MOTOROLA
Semiconductor Application Note

Order by AN1772/D
(Motorola Order Number)
Rev. 0 , 11/98

Efficient Compilation of Bit-Exact Applications for DSP563xx

Yuval Ronen
Emmanuel Roy
Lorraine McLuckie

Many of the standard algorithms in wireless and wireline communications, such as GSM speech coders and the G.723.1 and G.729a coders, use 16-bit, bit-exact C code and corresponding test vectors. These standard algorithms provided by the ITU/ETSI organizations employ ANSI C integer data types and implement 16-bit fractional arithmetic operations. To specify the fractional arithmetic model, which is foreign to the ANSI C language, the ANSI C code uses a set of subroutines that implement basic fractional operations (for example, addition with saturation, fractional multiplication, and fractional multiplication-accumulate).

An algorithm complying with the ITU/ETSI style compiles on any compiler that conforms with ANSI C. Thus, it theoretically requires little effort to compile the algorithm for any digital signal processor (DSP) or general-purpose processor for which an ANSI C compiler is available. In reality, an efficient implementation of the algorithm on a DSP requires some modifications to the C code before the DSP's C compiler can compile it effectively.

This application note describes the process of preparing standard 16-bit, bit-exact algorithms so that they compile for Motorola's DSP56300 family of processors, using the DSP56300 C compiler from TASKING, Inc.

Contents

1 The TASKING DSP56300 C Compiler

The techniques described in this document require that an application be compiled with the TASKING DSP56300 C compiler. Before a standard algorithm can compile efficiently for a DSP56300 family of processors, the TASKING compiler and tool chain must be installed. We recommend reading through the TASKING compiler and debugger user's manuals to become familiar with setting up a development environment under the TASKING integrated development environment, EDE, which is available on Windows-based systems. This document assumes familiarity with the C programming language, the TASKING compiler and debugger, and EDE.

On Windows-based systems, the first steps in preparing the C source code are: create an EDE project, import the C source files in the project, and set up the compilation and linking switches. On UNIX-based systems, the development process is based on makefiles, so the first step in preparing the C source code is to create a suitable makefile for the TASKING code-generation tools.

Terms and Acronyms

- *Standard algorithm* — a numeric algorithm that is part of a communications standard such as ITU/ETSI and is specified using a C program and test vectors.
- *Test vectors* — a set of numeric data that is fed to a program to verify its numerical conformance to the standard algorithm.
- *Bit exact algorithm* — a numeric algorithm for which correct functional behavior requires obtaining numerically identical results for a predefined set of test vectors.
- *Integrated development environment* — a software product running on a host computer that provides interfaces to the development tools (such as compiler, assembler, linker and debugger).
- *EDE* — the TASKING integrated development environment.
- *Function prototype* — an ANSI C construct that specifies the return type and the number of parameters and their types for the given function.
- *ITU/ETSI primitives* — a set of C subroutines, frequently used in ITU/ETSI standard algorithms, that implement fractional computations on variables with integer types. For example, the primitive mac( ) computes the multiply-accumulate function on three input variables, returning the accumulated result. In this application note, the term "fractional primitives" is used interchangeably with the term "ITU/ETSI primitives."
- *ADM* — Application Development Module, a development board that supports code development on DSP56301 devices.
- *ALU* — Arithmetic Logic Unit
- *ANSI* — American National Standards Institute
- *ETSI* — European Telecommunications Standards Institute
- *EVM* — Evaluation Module, a development board that supports code development on DSP56300 family devices
- *GSM* — Global System for Mobile Communications
- *IEEE* — Institute of Electrical and Electronic Engineers
- *ITU* — International Telecommunications Union

Consult the TASKING user's manuals for information on setting up an EDE project, using the compiler, and using options for invoking the linker. We briefly explain options cited in this document.

2 Development Flow

Setting up the application to run efficiently on a DSP56300 family processor requires these steps, which are discussed in detail in this application note:

1. Configuring the application code
2. Performing preliminary code transformations
3. Replacing integer variables with fractional variables
4. Inlining the primitives
5. Using optimized out-of-line primitives
6. Performing further optimizations

Additional Reading

Working through the code optimization process requires knowledge of several DSP56300-specific topics not explained in this application note. The following documents provide necessary information on these topics.

- *TASKING DSP56XXX C Cross-Compiler User's Guide*
- *TASKING DSP56XXX Cross Assembler User's Guide*
- *TASKING CrossView Debugger User's Guide*
- *Motorola DSP56300 Family Manual*
- Motorola DSP56300 device-specific user's manuals
- Harbison and Steele, *C: A Reference Manual*, Prentice Hall, 1995, Fourth Edition.

