Application Report
SPRA552B- February 2000

G.723.1 Dual-Rate Speech Coder: Multichannel TMS320C62x Implementation

Thomas J. Dillon, Jr. *C6000 Applications*

ABSTRACT

This application report describes how the G.723.1 Dual-Rate Speech Coder has been implemented on the Texas Instruments (TI™) TMS320C62x digital signal processor (DSP). Beyond the use of the ’C62x intrinsic functions, the application report includes specific changes required to allow this coder to operate in a real-time system with other speech coders. Also reported is information on several optimization techniques used to yield multiple channels running concurrently. Finally, the application report will include the performance resulting from this implementation of the algorithm.

Contents

List of Figures

List of Tables

1 Introduction

The purpose of this application report is to describe the implementation of the G.723.1 Dual-Rate Speech Coder in a multichannel system on the TMS320C62x DSP. Included in this application report is a description of the algorithm with details on the encoder, decoder, and the calling program. Coding guidelines used to implement the algorithm are covered with special attention to static and dynamic memory, algorithm tables, and context data for multiple channels of the algorithm. Also, changes necessary for this algorithm to coexist with other algorithms are discussed. Finally, the process of building, running, and testing this algorithm is outlined and the resulting performance and code size numbers are reported.

1.1 Algorithm Overview

The G.723.1 speech coder [1],[2] is designed to compress audio data as part of the H.324 family of multimedia communication standards. This coder has two bit rates associated with it, namely, 5.3 kbit/s and 6.3 kbit/s. This allows the system designer some flexibility by providing good quality representation of the voice data with a fixed level of algorithm complexity. It encodes frames of speech using linear predictive analysis-by-synthesis coding. The 6.3-kbit/s coder uses Multipulse Maximum Likelihood Quantization (MP-MLQ) for the excitation signal. The 5.3-kbit/s coder uses Algebraic-Code-Excited Linear Prediction (ACELP). For both rates, the frame size is 30 ms, which is larger than a number of the other ITU speech coder standards.

1.2 Multichannel System

TI's implementation of the G.723.1 is capable of multichannel processing. In addition, it is also compliant with the eXpressDSP™ Algorithm Standard (xDAIS). The general IALG interface, as well as the algorithm-specific interface defined by TI, has been applied in this implementation of the G.723.1. Please refer to References [6], [7], and [8] for more on xDAIS and on how to make algorithms xDAIS-compliant.

The multichannel system [5] has the capability to run more than one algorithm at the same time. Any algorithm running on the multichannel system must have three functional modules: initialization, free, and kernel. The initialization and the free modules initialize and release the algorithm, respectively, while the kernel module performs the algorithm processing.

The IALG API for initialization is defined as

```
Int     (*algInit)(IALG_Handle, const IALG_MemRec *, IALG_Handle, const
IALG_Params *);
```

and the IALG API to free the algorithm is defined as

```
Int     (*algFree)(IALG_Handle, IALG_MemRec *);
```

where the first parameter in both cases is of type IALG_Handle, which is a pointer that contains the current context data memory location.

During initialization, the multichannel system calls the algorithm initialization functions. All the initialization functions perform the same tasks: initialize the context data and store it to the desired memory location. The function returns an integer to indicate whether the function ended in success or not. After initialization is complete, the system repeatedly calls an algorithm or algorithms until all frames of input data have been processed.

eXpressDSP is a trademark of Texas Instruments Incorporated.

The system may need to call the algFree() function to free the algorithm instance after all processing is complete.

The APIs for processing frames with the G.723.1 algorithm are defined as

```
extern DAIS_Int8 G723E_TI_encode(IG723ENC_Handle handle, DAIS_Int16*
in, DAIS_UInt8* out);
extern DAIS_Int8 G723D_TI_decode(IG723DEC_Handle handle, DAIS_UInt8*
in, DAIS_Int16 out[]);
```

where handle points to the start address of the context data which contains the start address of the constant tables of the algorithm. The pointers in and out point to the input and output data, respectively. Since xDAIS does not specify standard APIs for algorithm processing, these two APIs are algorithm-specific.

2 Algorithm Description

The G.723.1 coder uses the principles of linear prediction analysis-by-synthesis coding to minimize a perceptually weighted error signal.

