to make sure that an argument supplied to the core portion of the inner-loop of a routine first has its argument reduced to fall between 0 and  $\pi/2$ . This is because the approximation process is valid only within a certain domain. All incoming argument supplied to the routine must be reduced to a period within the unit circle ( $2 \times \pi = 6.283185307\dots$ ), after which the argument must be reduced so that it lies between 0 and  $\pi/2$  (0 to 1.570796326...).

This reduction process can become quite complicated within the coded implementation. In many cases, if the reduction logic is slightly flawed, the results returned from the routine will be inaccurate. This is especially true if arguments supplied to a function are close to the boundaries of the domain of the function. For instance, if the argument

0.000001 were supplied to the sine routine, a poorly implemented function would return an incorrect answer.

The Wideband library uses very careful reduction techniques which result in functions which produce accurate answers to 7.75 digits until the domain boundary is encountered.

---

## 2.15 Complex Number Representation

---

Complex numbers are used throughout the scientific and engineering community. A complex number  $c$  can be described by the equation:

$$c = a + ib$$

where  $a$  and  $b$  are real numbers and  $i = \sqrt{-1}$

Complex numbers are atomic, but we can derive real-valued functions as follows:

$a = Re\{c\}$ , which derives the real part of  $c$  and

$b = Im\{c\}$ , which derives the imaginary part of  $c$ .

Note that both  $a$  and  $b$  are real numbers.

When a computer processes problems utilizing complex data, it is necessary to understand the manner in which complex data is accessed and processed. Typical operations involving vectors utilize arrays, each element of which contains a floating-point number or integer which represents a data value. The same concept hold true for complex numbers, except that the arrays must hold both real and imaginary numbers in order to represent complex numbers. This is done by interleaving both the real and imaginary portions of the data. In effect, the array becomes twice as long as a real-valued array as both a real and imaginary portion of data must be accounted for. This can be represented as follows:

---

**TABLE 5 Real Number Versus Complex Number Representation**

Index Of Array	Real Array	Complex Array
0	1st Real Number	1st Number - Real Portion
1	2nd Real Number	1st Number - Imaginary Portion
2	3rd Real Number	2nd Number - Real Portion
3	4th Real Number	2nd - Imaginary Portion

Note that striding concepts discussed earlier in the “Enhanced Striding” section also hold true complex data. For example, if you provide a stride of one as an argument to a routine, each iteration of a routine will result in striding over both the real and imaginary portions each iteration of the main loop. This what we meant by the term *atomic*: each complex number, although basically composed of two floating-point numbers in

memory, is treated as one number for computing purposes. However, a stride of 1 typically results in 2 numbers being produced each iteration of the inner loop: a real number and an imaginary number.

When discussing complex numbers in the equations section of this manual, we composed a method of conveying to the user the exact nature of the algorithm that was being examined without unnecessary confusing detail regarding the indices of the vectors. We chose to describe the real and imaginary portions of a complex number using the **Re** and **Im** nomenclature. For instance, let us suppose you wish to add two complex numbers together to produce a resultant third complex number. The routine that you would call to do this is **cvadd**. If you look up **cvadd** in the complex section, you'll see the following equation:

$$Re\{C_{mk}\} = Re\{A_{mi}\} + Re\{B_{mj}\} \quad m = \{0, 1, 2, \dots, n-1\}$$

$$Im\{C_{mk}\} = Im\{A_{mi}\} + Im\{B_{mj}\}$$

The first line of the equation states that the real portion (*Re*) of resultant complex output vector **c** is equal to the addition of the real portion (*Re*) of input complex vector **a** and the real portion (*Re*) of input complex vector **b**. **Re** always represent the real portion of the complex number.

The second line of the equation states that the imaginary portion (*Im*) of resultant complex output vector **c** is equal to the addition of the imaginary portion (*Im*) of input complex vector **a** and the imaginary portion (*Im*) of input complex vector **b**. **Im** always represent the imaginary portion of the complex number.

---

## 2.16 Setting Test Code Array Size & Data Type **complex**

---

Wideband has defined a data type called **complex** in its assembly language routines which defines a complex number as composed of 2 contiguous pieces of data (real and imaginary), located in adjacent locations in memory. Since the **complex** data type has two values (real and imaginary) for every memory location normally reserved by the **n** argument (the count of elements), a mechanism had to be invented to resolve the proper definition of memory allocation for arrays of varying types.