The topics to study in order to efficiently implement the code optimization process described in this application note are:

- Performing simulated Input and Output (simulated I/O) — Simulated I/O is necessary for reading test vectors into the compiled application as it runs on a development board (EVM or ADM) as well as for writing the results computed by the application out to a file. These operations are referred to as "simulated" I/O because they rely on the debugger to simulate the existence of files on the DSP system using files on the host system. Performing simulated I/O requires calling standard C library functions from the application code and invoking debugger commands to connect files on the host system to the data streams created by these standard C library functions.
- Measuring performance of the application — One way to measure the performance of the application, that is, the time the application requires to process a given amount of data, is to use the DSP563xx on-chip timers. Measuring the performance of various portions of the application can help to determine which portions require further optimization. Measurement can also help to gauge the effectiveness of the optimizations you apply.

Standard algorithms often come equipped with test vectors for verifying the correctness of the compiled application code. The sequence of steps proposed in this document enables you to run the verification procedure at various interim checkpoints. Performing verification at these checkpoints reduces the risk of introducing errors while preparing the application code.

The technique outlined in the sections that follow, combined with the Motorola header files included in this document, results in code that is portable between the DSP563xx environment and non-DSP development environments (such as a PC). The following sections detail the steps in the process. Each step has a defined purpose, a specific goal, and a sequence of actions for attaining the goal.

3 Configuring the Application Code

Setting up a convenient software development environment for conducting subsequent steps involves setting up the portable coding styles between the TASKING and the host compilers so that the code compiles and executes on both environments with all test vectors processed correctly. The resulting code serves as a baseline. In subsequent steps, as the code is transformed, this baseline is instrumental in isolating any programming errors. Steps in this technique are as follows:

1. Prepare a makefile or an EDE project.
2. Configure the code for the target DSP board.
3. Set up preliminary compilation switches.
4. Verify correctness – first time.
5. Include Motorola-specific header files.
6. Add Motorola-specific source files.
7. Set up the DSP operating modes.

3.1 Prepare a makefile or an EDE Project

When developing DSP code on a UNIX system, use the make utility to build the executable code from the source code. To that end, you must prepare a makefile. When developing DSP code on a Windows-based system, you can either choose to use make or to set up the software as a project under the EDE TASKING integrated development environment. This description of the compilation setup assumes the use of a makefile. It is straightforward to convert this setup into an equivalent EDE project configuration. You can also instruct EDE to accept a user-customized makefile. Typical makefile settings for the TASKING tools are:

```
# Select the Tasking C compiler steering program
CC=cc563
# Link the application by invoking the compiler. This is
# simpler than invoking the locator and the linker
# directly with explicit parameters.

LINK=cc563

# Choose an optimization mode out of the following list:
# Optimization disabled: -O0
# Optimization of code size: -O1
# Optimization of code size, debugging enabled: -O2
# Optimization of code speed: -O3
# Optimization of code speed, debugging enabled: -O4

OPTIMIZE = -O1
```

```
# Enable debugging using Tasking's Crossview debugger.
DEBUG = -g

# Combine the optimization and debug options to be passed to the

# compiler. –c specifies that each source file is compiled
# and assembled separately, leaving the link phase to be
# invoked separately.
# See explanation of the –M16 switch below.

CFLAGS = -c –M16 $(OPTIMIZE) $(DEBUG)
```

3.2 Configure the Code for the Target DSP Board

The TASKING compiler requires information on both the on-chip and off-chip memory available on the target DSP board; it uses this information during the linking and locating phases. The TASKING compiler comes equipped with definition files for the various Motorola EVM and ADM target boards. To select a target board out of the boards the TASKING compiler supports, use the compiler's option `-Wlc -dtargetboard.dsc` (where `targetboard` is the name of the board). For example, the option `–Wlc –d56301adm.dsc` directs the compiler to generate code suitable for executing on the DSP56301 ADM. When the software is built as an EDE project, the target board selection is set up in the Target Hardware tab in the "EDE|Linker options …" dialogue box.

3.3 Set Up Preliminary Compilation Switches

Processors of the DSP56300 family are 24-bit devices that support an optional 16-bit arithmetic mode. Portability of C code to these processors is greatly enhanced by running them in this mode. To set up the DSP to run in this mode and instruct the compiler to generate code for this mode, use the `–M16` switch of the TASKING compiler. This switch is set in the makefile or in the EDE project compiler options. Details on the DSP modes and on memory size limitations follow in Section 3.7, "Set Up the DSP Operating Modes," on page 6.

3.4 Verify Correctness – First Time

Once the preliminary compilation switches are set up, you can perform the first iteration of compiling and running the application on the DSP hardware. Even if no DSP-specific changes are made to the code, verifying that the application compiles and runs correctly is beneficial in uncovering any tools installation or makefile setup problems.

3.5 Include Motorola-Specific Header Files

Code modifications performed in subsequent steps use definitions and declarations in the Motorola-specific header files. These definitions and declarations include, for example, data types and macro definitions. Often, all application source files include common application-specific header files. Typically, one of these header files defines C function prototypes for the ITU/ETSI primitives. The recommended method for including the Motorola-specific header files is to include them in this header file using the C preprocessor `#include` directive.

The original standard algorithm code usually contains one header file in which the function prototypes for the ITU/ETSI primitives are defined, or two header files (one for the data type definitions, one for the prototypes). The Motorola-specific header files provide a replacement for these header files. Once they are inserted into the algorithm source code, the original function prototypes must be excluded. A convenient way to do this is to wrap them with `#if 0` and `#endif` C preprocessor directives.