The encoder starts with a 30-ms frame of 240 samples. This frame is high-pass filtered and is broken into 4-ms to 60-ms subframes. A 10th-order Linear Prediction Coder (LPC) filter is computed on each subframe. A perceptually weighted speech signal is computed from these LPC coefficients. Next, the weighted speech signal is used to compute the open-loop pitch period on two 120-sample subframes.

All remaining processing is done on 60 sample frames. It starts with the computation of a harmonic noise-shaping filter, which is then used with the LPC synthesis filter to create an impulse response. The impulse response is used to compute a closed-loop pitch predictor. The pitch period and a differential value computed from the previous step are sent to the decoder.

Finally, two methods are used to compute the nonperiodic component of the excitation in the encoder:

- The 5.3 rate coder uses Algebraic-Code-Excited Linear Prediction (ACELP)
- The 6.3 rate coder uses Multipulse Maximum Likelihood Quantization (MP-MLQ).

The decoder operates on a frame-by-frame basis. Decoding begins with the quantized Linear Prediction Coefficients, which are used to create the LPC synthesis filter. The excitation for each subframe is computed using the decoded adaptive and fixed codebooks. The excitation signal is filtered by the pitch postfilter and then filtered by the synthesis filter to produce a decoded result, which may have a gain step applied.

Refer to ITU recommendation G.723.1 for a detailed description of this vocoder [1],[2].

2.1 Features

These are the speech coder features:

- The complete speech coder is implemented in mixed C and TMS320C62x Assembler languages. Each basic math operation defined in basop.c either is replaced by its corresponding TMS320C62x intrinsic or is static-inlined.

- The coder is multichannel enabled. After processing one frame of one channel, the coder is capable of processing one frame of another channel. Further effort is needed to make the coder interruptible within a frame that is processing, to allow the decoder higher priority than the encoder and/or timely delivery of the parameters to the next stage of the whole process.
- Features, including voice activity detection (VAD) and comfort noise generation (CNG) are implemented.
- The speech coder is fully bit-compatible with the bit-exact reference C code on all testing sequences.

2.2 Encoder Structure

The G.723.1 speech encoder is divided into the following modules:

- Coder, which includes the framer and initialization routines. The framer reads input frames (240 samples) and subdivides them into subframes (60 samples).
- Linear Predictive Analysis, which includes LPC, Formant perceptual weighting filter, impulse response calculator, zero input response and ringing subtraction, and memory update
- Line Spectrum Analysis, which includes LSP quantizer, decoder, and interpolation
- Adaptive and Fixed Excitation, which includes pitch estimation, harmonic noise shaping, pitch predictor, high-rate excitation (MP–MLQ), low-rate excitation (ACELP), excitation decoder, and decoding of the pitch information
- Silence Compression, which includes voice activity detection (VAD) and comfort noise generation (CNG)
- Miscellaneous utility functions, which include high-pass filter, CNG utilities, and bit allocation

2.3 Decoder Structure

The G.723.1 speech decoder is divided into the following modules:

- Decoder, which includes the initialization routines as well as the decoder
- Line Spectrum Analysis, which includes LSP decoder and interpolation
- Linear Predictive Analysis, which includes LPC synthesis and Formant postfilter
- Adaptive and Fixed Excitation, which includes decoding of pitch information, excitation decoder, pitch postfilter, and frame interpolation
- Silence Compression, which includes decode of CNG affected frames
- Miscellaneous utility functions, which include gain scaling, and CNG utilities

2.4 Main Program

The G.723.1 speech coder contains the following modules:

- Main, contained in lbccodec.c, which is used for initializing and calling the encoder/decoder outside the demo (for example, validation of test vectors)

- Coder, which is described above and includes the 5.3 rate and 6.3 rate coders
- Decoder, which is also described above and supports both rates
- Tables of constants used by the encoder and decoder
- Basic operations for fixed-point arithmetic and logical operation
- Header files, which include data type definitions for C62x, function prototypes, and external declaration for constant tables

3 Coding Guidelines

This program is a mixed C and Assembly language implementation.

3.1 Variable, Array, and Pointer Names

The same naming conventions for variables, arrays, and pointers were used as in the reference C code.

3.2 Math Operations

All math operations defined in basop.c in the reference C program are replaced by TMS320C62x intrinsics, if possible. In addition to the usage of intrinsics, the basic operations div_s, div_l, L_mls, L_shr_r, and negate are inlined. Their expressions in intrinsics or inlined versions reside in basop.h. For the description and the usage of the TMS320C62x intrinsics, refer to *TMS320C6000 Programmer's Guide* (SPRU198).