If you examine the example code provide with each routine, you'll find that the routines associated with complex number have an array size indicated by a construct called **NCOMPLEX**. The define statement is always set to the number of elements you wish to process. However, it is typically multiplied by a number which ensures that the conflict between defining a complex number as having 2 pieces and defining **n** as representing the number of elements you wish to process is properly resolved.

For instance, the following routine test routine is for **cvrsub**, where a floating-point value in real input vector **b** is subtracted from the real component (*Re*) of complex input vector **a**. The results are stored in complex output vector **c**. Note that **NCOMPLEX** is assigned the value of 15, which is the value passed through the argument **n**.

However, complex input vector **a**, by definition, has 30 values, and not 15. Complex input vector **a**'s size must therefore double in order to handle both the real and imaginary components specified initially by the element count argument **n**. This is accomplished by multiplying **NCOMPLEX** by 2 in order to double its size. Note that input vector **b** is of type real and therefore does not need to be doubled in order to stride properly.

```
#define NCOMPLEX 15

float a[2*NCOMPLEX] = { -1.4, 1.3, 2.1, 3.3, 1.2, 8.4,
                        3.9, 8.3, 4.6, -8.2, 1.7, 4.9,
                        3.6, 4.8, 6.4, 2.2, 3.9, 1.2,
                        3.3, 4.3, 2.3, 3.4, 3.6, 6.4,
                        2.3, 4.2, 6.3, 2.3, 1.2, 3.7
                        };

float b[1*NCOMPLEX] = { -4.3, 4.3, 9.5, 4.3, 1.2, 9.4,
                        4.3, 4.7, 6.4, -2.3, 5.6, 2.8,
                        6.5, 4.3, 8.4, };

float c[2*NCOMPLEX];

main ()

{
cvrsub ( a, 1, b, 1, c, 1, NCOMPLEX );
}
```

In summary, here are the rules which you should follow:

- When specifying the element count **n**, think in terms of floating-point numbers. If your vector is real, **n** will be translated to **m**, and will reflect the number of data points you wish to process. This will be accomplished automatically.
- When working with complex numbers, think of each complex number as one number which internally consists of a real and an imaginary portion. You do not need to double the number **n** when specifying the routine in your code as the assembly routine automatically does this for you. You only need to be concerned about this in your test code.
- When you reach the point of supplying test data to a complex routine, you must consider the size of the arrays as an important consideration. The array size for complex types will always be twice the size of the real types, and you must compensate, as we have shown in the **cvrsub** routine by doubling the argument **n**.
- The complex type data structure is atomic. Adjacent memory location contain a real and an imaginary portion, which are considered to be one complex number.

## 2.17 Execution Timing

---

The routines within the Wideband library have been designed to execute as fast as we could efficiently code them. Since many users are often running processes which are time dependent, they need to know exactly how long it takes for a given routine to execute. Timing benchmarks are included in the library. This section explains how we derived the timing benchmarks so that no ambiguity exists as to execution speed of the library.

The speed with which a routine will execute is dependent upon the following:

- The language the algorithm is coded in - the Wideband library routines are coded in TMS320 assembler, as assembler code is usually more efficient ( for coding a detailed numerically intensive DSP algorithm) then straight C code.
- The degree to which the programmer utilized all capabilities of the processors assembly language - the Wideband library goes to great lengths to utilize parallel instruction whenever possible.
- The implementation of the algorithm. In many cases the algorithms are very simple and easy to implement. In these cases the inner-loop of the routine is generally short, and there isn't much programming skill utilized to implement the algorithm. An example of a very simple algorithm is vector absolute value routine, where the inner-loop is very short. More complex algorithm such as those implemented in the complex and trigonometric algorithms are much more difficult to implement. In the case of the complex functions, complex data types are used, where the programmer must implement sufficient skill to handle both the real and imaginary portion of the complex number. In the case of the trigonometric functions, considerable math skills must be utilized along with various insights into the coding process to guarantee that the arguments are properly reduced, the correct approximation chosen, and a sufficient degree of accuracy is obtained.
- Execution speed is dependent on the clock cycle the processor utilizes. Obviously, a processor utilizing a 50 MHz clock is going to execute code faster than a processor using a 40 MHz clock.
- Execution speed is also dependent on where you place your programmed data and what type of memory you are using. Some systems utilize DRAM memory. Processors attached to such memory will have difficulty achieving the performance associated with fast memory such as SRAM. Be sure to observe the type of memory your board currently uses and make software changes as necessary. Programmers using SRAM or DRAM should be sure to consult the Texas Instruments Linker Manual to be sure to link the most frequently used Wideband Routines, along with user data is linked into the on-chip memory. If the user exceeds the amount of on-chip zero-wait state memory, link the remaining programs to zero-wait state on-board memory offered in typical SRAM hardware configurations.