Motorola provides the following two header files:

- `mottype.h` – defines the data types in the algorithm (e.g. Word16).
- `mathops.h` – defines function prototypes and various mappings of the ITU/ETSI primitives.

These files correspond to such files as `basicop.h` (in G.723.1) or `typedef.h` and `basic_op.h` (in G.729a).

For example, to exclude the original function prototypes from the `basic_op.h` file, the two lines denoted by `_DSP` below are added to the original file.

```
#if 0                                       /* _DSP */
Word16 add (Word16 var1, Word16 var2);      /* Short add,*/
Word16 sub (Word16 var1, Word16 var2);      /* Short sub, */
Word16 abs_s (Word16 var1);                 /* Short abs, 1 */

Word16 shl (Word16 var1, Word16 var2);      /* Short shift left, 1 */
Word16 shr (Word16 var1, Word16 var2);      /* Short shift right,1   */
Word16 mult (Word16 var1, Word16 var2);     /* Short mult, 1   */
…
Word16 norm_s (Word16 var1);                /* Short norm, 15  */
Word16 div_s (Word16 var1, Word16 var2); /* Short division, 18 */
Word16 norm_l (Word32 L_var1);              /* Long norm, 30  */
#endif                                      /* _DSP */
```

3.6 Add Motorola-Specific Source Files

Code modifications performed in subsequent steps use subroutines provided in these source files:

- `mathops.c` — provides optimized implementations of ITU/ETSI primitives that require several instructions to compute. This file should be added to the build process.
- `motutil.c` — provides routines to profile the execution of a few specific ITU/ETSI primitives in the algorithm. This file is for use during development only, as described later. Do not add it into the build process yet.

3.7 Set Up the DSP Operating Modes

Efficiently executing the bit-exact standards on the DSP requires setting up the DSP to run in the following operating modes:

- Arithmetic saturation.
- Twos complement arithmetic.
- 16-bit arithmetic mode (already set up in a previous step).
- 16-bit compatibility is optional. It results in more efficient code, though if the application program or data size is large (requiring more than 32K words), the processor must run in 24-bit addressing mode.

Setting up the processor to run in these modes requires:

- Setting up the compilation switches (in the makefile or in EDE) to use 16-bit arithmetic mode and possibly also 24-bit addressing. When invoked with the appropriate switches, the compiler links in start-up code that sets up these modes. To set up 16-bit arithmetic mode with 16-bit compatibility mode, use the compiler switch `-M16`. To set up 16-bit arithmetic mode with 24-bit addressing, use the compiler switch `-M1624`.
- Inserting assembly instructions to set up arithmetic saturation and twos-complement arithmetic modes on start-up.

Example:

In the following code example, the statements marked by ***_asm()*** are added to the application to set up arithmetic saturation and twos complement arithmetic modes. The code is compiled for 16-bit arithmetic mode, and 16-bit compatibility mode is turned on (-M16).

```
main(int argc, char *argv[])
{
    Word16  DataBuff[Frame] ;
    Word16  Line[26] ;
    …
    /* Lines below added to set up twos complement rounding and
    arithmetic saturation */
    _asm ("bclr #13,sr");     /* 16-compat mode off */
    _asm ("opt noopnop");     /* Prevent the assembly
                                 optimizer from removing NOPs */
    _asm ("nop");             /* Pipeline delay for mode change */
    _asm ("nop");
    _asm ("nop");
    _asm ("bset #21,sr");     /* Set Twos complement rounding on */
    _asm ("bset #20,sr");     /* Set Saturation Arithmetic mode on */
    _asm ("bset #13,sr");     /* 16-compat mode on */
    _asm ("nop");             /* Pipeline delay for mode change */
    _asm ("nop");
    _asm ("nop");
    _asm ("opt opnop");       /* Return assembly optimizer to normal
                              operation */
                              /* End of mode setup code */
```

4 Performing Preliminary Code Transformations

Architecture-independent code transformations set the baseline for optimization transformations in subsequent steps. Preliminary code transformations replace expensive arithmetic operations with less expensive ones that are computationally equivalent. The standard algorithms from ITU/ETSI frequently invoke primitives that are expensive to implement on Motorola DSPs, although less expensive primitives would suffice. For example, implementing an expensive primitive, such as a shift with a possibly negative shift amount on DSP56300 processors, is much more expensive than implementing a shift with an amount known to be positive (or known to be negative). We recommend altering code at instances where such an expensive primitive is required and mapping the rest of the primitive instances to the inexpensive alternative.

In some instances, static analysis of the source code reveals that a less expensive primitive would suffice. In other instances, such a determination can be made only by profiling the algorithm while running it through the full test vector suite. The technique described here addresses both cases.

The following process generates a list of all instances of fractional primitives in the standard algorithm that require the more expensive primitives. The list is created by running the test vectors through the algorithm. The list is dependent on the test vectors.