3.3 Tables and Functions

The names of the tables are the same as the ones in the reference C program. Refer to Tables 1–8 in this document for the name and size of all the tables in this coder.

The names of the functions are the same as the ones in the reference C program, except for Cor_h and Cor_h_X. The names of these functions were changed to G723Cor_h and G723Cor_h_X, respectively, to avoid conflicting with functions of the same names in other ITU coders.

Each function is either in C or assembly. Many of the functions are contained in larger modules, as described earlier. A preprocessor directive is used to turn off the C file and turn on the assembly function. The name of an assembly function closely matches its name in the reference C program. For a function named * in the reference C program, the hand-optimized assembly function is named *.asm in this implementation.

3.4 Interrupt Issues

The coder is multichannel-enabled in the sense that after processing one frame of one channel, the code is capable of processing one frame of another channel. That is, the coder is interruptible at the frame process boundary. Further study is required to make the coder interruptible within a frame process.

The coder can be interrupted at any place other than software pipelined loops. Of course, it can be elected to not interrupt the coder until the end of a frame process, as long as the overall system design requirement is met. If interrupts are necessary, it is suggested, in general, that interrupts occur in certain places so that the increases in data memory and cycle counts would be kept minimal. One particular interrupt scheme for this coder, where an interrupt occurs at the end of each submodule, is described.

4 Static and Dynamic Memory Requirements

The data memory can be characterized into static data memory and dynamic data memory. Static data memory includes static variables and arrays whose values have to be kept from one frame to the next. This includes the context data associated with each channel of data being processed and the algorithm tables. Dynamic data memory includes the stack memory required to process the audio data in the vocoder.

The stack has been measured for the encoder/decoder combination as follows:

- 5.3 rate encoder and decoder stack –> 0x1FF0 = 8176 bytes
- 6.3 rate encoder and decoder stack –> 0x0D00 = 3328 bytes

4.1 Encoder

4.1.1 Context Data – Encoder

A number of data structures is defined in the ITU code for the G.723.1 Encoder [1],[2], as shown in Figure 1, Figure 2, and Figure 3.

```
typedef struct
{
  /* Lsp previous vector */
  Word16  PrevLsp[LpcOrder];
  /* Used delay lines */
  Word16  WghtFirDl[LpcOrder];
  Word16  WghtIirDl[LpcOrder];
  Word16  RingFirDl[LpcOrder];
  Word16  RingIirDl[LpcOrder];
  /* High pass variables */
  Word16  HpfZdl;
  Word32  HpfPdl;
  /* Sine wave detector */
  Word16  SinDet;
  /* All pitch operation buffers */
  Word16  PrevWgt[PitchMax];
  Word16  PrevErr[PitchMax];
  Word16  PrevExc[PitchMax];
  /* Required memory for the delay */
  Word16  PrevDat[LpcFrame–SubFrLen];
  /* Taming procedure errors */
  Word32 Err[SizErr];
} CODSTATDEF;
```

Figure 1. CODSTATDEF – Encoder Static Variables and Arrays Data Structure

```
typedef struct
{
  Word16 NLpc[LpcOrder];
  Word16 Hcnt;
  Word16 Vcnt;
  Word32 Penr;
  Word32 Nlev;
  Word16 Aen;
  Word16 Polp[4];
} VADSTATDEF;
```

Figure 2. VADSTATDEF – Encoder VAD Static Variables and Arrays Data Structure

```
typedef struct {
  Word16 SidLpc[LpcOrder];
  Word16 LspSid[LpcOrder];
  Word16 CurGain;
  Word16 PastFtyp;
  Word16 Acf[SizAcf];
  Word16 ShAcf[NbAvAcf+1];
  Word16 RC[LpcOrderP1];
  Word16 ShRC;
  Word16 Ener[NbAvGain];
  Word16 NbEner;
  Word16 IRef;
  Word16 SidGain;
  Word16 RandSeed;
} CODCNGDEF;
```

Figure 3. CODCNGDEF – Encoder CNG Static Variables and Arrays Data Structure

The approximate memory requirements for these data structures, before alignment, are as follows:

- CODSTATDEF – 1238 bytes
- VADSTATDEF – 42 bytes
- CODCNGDEF – 178 bytes

The file cst_lbc.h contains the description of each of the elements of these structures. The total memory required for context is listed in the *Performance / Code Size Results* section of this application report.

