

CHAPTER 4

Using the Wideband TMS320C3x/4x Optimized Math Library

4.1 Getting Started

The latest changes and updates information that may be of importance to you is provided in the README.DOC file. Please read this file before installing the software. To do so, place the Optimized Math Library software diskette into drive a of your personal computer and type:

TYPE A:README.DOC | MORE

Alternately you may print the material if you have a printer attached to your computer. Type the following command:

PRINT A:README.DOC

Late breaking changes are included in the REAME.DOC file.

4.2 Calling An Optimized Routine Within Your C Code

We assume that you have located the routine that you wish to link to your C code. To include one of the Wideband Optimized Math Library routines in your code, simply embed a call to the routine in your C code. For example, the following code contains a call to the Vector Add function, called **vadd** in the Optimized Math Library. A call to a scalar routine is the same except that the arguments are different.

```
#include <stdio.h>
#include "vecpac.h"

main()
{
float a[16], b[16], c[16];
int i;
    vadd(a,1,b,1,c,1);
}
```

The call to the **vadd** routine is contained within the line **vadd(a,1,b,1,c,1);**. The input to the computation are provided in vectors **a** and **b**. The resulting values are returned in output vector **c**.

The function prototypes for all routines in the Optimized Math Library are provided in the **wcilib.h** include file. You must include this file or build your own include file to prototype the functions for your application.

Associated with each Optimized Math Library routine is an example file which is prefixed with the letter **t**. These files, found under the examples directory, provide an example for using each routine. For instance, the file **tvadd.c**, found in the examples directory, is a simple C program which exercises the **vadd.obj** object code with test data. You can therefore test each routine before you use it to understand its operation.

4.3 Compiling Optimized Math Library Routines With Your C Code

There are no special considerations when compiling C-language applications code calls to Optimized Math Library routines. You must select an appropriate memory model and argument passing convention and use this selection for all code within an application. If you select the register argument passing convention you should ensure the function prototype arguments are available to the compiler as the routines are processed. For further information see section 2.2.2 on page 2-16 and section 2.3.2 on page 2-20 of the *TMS320 Floating-Point DSP Optimizing C Compiler User's Guide*. Finally, use the **-o** optimizing option when compiling with the register passing convention, as the optimizer model exploits this convention extensively.

4.4 Linking Optimized Math Library Routines With Your C Code

The Texas Instruments linker is based on the industry standard **Common Object File Format (COFF)** originally created for the Unix environment. The COFF format handles the relocation of compiled code to physical memory. This means that you will have to choose a memory model before linking. The standard default memory model is the small memory model. This model is generally adequate for most applications. Applications that include large data sections may want to use the big memory model. You must not mix memory models by using Optimized Math Library routines or your own routines that include object code compiled with both the small and big memory models. Also, if you include the TI run-time libraries, make sure the libraries are compiled with the correct memory model before linking.

When linking your program to a Optimized Math Library routine you will have to specify to the linker where the object modules (including the Optimized Math Library object routines) reside on your system. To use the compiler and linker together you may construct a command files which will compile and link the routines. Chapter 8 of the *TMS320 Floating-Point DSP Assembly Language Tools User's Guide* contains a detailed explanation of the linker and the command files associated with it. Also, Chapter 2 of the *TMS320 Floating-Point DSP Optimizing C Compiler User's Guide* provides additional information for linking.

4.5 Calling Conventions

The Texas Instruments C compiler has a number of user-selectable conventions which allow an operator to choose between specific memory utilization techniques and argument passing strategies. These choices allow the operator to make operational trade-offs between speed of execution versus memory size and alignment, and speed of execution versus how arguments are passed to the processor's registers.

The Optimized Math Library supports all four (4) possible memory models by supplying four versions of the library installed in the default directories defined in paragraph 3.2

4.6 Small Memory Model

The small memory model is a convention defined for the Texas Instruments TMS320C30 and TMS320C40 compiler and linker. Since it is the default option for the compiler, the use of it declares to the compiler that it can expect all external variables, static variables, and compiler-generated constants to reside (after linking) within a single 64K word long data page (65,536 words) memory section. The compiler can therefore access these global and static data objects without modifying the data page pointer (DP) register.

Use the small memory model whenever possible. The small model is more efficient than the big memory model as the compiler does not have to reload the data page pointer (DP) using additional LDP instructions before accessing global and static data.

If you wish to run the small memory model version of the Optimized Math Library routines, include the library from one of the two default directories where Optimized Math Library was installed depending on the argument passing model used:

- `\WC\LIB\SSP` - Small memory model, stack passing
- `\WC\LIB\SRP` - Small memory model, register passing

Once again, do not mix memory models, nor memory models with varying argument passing conventions in your code. For example, If you start your code using the "small memory model, register passing," routines, you may not combine these routines with portions from the "big memory model, register passing" or the "small memory model, stack passing" sections.