1. In `mathops.h`, enable the definition of the preprocessor macro MATH_CHECK by changing the line

   ```
   #undef MATH_CHECK
   ```

   to

   ```
   #define MATH_CHECK
   ```

2. One of the original C files in the standard algorithm contains the emulation routines for the primitives. In a previous step, the `mathops.h` file was included in this file (using `#include "mathops.h"`). Insert the following define statement before the line on which `mathops.h` is included:

```
#define SKIP_MATH_CHECK
#include "mathops.h"
```

3. Add the source file `motutil.c` to the build.

   This file supplies profiling routines for use in this phase only and is removed from the build before this phase is complete.

4. Recompile the standard algorithm and run the executable through the entire set of test vectors.

   As the application runs, it prints out messages to the standard output. These messages identify the source lines where primitives are invoked that must use their expensive implementation. For example, the following line indicates that the primitive L_shl( ), which is called from the source file `lsp.c` at line 23, must be implemented with saturation checking.

```
lsp.c, line 23: saturation occurred in L_shl.
```

5. Replace these instances with a call to the corresponding primitives that provide the full arithmetic functionality, using the conversion table shown here:

Change primitive …	**to this primitive, only if it appears in the profile list**
shl	_ e_shl
shr	_ e_shr
L_shl	_e_L_shl
L_shr	_e_L_shr

   Note that some warnings in the profile list may refer to add() and sub() primitives. In this step the add() and sub() primitives are not to be modified. Keep the profile list for use in a subsequent step.

6. Remove the `motutil.c` source file from the build.

5 Replacing Integer With Fractional Variables

Replacing integer variables with fractional variables proceeds in two phases:

- Identify the variables in the program, which are used for computations that are inherently fractional.
- Redefine these variables to use the TASKING C fractional data types.

Such replacement is crucial for extracting the maximal efficiency possible for compiled code on the DSP. When replacement is complete (and the application passes the test vectors), the variables used for inherently fractional computations are correctly redefined as fractional, and they are used in a manner consistent with their types. Variables for computations that are inherently integer retain their definitions, and integer operators are consistently applied to them.

Efficient code generation for the DSP requires that the fractional data types be used for the variables and expressions that compute fractional values. The ITU/ETSI coding style usually includes user-defined types for the variables that serve for fractional computations. Typically, these user-defined types are called Word16 and Word32, or Word and Longword, and are mapped to short (16-bit) and long (32-bit) fractional values. This coding style is not consistently followed. Often the standard algorithms employ these user-defined fractional types for variables and constants that are actually integers.

The remainder of this section presents a technique for replacing integer and fractional types with user-defined types. The steps in this technique are as follows:

1. Define user-defined types for integer and fractional types.
2. Identify and redefine inherently fractional variables.
3. Verify and correct type definitions.

Some type names are frequently used in the DSP applications for either integer or fractional variables. We recommend using the following naming conventions, as appropriate for the given initial code:

Name	Type on DSP	Type on Non-DSP Architecture
Word16	Fractional (16 bits)	Integer (16 bits)
Word32	Fractional (32 bits)	Integer (32 bits)
Word	Fractional (16 bits)	Integer (16 bits)
Longword	Fractional (32 bits)	Integer (32 bits)
Int16	Integer (16 bits)	Integer (16 bits)
Int32	Integer (32 bits)	Integer (32 bits)

The integer types are defined in the file `mottype.h`. You should disable (comment out) the definitions in the algorithm's original source code and use the definitions in `mottype.h` instead. Perform the following steps:

5.1 Define User-Defined Types for Integer and Fractional Types

If the application is coded using user-defined types for the inherently fractional variables, then the variables are basically identified. Redefining the variables to use the TASKING C fractional types (i.e. _fract and long _fract) only requires remapping the user-defined types to the built-in types. For example, the following lines from `mottype.h` remap the user-defined types to the built-in fractional types.

```
…
#define FDATA _fract        /* Create an auxiliary type name FDATA …*/
#define LFDATA long _fract  /* … and a type named LFDATA */
…
#define Word16  FDATA       /* Use the auxiliary type names to map
                               Word16 and Word32 */
#define Word32  LFDATA
```

The G.723.1 makefile maps the Word16 and Word32 user-defined types using the following C preprocessor switches:

```
CFLAGS = … -DWord16=short –Dword32=long …
```

When these switches are removed from the makefile, the definitions from `mottype.h` take effect. Next, the type definitions of variables are corrected to conform to the manner in which they are actually used. This redefining process is based on the type inconsistency warnings that the TASKING compiler issues when variables are defined and used inconsistently.