4.1.2 Stack Data – Encoder

The stack memory required for this encoder varies depending upon the functions called and which options are selected for rate, filter, VAD, etc. The maximum for the encoder has not been measured at this time, but it is less than 0x2000.

4.2 Decoder

4.2.1 Context Data – Decoder

A number of data structures is defined in the ITU code for the G.723.1 Decoder [1],[2], as shown in Figure 4 and Figure 5.

```
typedef struct
{
  /* Lsp previous vector */
  Word16  PrevLsp[LpcOrder];
  /* All pitch operation buffers */
  Word16  PrevExc[PitchMax];
  Word16  Ecount;
  Word16  InterGain;
  Word16  InterIndx;
  Word16  Rseed;
  Word16  Park;
  Word16  Gain;
  /* Used delay lines */
  Word16  SyntIirDl[LpcOrder];
  Word16  PostFirDl[LpcOrder];
  Word16  PostIirDl[LpcOrder];
} DECSTATDEF;
```

Figure 4. DECSTATDEF – Decoder Static Variables and Arrays Data Structure

```
typedef struct {
  Word16 CurGain;
  Word16 PastFtyp;
  Word16 LspSid[LpcOrder];
  Word16 SidGain;
  Word16 RandSeed;
} DECCNGDEF;
```

Figure 5. DECCNGDEF – Decoder CNG Static Variables and Arrays Data Structure

The approximate memory requirements for these data structures, before alignment, are as follows:

- DECSTATDEF – 382 bytes
- DECCNGDEF – 28 bytes

The file cst_lbc.h contains the description of each of the elements of these structures. The total memory required for context is listed in the *Performance / Code Size Results* section of this application report.

4.2.2 Stack Data – Decoder

The stack memory required for this decoder varies depending upon the functions called and which options are chosen for rate, filter, etc. The maximum for the decoder has not been measured at this time, but it is less than 0x2000.

4.3 Tables for the G.723.1 Coder

Tables 1 through 7 are the tables used by the G.723.1 coder. Some of the tables are common to both the encoder and the decoder. The tables are found in the files tab_lbc.c and tab_lbc.h. Memory for the tables is independent of the number of channels running on the chip.

Table 1. LPC Computation and Filtering Tables

Table	Size (bytes)
HammingWindowTable	360
BinomialWindowTable	20
BandExpTable	20
Total	**400**

Table 2. LSP Calculation and Quantization Tables

Table	Size (bytes)
CosineTable	1024
LspDcTable	20
BandInfoTable	12
Band0Tb8	1536
Band1Tb8	1536
Band2Tb8	2048
BandQntTable	12
Total	**6188**

Table 3. Perceptual Filtering and Post Filtering Tables

Table	Size (bytes)
PerFiltZeroTable	20
PerFiltPoleTable	20
PostFiltZeroTable	20
PostFiltPoleTable	20
LpfConstTable	4
Total	**84**

Table 4. Excitation Tables – ACELP and MP-MLQ

Table	Size (bytes)
Nb_puls (MP-MLQ)	8
FcbkGainTable (ACELP) and (MP-MLQ)	48
MaxPosTable (MP-MLQ)	16
CombinatorialTable (MP-MLQ)	720
epsi170 (ACELP)	340
gain170 (ACELP)	340
Total	**1472**

Table 5. Pitch Prediction Tables

Table	Size (bytes)
AcbkGainTable085	3400
AcbkGainTable170	6800
AcbkGainTablePtr	8
Total	**10208**

Table 6. Taming Procedure Tables

Table	Size (bytes)
Tabgain170	340
Tabgain85	170
Total	**510**

Table 7. Comfort Noise Generation Tables

Table	Size (bytes)
Fact	8
L_bseg	12
Base	6
Total	**26**

5 Multichannel Modifications

This section includes details on what was required to make this algorithm work in a multichannel implementation.

5.1 Wrapper API

To ensure access to the correct memory of each channel, from one frame to the next, the static variables and arrays are sorted into one of two structures. The encoder uses an instance of the CODCNTXTDEF structure accessible via *G723_CodCntxtPtr. The decoder uses an instance of the DECCNTXTDEF structure via *G723_DecodCntxtPtr. Both of these two structures are defined in mult.h.