Make sure that your uninitialized data (COFF `.bss` section) is aligned within a 64K word memory boundary and does not exceed that boundary. To do this, use the `.SECTIONS` assembler directive discussed in Chapter 4 of the *TMS320 Floating-Point Assembly Language Tools User's Manual*. Also see the discussion of the `block` directive found in Chapter 8 of the *TMS320 Floating-Point Optimizing C Compiler User's Guide*. Note that neither the compiler nor the linker will warn you if you exceed this boundary limitation.

If you have declared variables outside of the COFF `.bss` section, you can still use the small memory model but you must access the variables indirectly. See section 4.5 of the *TMS320 Floating-Point Optimizing C Compiler User's Guide* for further information.

If you are using large arrays of data you may circumvent the 64K word memory limits and still run the more efficient small memory model by setting the `-heap` option in the `.system` section and using the `malloc`, `realloc` or `calloc` functions dynamically allocates the data, allowing it to be indirectly addressed. This will allow you to run static and global data in a program space that exceeds the 64K boundary defined by this compiler option. See Chapters 2, 4 and 5 of the *TMS320 Floating-Point Optimizing C Compiler User's Guide* for further information.

4.7 Big Memory Model

The big memory model is a convention defined for the Texas Instruments TMS320C30 and TMS320C40 compiler and linker which does not restrict the size of the COFF `.bss` section, thus allowing unrestricted memory space for global and static data. The use of this compile option declares to the compiler that external variables, static variables, and compiler-generated constants can reside (after linking) outside a single 64K word long data page (65,536 words) memory section. To access this data, however, the compiler must therefore access these global and static data objects by having the data page pointer (DP) identify the memory page where the data object is stored and then access it.

The big model is less efficient than the small memory model as the compiler has to reload the data page pointer (DP) using the LDP or LDPK instructions before accessing global and static data. This can cost up to 2 instruction cycles within the TMS320 architecture.

If you wish to run the big memory model using Optimized Math Library routines, include the library from one of the two default directories where Optimized Math Library was installed depending on the argument passing model used:

- `\WCI\LIB\BSP` - Big memory model, stack passing
- `\WCI\LIB\BRP` - Big memory model, register passing

When you compile C code to use the big memory model the `-mb` option of the compiler is used.

4.8 Stack Passing Model

The stack passing model is a convention defined for the Texas Instruments TMS320C30 compiler which declares the method that arguments are passed to functions. The use of the stack passing option declares to the compiler that you wish to pass arguments to the called function via a stack.

The stack passing model is slightly less efficient than the register passing argument model since the compiler has to access argument addresses from the stack before data can be used by the routine.

If you wish to run the stack passing model using Optimized Math Library routines, include the library from one of the two default directories where Optimized Math Library was installed depending on the argument passing model used:

- `\WCI\OPTMATH\SSP` - Small memory model, stack passing
- `\WCI\OPTMATH\BSP` - Big memory model, stack passing

The stack passing model is the default option for the argument passing convention on the Texas Instruments C compiler. When you compile C code to use the stack passing convention no options are required.

You cannot mix register and stack passing routines in the same C code application. For instance, if you specify a small memory model, stack passing routine you cannot later use a small memory model, register passing routine in the same code. This will cause errors.

If you are using a real-time DSP operating system or micro-kernel in your development process, the Optimized Math Library is compatible with most of these environments. You should note the vendors convention when selecting which routines to use from the Optimized Math Library.

4.9 Register Passing Model

The register passing model is a convention defined for the Texas Instruments TMS320C30 compiler which declares the method that arguments are passed to the called function. The use of the register passing option declares to the compiler that you wish to pass arguments to the called function via registers as available.

Note the following regarding the register passing model:

The register passing model is slightly more efficient than the stack passing model as the compiler does not have to save (push) and restore (pop) extra registers before data can be loaded to the routine. Instead, values are loaded directly off of pre-determined registers directly into the called routine. This saves a few instructions per routine. Although this cost may be negligible over a large set of data, for those running one iteration of a routine this may be an important consideration. See pages 2-19 through 2-20 and pages 4-15 through 4-20 of the *TMS320 Floating-Point Optimizing C Compiler User's Guide* for further details.

If you wish to run the register passing model using Optimized Math Library routines, include the library from one of the two default directories where Optimized Math Library was installed depending on the argument passing model used:

- `\WCI\LIB\SRP` - Small memory model, register passing
- `\WCI\LIB\BRP` - Big memory model, register passing

The register passing model is not the default option for the argument passing convention on the Texas Instruments C compiler. When you compile your C code with a Wideband Optimized Math Library routine note the following:

The small memory model, register passing routines take the command line argument “**-mr**” to compile, as the register passing option needs to be specified.