5.2 Identify and Redefine Inherently Fractional Variables

The first step in verifying and correcting variable type definitions is to add the compilation switch `–DMATH_FRACT` to the build (makefile or EDE project setting). Next, we recompile the application. The compiler flags all uses of variables that are inconsistent with the variables' defined types. Based on the compiler error and warning messages, we change the offending variable definitions. All warnings related to integer and fractional type inconsistencies point to real errors and thus should be treated as error messages. Following are warning and error messages that can result from integer and fractional type inconsistencies.

```
c563 W519: <file> line <number>: conversion of integer to fractional
type occurred
c563 W502: <file> line <number>: _fract constant saturation occurred
c563 E131: <file> line <number>: bad operand type(s) of '+'
```

Note: Many compiler error messages and warnings may appear when the code first compiles. We recommend that you first deal with the problems flagged by error messages and later deal with those flagged by warnings. For convenience, you can use the –w switch to direct the compiler to suppress warnings. You can identify a sample of the most common cases of variables used for inherently integer roles. The definitions for these variables can be modified to Int16 or Int32 before the code is compiled, considerably reducing the number of compiler error messages and warnings. The most common cases are variables used as loop indexes or as array indexes.

Note: The code is not clean until all error messages and warnings are eliminated. The warnings indicate problems that result in the generation of incorrect code.

Note: The transformations previously described need not be applied to the source file in which the emulation routines for the ITU/ETSI primitives are defined (typically this file is called `basicop.c` or `basic_op.c`).

5.3 Verify and Correct Type Definitions

The following situations occur frequently in the standard algorithms. For each situation a remedy is presented:

- Using integer constants in fractional expressions
- Unnecessary type casts
- Inconsistent use of the user-defined types
- Logical operations on fractional values
- Shift operations on fractional values
- Using primitives for integer arithmetic
- Using a variable as both Fractional and Integer

5.3.1 Using Integer Constants in Fractional Expressions

Some of the following cases use type casting macros defined in `mathops.h`:

```
#define _CI(X)  *(INT )&(X)         /* Convert to int */
#define _CPI(X) (INT *)(X)          /* Convert to pointer to int */
#define _CLI(X) *(long int *)&(X)   /* Convert to long */
#define _CF(X)  *(_fract *)&(X)     /* Convert to fractional */
#define _CPF(X) (_fract *)(X)       /* Convert to pointer to fract. */
#define _CLF(X) *(long _fract *)&(X)/* Convert to long fract. */
```

Warning: The compiler cannot detect use of the _CI() operator on a 32-bit value or use of _CLI() on a 16-bit value, so great care must be taken when applying the correct conversion operator.

First, we replace casts of constants with casts to the integer user-defined types (Int16 or Int32). If the constant is used in an assignment to a fractional variable, we wrap the variable instance using the _CI() or _CLI() macros. These macros cast the reference to be a reference of integer type, as shown in the following example.

```
Word32 Acc0;
…
Acc0 = (Word32) 0x04000000;
```

changes to:

```
Word32 Acc0;
…
_CLI(Acc0) = (Int32) 0x04000000;
```

If the assignment occurs in the variable definition, then the definition and initialization must be separated out of the definition, as follows.

```
Word32 Acc0 = (Word32) 0x04000000;
```

changes to:

```
Word32 Acc0;
_CLI(Acc0) = (Int32) 0x04000000;
```

If the constant is used in an expression or as an actual parameter to a function, then a temporary variable must be introduced as follows.

```
Word16 wf[10],temp;
wf[i] = sub(1843, mult(temp, 6242));
```

changes to:

```
Int16 tmp_1843 = 1843, tmp_6242 = 6242;
…
wf[i] = sub(_CF(tmp_1843), mult(temp, _CF(tmp_6242)));
```

This example uses the _CF() macro, which converts a reference into a reference to a fractional type. If the temporary variable definitions are placed in an inner block close to their use, the TASKING compiler can usually generate code that does not assign them memory or stack space.

5.3.2 Unnecessary Type Casts

Often the code contains casts to the user-defined types in situations that do not require an explicit cast. Replace the unnecessary type cast with one that casts the value to the corresponding integer user-defined type, as follows.

```
a = shr(a, (Word16) 1);
```

changes to:

```
a = shr(a, (Int16) 1);
```

5.3.3 Inconsistent Uses of the User-Defined Types

Sometimes variables defined using the user-defined type are used in roles that are inherently integer. For example, the Exp variable in the following example is inherently integer (it computes a bit offset inside a data word).

Example:

```
Word16 Exp, Acc0;
Exp = norm_s( Acc0 );
Acc0 = shr(Acc0, Exp);
```

This code is changed to:

```
Int16 Exp; Word16 Acc0;
Exp = norm_s( Acc0 );
Acc0 = shr(Acc0, Exp);
```

Other examples are loop indexes and array offsets that use variables defined with user-defined types. To correct this situation, redefine the variable as Int16 or Int32 as appropriate.

5.3.4 Logical Operations on Fractional Values

The logical operators in C (&, |, ^) cannot be applied to variables with the TASKING C types _fract or long _fract. Occasionally, the standard algorithm requires that such operators be applied to variables that are fractional, as shown in the following example. We change the access to the fractional variable to use the appropriate conversion operator (_CI() for 16-bit fractional variables, _CLI() for 32-bit fractional variables).