## 2.18 A Word of Caution On Theoretical Execution Timing

---

The user should be advised to use caution when calculating the theoretical execution speeds using the global processor clock as the execution standard. The TMS320C3x

processor uses an instruction set where all instructions are specified as single word, 32-bit machine long. The User's Manual advises that in most cases (outside of repeat block and branching instructions) one instruction is executed every clock cycle. This can lead to mistaken performance expectations. A user may mistakenly believe, for instance, that a 40 MHz version of TMS320C3x processor will execute 40 million instructions per second, as all instructions are single word length and execute in one clock cycle. Unfortunately, this isn't quite the case.

In ideal conditions, the user can realistically expect a 20 million instructions per second (or half the clock processor clock speed) execution figure as the limiting factor in speed analysis turns out to be the speed with which an instruction can be executed. In the case of a 40MHz machine, for example, the instruction execution speed is 50 nanoseconds which means that under ideal conditions (no bus contention, full pipelines, data and instructions in the on-board cache, maximum use of parallel instructions, etc.) the processor will execute 20 million instructions per seconds, or 20 MIPS. We arrive at this figure as follows:

$$\frac{50 \text{ nanoseconds}}{1000000000} = 0.00000005 \text{ seconds / instruction}$$

$$\frac{1}{0.00000005 \text{ seconds/instruction}} = 20000000 \text{ instructions/second}$$

The Texas Instruments clocking scheme executes one assembly language instruction every two clock cycles. This is equivalent to having a global clock which executes at the advertised processor speed, and an instruction clock which runs half as slow as the global clock. So, when the statement is made that one instruction is executed every clock cycle, one is typically referring to the global clock and not the instruction execution clock. Realistically, the TMS320 family will execute 1 assembly language instruction every 2 global clock cycles.

For instance, if you are using a 40 MHz TMS320C3x processor, you should expect to realistically obtain 20 Million Instructions Per Seconds (MIPS) true execution speed operating in near ideal code execution conditions.

The following table summarizes currently available processor configurations:

**TABLE 6**

**Summary of Processor Types and Theoretical Performance Limits**

Processor Type	Clock Speed	Instruction Speed	Theoretical MIPS
TMS320C3x/4x	27 MHz	74-ns	13.5 MIPS
TMS320C3x/4x	33 MHz	60-ns	16.7 MIPS
TMS320C3x/4x	40 MHz	50-ns	20 MIPS

## 2.19 How Timing Benchmarks Are Reported for Vector Routines

---

Given all these consideration, the technical staff at Wideband decided to report performance benchmarks based on counting the exact number of instructions internal to each routine. Reporting the number of instruction per inner-loop plus the setup times for entry into and exit from a routine is less ambiguous than reporting cycles per second. In this manner, a user has a benchmark that is valid for all configurations. If a user knows that a routine will be executed by a 40 MHz processor instead of a 50 MHz processor valid planning for execution timing can be made.

Therefore, for every routine, we report two numbers as benchmarks: an **overhead number** and an **inner-loop number**. For instance, a typical timing benchmark looks as follows:

**EXECUTION TIME**  $23 + 2*N$  cycles

- The first number (in this case 23) is the total number of instructions needed to enter the routine and exit the routine. This is called the **overhead number**. This includes the entry setup time to load values off of the stack or registers, save any necessary registers, traverse the intelligent-striding code to branch to the most optimized portion of code, and the 4 cycles necessary to setup the repeat block counter. This entry overhead number is added to the exit overhead number, which is derived from the total number of instructions used to flush any parallel instruction pipeline, makes any final storage of answers, branch to any exit points, restoration of any saved registers, and a return to the calling routine. Note that the average number represents an expense, measured in cycles that is incurred only once during the call to the routine. Also note that the number of cycles reported are measured in global clock cycles. To obtain the theoretical ideal execution speed you should multiply this number by 2, as assembly instructions are realistically executed at 1/2 the speed of the global clock.
- The second number (in this case 2) indicates the size of the inner-loop of the routine. This is called the **inner-loop number**. This parameter indicates that a computation will take place every 2 clock cycles. To derive the number of clock cycles necessary to execute your problem, multiply the number of data items you desire to process (called **n**) times the number of cycles resident in the inner-loop. Note that the inner-loop cycle cost is incurred each time the a traversal of the inner-loop is made. Therefore, this parameter represents the bulk of the processing time a processor will expend upon a problem. Also note that the number of cycles reported are measured in global clock cycles. To obtain the theoretical ideal execution speed you should multiply this parameter by 2, as assembly instructions are realistically executed at 1/2 the speed of the global clock.