The big memory model, register passing routines take the command line argument “**-mrb**” to compile, as both the big memory model and the register passing convention need to be specified.

Use the **-o** option when using the register passing model as this flag initializes optimizing logic which is complementary with the register-argument run time model. For more information on this see pages 2-20 through 2-22 of the *TMS320 Floating-Point Optimizing C Compiler User's Guide*.

If you use the register passing convention you must prototype your functions. All Wideband routines have prototyped functions. Each Optimized Math Library routine provides a synopsis in ANSI C notation as to the calling convention for each library routine. In addition, an include file `vecpac.h` is provided for function prototypes with ANSI C or non-ANSI compilations. For further information on register argument passing and prototyping see page 2-20 of the *TMS320 Floating-Point Optimizing C Compiler User's Guide*.

You cannot mix and match register and stack passing routines in the same C code. For instance, if you specify a small memory model, stack passing routine you cannot later use a small memory model, register passing routine in the same code. This will cause an error.

If you are using a real-time DSP operating system or micro-kernel in your development process, the Optimized Math Library is compatible with most of these environments. You should note the vendors convention when selecting which routines to use from the Optimized Math Library.

4.10 Overflow and Underflow Conditions

The TMS320C30 implements a dedicated status register called the **ST** to contain global information relating to the CPU processing state. Bit 4 reports of the status register reports floating-point underflow conditions, while bits 1 and 5 report floating-point overflow conditions. Pages 3-5 through 3-6 and 11-10 through 11-11 of the *TMS320C4x User's Guide* contain detailed discussions of condition flag. Pages 3-5 through 3-7 and of the *TMS320C3x User's Guide* contain detailed discussions of condition flag.

Each routine within the Optimized Math Library manual contains a domain field which discusses restrictions or combinations of conditions which may result in undefined results. In the majority of cases there are no restrictions on arguments that can be supplied to a Optimized Math Library routine but careful requires a review of the TMS320 processor limits. All Optimized Math Library computations are conducted in 32-bit single-precision mode, resulting in the ranges of precision described in the following paragraphs.

The largest positive number which the machine is capable of representing is 3.4028234×10^{38} . Any operation (such as multiplication, division, subtraction or addition) which results in a number larger than this will result in an overflow condition. In this scenario the overflow flag is set in the **ST** register and the result of the operation is set to the largest positive extended-precision value representable by the machine: 3.4028234×10^{38} .

The largest (most negative) negative number which the machine is capable of representing is $-3.4028236 \times 10^{38}$. Any operation (such as multiplication, division, subtraction or addition) which results in a number larger than this will result in an overflow condition. In this scenario the overflow flag is set in the **ST** register and the result of the operation is set to the largest negative (most negative) extended-precision value representable by the machine: $-3.4028236 \times 10^{38}$.

The smallest positive number which the machine is capable of representing is $5.8774717 \times 10^{-39}$. Any operation (such as multiplication, division, subtraction or addition) which results in a number smaller than this will result in an underflow condition. In this scenario the underflow flag is set in the **ST** register and the result of the operation is set to 0.0.

The smallest negative number which the machine is capable of representing is $-5.8774724 \times 10^{-39}$. Any operation (such as multiplication, division, subtraction or addi-

tion) which results in a number smaller than this will result in an underflow condition. In this scenario the underflow flag is set in the **ST** register and the result of the operation is set to 0.0.

Execution of a division algorithm on the TMS320C4x is done through the use of the reciprocal instruction, which performs an 8-bit initial seed. Overflow conditions which could be the product of dividing a very large number by a very small number will result in the setting of the overflow flag on the ST and setting the output number to 3.4028234×10^{38} . Division on the TMS320C3x requires the programmer to first approximate the 8-bit reciprocal in software, hence making for slightly longer execution times.

Execution of a square algorithm on the TMS320C4x is done through the use of the reciprocal square root instruction, which also performs an 8-bit initial seed. Computing the square root on the TMS320C3x requires the programmer to first approximate the 8-bit reciprocal square root in software, hence making for slightly longer execution times. An attempt to compute the square of a negative number will result in a processor halt with an error message. Attempted computation of the square root of zero will result in zero.

Chapter 4, pages 4-1 through 4-35 *TMS320C4x User's Guide* details all floating-point addition, multiplication, reciprocal and square root reciprocal logic.

Chapter 4, pages 4-1 through 4-24, and Chapter 11, pages 11-24 through 11-33 of the *TMS320C3x User's Guide* details all floating-point addition, multiplication, reciprocal and square root reciprocal logic.