Example:

```
Word16 Acc0; Word32 Lvar;
Acc0 = extract_h(Lvar);
if (Acc0 & 1) ...
if (Lvar ^ 0xc0000000) ...
```

changes to:

```
Word16 Acc0; Word32 Lvar;
Acc0 = extract_h(Lvar);
if (_CI(Acc0) & 1) ...
if (_CLI(Lvar) ^ 0xc0000000) ...
```

The conversion operators are defined in `mathops.h`. The conversion operators cannot be applied to results of arithmetic computations (e.g. values returned by functions). In such cases, the arithmetic expression must be computed into a temporary variable and the operator applied to the variable, as shown here.

Example:

```
if (extract_h(Lvar) & 1) ...
```

converts to:

```
Word16 ftmp;
ftmp = extract_h(Lvar);
if (_CI(ftmp) & 1) ...
```

5.3.5 Shift Operations on Fractional Values

The shift operators in C (<<, >>) cannot be applied to variables defined with the TASKING C types _fract and long _fract. Occasionally, the standard algorithm requires that such operators be applied to variables that are fractional. Typically, these operations are applied from the standard C program by

calling the primitives (shr() and shl()). If the original code applies the built-in C operators (<< or >>) to fractional variables, then the most convenient technique for correcting the code is to cast the fractional variable accesses using the _CI() and _CLI() operators (similarly to Section 5.3.4). The rest of this description addresses cases where the shift operations are specified using the shift primitives (shl() or shr()).

If the shift is by an amount that is computed at run time (i.e. the second parameter to the shl() or shr() functions is not a literal), then we use the technique described previously for logical operations. If the shift is by an amount that is known at run time (and that amount is larger than 0), we replace the call to the shift function (shr() or shl()) by calls to the macros _f_shr() and _f_shl(). These macros are transformed by the compiler to DSP56300 shift instructions. For example:

```
a = shr(a, (Word16) 1);
```

changes to:

```
a = _f_shr(a, (Int16) 1);
```

5.3.6 Using Primitives for Integer Arithmetic

Occasionally fractional primitives, such as add( ) and shl( ), are used for computing integer expressions. Such instances are characterized by the appearance of inherently integer variables, such as loop indexes and array indexes, as parameters to the primitives or as the destination for their results. Although this is only a warning, the compiler may generate incorrect code for the expression. We recommend that you modify the code to prevent such warnings. The profiling in an earlier step identified instances that must use the fractional arithmetic, if such instances exist. We replace invocation of the primitives by invocations of corresponding primitive. These primitives are defined in `mathops.h`:

Change primitive …	**to this primitive, unless it appears in the profile list**
add	_i_add
sub	_i_sub
shl	_i_shl
shr	_i_shr

In theory, these instances may require the exact functionality of the fractional operation (e.g. saturation on the addition overflow). If the instance of the primitive appears in the profile list, the operation must retain its full fractional arithmetic characteristics. The parameters and the return value for the invocation must be treated according to the previous description for inconsistent use of variables. For example:

```
Word16 Pr;
...
Pr = sub(Pr, (Word16) 1);
...
Acc0 = Buf[Pr];
```

The computation of Pr changes to

```
Int16 Pr;
...
Pr = _i_sub(Pr, (Int16) 1);
. . .
Acc0 = Buf[Pr];
```

5.3.7 Using a Variable as Both Fractional and Integer

Occasionally, the application uses a variable both as fractional and as integer, although in mutually exclusive lifetimes. For example, a variable is assigned an integer value and used as an integer value, and then later it is assigned to a fractional value and used as such. Since the variable is defined as either a fractional or an integer, the compiler flags all uses that are inconsistent with the definition.

```
Ccr = norm_l(Acc0);      /* Ccr is used as an integer (offset) */
Acc0 = L_shl(Acc0, Ccr);
...
Ccr = round(Acc0);       /* the same Ccr is used as a fractional */
Acc0 = L_mult(Ccr, Ccr);
```

We could convert all uses of the conflicting type using the conversion operators, but this conversion would be tedious. A simpler solution is to define an auxiliary variable and use it for all occurrences of the original variable requiring the conflicting data type. The advantage of this method is that it prevents having to insert many conversion operators. The disadvantage of this method is that we must first ensure that the uses of the original variable with conflicting types are mutually exclusive.

For the preceding example, the two possible solutions are as follows:

```
/* --- Solution 1, using conversions --- */
Int16 Ccr;
...
Ccr = norm_l(Acc0);
Acc0 = L_shl(Acc0, Ccr);
...
_CF(Ccr) = round(Acc0);
Acc0 = L_mult(_CF(Ccr), _CF(Ccr));

/* --- Solution 2, using an auxiliary variable --- */
Int16 Ccr; Word16 fCcr;
...
Ccr = norm_l(Acc0);
Acc0 = L_shl(Acc0, Ccr);
...
fCcr = round(Acc0);
Acc0 = L_mult(fCcr, fCcr);
```

Note: For maximum code performance, try both techniques and compare the results.

This step completes only when the application code compiles without any error messages or warning messages. When this step completes, the application uses fractional data types consistently. The fractional primitives are still computed by subroutines. In the next step, the calls to most of these primitives are replaced by inlined primitives, which yield much higher performance.