## 2.20 Scalar Timing Benchmarks

---

Execution timing benchmarks for the scalar routines represent the number of processor cycles necessary to compute the final answer for each element that is to be processed. The scalar routines operate on one argument only.

A typical timing benchmark looks as follows:

**EXECUTION TIME** 17 to 40 cycles register passing, 14 to 43 cycles stack passing, 9 to 6 for cycles for domain error

Note that this category may contain 3 types of numbers.

A parameter often reported as "Error" may be defined as the number of cycles necessary for the routine to determine a domain error in the user supplied argument. For instance, the log function has a domain of 0.0 to  $3.4 E^{38}$ . If the user supplies the routine with -4, the routine will detect and respond with a domain error within 7 to 9 cycles.

Also reported within the "Execution Time" category are the number of cycles necessary to compute final answer for both stack and register passing argument model. Note that generally, the register passing model will execute a few cycles faster than the stack argument model.

Note that the timing parameters supplied within the actual functional descriptions are, in some case, more detailed than those supplied with the timing tables. For instance, the hyperbolic tangent, **tanh**, has multiple intermediate timing parameters supplied within the functional description, while the timing tables discloses just a start and endpoint for the entire range. The multiple intermediate points supplied within the actual function descriptions are benchmarks for execution timing based on the range of the argument supplied.

If, for instance, you supply an argument to a function that is within the specified domain, but falls on one of the extreme points of the domain, the Wideband algorithms will probably revert to lengthy code to range reduce the arguments before processing the approximations. You will then incur the maximum execution time specified.

However, if you supply a Wideband Library routine with an argument that falls within a range that does not require lengthy range reduction, the routine will execute faster. The lengthy numbers reported within the functional descriptions represent the various combinations of range processing that the algorithm may resort to when processing your arguments. In most cases, assume either the worst case execution time or the best case execution time.

## 2.21 Embedding Wideband Routines In PROMs, EPROMs and EEPROMs

---

All Wideband routines are individually linkable and embeddable. This leads to two important implications:

- All routines are individually linkable. If you are an embedded programmer, you may embed a series of routines within a PROM without having to embed the entire library. This makes sense as most programmers typically utilize only a small amount of the available Wideband routines within their code. Valuable memory space is thus conserved.
- There are no inter-routine dependencies within the Optimized Math Library library code. Each routine stands alone, without the need to link in other supporting routines

in order to make the routine work. For instance, in a function such as the root mean square, there is no need to link in a division and square root routine as separate entities as they are already included in the native code.

---

## 2.22 How Setup and Execution Timing Tables Were Derived

---

Chapter 6 documents the routine names, routine definitions, the (entry and exit) overhead, and the number of cycles per element. The cycle and overhead counts were arrived at by examining the assembler code itself and counting the number of instructions necessary to enter and exit the code, and the number of instructions situated in the inner-loop. Most TMS320 family assembler instructions are executed in one processor cycle (global clock). There are, however, some instructions such as the RPTB instruction or conditional branches which take up to 4 cycles to execute. These issues were considered when tabulating the code count for the timing tables, and numbers are added accordingly. We have tried to make the timing specifications conservative and fair.

---

## 2.23 Element Count **m** For The Corrected Number of Executable Iterations

---

The statement  $m = \{0, 1, 2, \dots, n - 1\}$  shown in the complex add example is a way of stating the number of iterations that you wish the routine to process. **n** is always an integer and represents the number of elements you wish the routine to process. This number should represent the number of inner-loop iterations that you expect the process to execute. **n** is always decremented by 1 value internally within each routine. This is because of the manner in which the repeat block counter works internal to the Texas Instruments microprocessor. This is also convenient for C language programs which implements a "zero-origin" counting system. For example, if you are thinking of adding 2 vectors, both consisting of arrays which have 100 numbers each, you should call your routine with **n** = 100. While you are probably mentally conceiving your algorithms executing 1,2,3,4,5...99,100 it is, in reality, actually executing as 0,1,2,3,4...99.

**m** stands for the corrected number of iterations that you plan on having the routine execute. **m** also is used to indicate which memory position is currently being pointed to within an array.