6 Inlining the Primitives

Inlining most of the fractional primitives significantly improves the performance and the code size of the compiled standard algorithm. When this step is completed (and the application passes the test vectors), the primitives are correctly inlined, and the compiler generates assembly instructions for performing most of the computationally intensive operations. First, we set up the compilation to include the compiler switches -DMATH_FRACT and -DMATH_INLINE and then we recompile the standard algorithm.

Inlining the primitives is treated as a separate phase to facilitate easier debugging. Since many code transformations are performed when integer variables are replaced with fractional ones, verify correctness of the transformations before inlining the primitives.

7 Using Optimized Out-of-Line Primitives

The purpose of using optimized out-of-line primitives is to map additional fractional operations not previously inlined to efficient hand-coded assembly implementations. When this step is completed (and the application passes the test vectors), the fractional primitives not inlined in prior steps are now implemented using efficient hand-coded assembly subroutines.

First, we set up the compilation to include the compiler switches -DMATH_FRACT_OPT, and then we recompile the standard algorithm. When this compilation switch is set, some optimized primitives available in `mathops.c` become part of the build. These primitives have the same names as primitives already defined (and emulated) in the original standard algorithm code. It is now necessary to disable (comment out) the conflicting definitions in the original standard algorithm source code. To do this, we simply identify which multiple definitions of subroutines (primitives) the TASKING linker reports, find the source code for these subroutines in the original source code (not in `mathops.c`!) and comment them out using #ifndef _DSP and #endif directives. When the code is compiled again, the duplicate definitions are removed and the link phase thus can complete successfully.

8 Further Optimizations

At this point the application is fully converted to use the TASKING fractional data types with inlined arithmetic operations. We can optimize the code even further using the optimization hints that are discussed in the TASKING compiler user manual. Examples of optimizations that can be extremely beneficial when the standard algorithm is implemented using the TASKING C compiler are:

- *Separating variables into X and Y data memory spaces*. The compiler assigns all variables into one data memory space unless the user specifically assigns variables to the non-default data memory space. Tagging variable declarations with TASKING C data memory space specifiers (_X or _Y) performs this assignment. Achieving maximal levels of performance on DSP56300 requires that the user judiciously assign variables to the non-default data memory space. If the application data size requirements are high, applying this optimization technique may be helpful in reducing the data size requirements to the level that allows the code to run on the target development board.
- *Using pointer references instead of array references in loops*. Occasionally, the compiler may miss optimization opportunities when memory references are written using array references, and may optimize the code better if the memory references are rewritten using equivalent pointer references.
- *Moving invariant code out of loops*. Occasionally the compiler may miss opportunities to move computations that are invariant to the loop iteration from inside the loop to its surroundings. It may be possible to perform this optimization at the C program level and thus reduce the computational requirements of the loop.
- *Software pipelining*. Occasionally the compiler may miss opportunities to “pipeline” a loop body, meaning to transform the structure of the loop statements into one that allows for increased parallelism. In such cases, we can “software pipeline” the loop at the C program level, exposing the increased parallelism for the compiler.
- *Separating uses of the same variable (having separate lifetimes) into uses of variables having different names*. This sometimes helps the compiler allocate registers to these variables more efficiently.
- *Loop unrolling*. Occasionally, performance is improved when loops that iterate for a small and predefined number of times are unrolled (i.e. the body of the loop is replicated). Because code size typically increases when this optimization technique is used, this technique should be used judiciously.

9 Summary

The technique described in this application note has been successfully applied to the standard C implementation of the G.723.1 and G.729a codecs. The transformation of the entire C implementation based on the steps detailed in this application note require no more than several hours for one programmer. The performance of the application depends on the additional time that is spent on further applying the additional optimization techniques briefly described.

Following is the original code of one routine from a standard C application:

```
PWDEF Comp_Pw( Word16 *Dpnt, Word16 Start, Word16 Olp )
{
    int   i,j   ;
    Word32   Lcr[15] ;
    Word16   Scr[15] ;
    PWDEF    Pw ;

    Word32   Acc0,Acc1   ;
    Word16   Exp   ;
    Word16   Ccr,Enr  ;
    Word16   Mcr,Mnr  ;

    Lcr[0] = (Word32) 0 ;
    for ( i = 0 ; i < SubFrLen ; i ++ )
        Lcr[0] = L_mac( Lcr[0], Dpnt[Start+i], Dpnt[Start+i] ) ;
    …
    Acc1 = (Word32) 0 ;
    for ( i = 0 ; i < 15 ; i ++ ) {
        Acc0 = Lcr[i] ;
        Acc0 = L_abs( Acc0 ) ;
        if ( Acc0 > Acc1 )
             Acc1 = Acc0 ;

    }
    Exp = norm_l( Acc1 ) ;
    for ( i = 0 ; i < 15 ; i ++ ) {
        Acc0 = L_shl( Lcr[i], Exp ) ;
        Scr[i] = round( Acc0 ) ;
    }

    Pw.Indx = (Word16) -1 ;
    Pw.Gain = (Word16) 0  ;
    Mcr = (Word16) 1 ;
    Mnr = (Word16) 0x7fff ;
    for ( i = 0 ; i <= 2*PwRange ; i ++ ) {
        Enr = Scr[2*i+1] ;
        Ccr = Scr[2*i+2] ;
        if ( Ccr <= (Word16) 0 )
             continue ;
        Exp = mult_r( Ccr, Ccr ) ;
        Acc0 = L_mult( Exp, Mnr ) ;
        Acc0 = L_msu ( Acc0, Enr, Mcr ) ;
        if ( Acc0 > (Word32) 0 ) {
             Mcr = Exp ;
             Mnr = Enr ;
             Pw.Indx = (Word16)i ;
        }
    }
    …
    Acc0 = L_mult( Scr[0], Mnr ) ;
    Acc1 = Acc0 ;
    Acc0 = L_shr( Acc0, (Word16) 2 ) ;
    Acc1 = L_shr( Acc1, (Word16) 3 ) ;
    Acc0 = L_add( Acc0, Acc1 ) ;
    Acc1 = L_mult( Scr[2*Pw.Indx+2], Scr[2*Pw.Indx+2] ) ;
```

```
    Acc0 = L_sub( Acc0, Acc1 ) ;
    …
    Pw.Indx = Olp - PwRange + Pw.Indx ;
    return Pw ;
}
```

Following is the code of the same routine after application of the steps described in this application note and before the application of the additional optimizations. Modifications to the code are emphasized in bold italics.

```
PWDEF Comp_Pw( Word16 *Dpnt, Int16 Start, Int16 Olp )
{
    int   i,j   ;
    Word32   Lcr[15] ;
    Word16   Scr[15] ;
    PWDEF    Pw ;

    Word32   Acc0,Acc1   ;
    Int16   Exp   ;
    Word16   Ccr,Enr  ;
    Word16   Mcr,Mnr  ;

    Lcr[0] = (Word32) 0 ;
    for ( i = 0 ; i < SubFrLen ; i ++ )
        Lcr[0] = L_mac( Lcr[0], Dpnt[Start+i], Dpnt[Start+i]) ;

    …
    Acc1 = (Word32) 0 ;
    for ( i = 0 ; i < 15 ; i ++ ) {
        Acc0 = Lcr[i] ;
        Acc0 = L_abs( Acc0 ) ;
        if ( Acc0 > Acc1 )
             Acc1 = Acc0 ;

    }

    Exp = norm_l( Acc1 ) ;
    for ( i = 0 ; i < 15 ; i ++ ) {
        Acc0 = _e_L_shl( Lcr[i], Exp ) ;
        Scr[i] = round( Acc0 ) ;
    }
    Pw.Indx = (Int16) -1 ;
    Pw.Gain = (Int16) 0  ;
    _CI(Mcr) = (Int16) 1 ;
    _CI(Mnr) = (Int16) 0x7fff ;
    for ( i = 0 ; i <= 2*PwRange ; i ++ ) {
        Word16 F_Exp;
        Enr = Scr[2*i+1] ;
        Ccr = Scr[2*i+2] ;
        if ( Ccr <= (Word16) 0 )
             continue ;
        F_Exp = mult_r( Ccr, Ccr ) ;
        Acc0 = L_mult( F_Exp, Mnr ) ;
        Acc0 = L_msu ( Acc0, Enr, Mcr ) ;
        if ( Acc0 > (Word32) 0 ) {
             Mcr = F_Exp ;
             Mnr = Enr ;
             Pw.Indx = (Int16)i ;
        }
    }
    …
    Acc0 = L_mult( Scr[0], Mnr ) ;
    Acc1 = Acc0 ;
    Acc0 = L_shr( Acc0, (Int16) 2 ) ;
    Acc1 = L_shr( Acc1, (Int16) 3 ) ;
    Acc0 = L_add( Acc0, Acc1 ) ;
```

```
        Acc1 = L_mult( Scr[2*Pw.Indx+2], Scr[2*Pw.Indx+2] ) ;
        Acc0 = L_sub( Acc0, Acc1 ) ;
        …
        Pw.Indx = Olp - PwRange + Pw.Indx ;
        return Pw ;
    }
```

Order Number: #AN1772/D

How to reach us:

USA/Europe/Locations Not Listed:
Motorola Literature Distribution
P.O. Box 5405
Denver, Colorado 80217
1 (800) 441-2447
1 (303) 675-2140

Asia/Pacific:
Motorola Semiconductors H.K. Ltd.
8B Tai Ping Industrial Park
51 Ting Kok Road
Tai Po, N.T., Hong Kong
852-26629298

Japan:
Nippon Motorola Ltd
SPD, Strategic Planning Office141
4-32-1, Nishi-Gotanda
Shinagawa-ku, Japan
81-3-5487-8488

Motorola Fax Back System (Mfax™):
TOUCHTONE (602) 244-6609
1 (800) 774-1848
RMFAX0@email.sps.mot.com

Technical Resource Center:
1 (800) 521-6274

DSP Helpline
dsphelp@dsp.sps.mot.com

Internet:
http://www.motorola-dsp.com/