The structures described above are initialized, respectively, within the standard IALG functions, Init_EncodG723() and Init_DecodG723(). Rate information (5.3 or 6.3) is passed into the init_EncodG723() function. These functions are defined in the file mult.c.

A data pointer structure is defined in the file tab_lbc.h, which merges the context and table information for coder processing as shown in Figure 6.

```
typedef struct
{
      void *Context;
      void *Table;
} G723_DATA_PTR;
```

Figure 6. Data Pointer Structure Used for the Encoder and Decoder

To start an encoding process of a channel, the xDAIS API is

```
extern DAIS_Int8 G723E_TI_encode(IG723ENC_Handle handle, DAIS_Int16*
in, DAIS_UInt8* out);
```

where the pointer handle denotes the start address of the Data Pointer structure of that particular channel; the pointer in points to the start address of the input speech buffer of the frame to be processed; and the pointer out points to the start address of the encoded bit stream buffer. This xDAIS API function then makes a call to the actual encoder function defined as

```
void Encoder_G723(void *ChanData, short **Input, char **Output)
```

where the ChanData actually points to a G723_DATA_PTR data structure that contains the starting address of context data as well as the constant table.

Similarly, to start a decoding process of a channel, the xDAIS API is

```
extern DAIS_Int8 G723D_TI_decode(IG723DEC_Handle handle, DAIS_UInt8*
in, DAIS_Int16 out[]);
```

where the pointer handle denotes the start address of the Data Pointer structure of that particular channel; the pointer in points to the start address of the input bit stream buffer; and the pointer out points to the start address of the synthesized speech buffer. This functions also makes a call to the actual decoder function defined as:

```
void Decoder_G723(void *ChanData, short **Input, char **Output)
```

Note that rate information is included in the ChanData structure; therefore, the Encoder_G723 and Decoder_G723 functions do not require a 5.3 and a 6.3 rate version.

5.2 Static Variables and Tables

5.2.1 Context Data – Encoder

These data structures were identified previously for the G.723.1 Encoder:

- CODSTATDEF – Encoder Static Variables and Arrays
- VADSTATDEF – Encoder VAD Static Variables and Arrays
- CODCNGDEF – Encoder CNG Static Variables and Arrays

All of these structures, along with some static variables, which define the rate and flag the filter and VAD status, are combined into a context data structure as shown in Figure 7.

```
typedef struct {
 CODSTATDEF CodStat;
 VADSTATDEF VadStat;
 CODCNGDEF CodCng;
 enum Crate WrkRate;
 Flag UseHp;
 Flag UseVx;
} CODCNTXTDEF;
```

Figure 7. Encoder Context Data Structure

The file cst_lbc.h contains the description of each of the elements of the structure and mult.h includes this data structure. The total memory required for context is listed in the *Performance / Code Size* section of this application report.

5.2.2 Context Data – Decoder

These data structures were identified previously for the G.723.1 Decoder:

- DECSTATDEF – Decoder Static Variables and Arrays
- DECCNGDEF – Decoder CNG Static Variables and Arrays

These structures and some static variables, which define the rate and flag the filter status, are combined into a context data structure as shown in Figure 8.

```
typedef struct {
 DECSTATDEF DecStat;
 DECCNGDEF DecCng;
 enum Crate WrkRate;
 Flag UsePf;
} DECCNTXTDEF;
```

Figure 8. Decoder Context Data Structure

The file cst_lbc.h contains the description of each of the elements of this structure and mult.h includes this data structure. The total memory required for context is listed in the *Performance / Code Size* section of this application report.

5.2.3 Super Table – Encoder and Decoder

Tables 1 through 7 of the G.723.1 vocoder were described previously in the *Static and Dynamic Memory Requirements* section of this application report. These tables are combined into one super table so that it can be relocated during the operation of the multichannel system, as shown in Table 8.

Table 8. Combination of G.723.1 Tables

Tables	Size (bytes)
LPC Computation and Filtering	400
LSP Calculation and Quantization	6188
Perceptual Filtering and Post Filtering	84
Excitation – ACELP or MP-MLQ	1472
Pitch Prediction	10208
Taming Procedure	510
Comfort Noise Generation	26
Miscellaneous	104
Total	**18992**

6 Running the Program

This section describes the procedure to configure and run the G.723.1 speech coder program.

6.1 Build Procedure

The G.723.1 coder can be configured to run on its own or it can be linked to the multichannel system. Building the code requires the following steps:

1. Compile and assemble the individual modules without main.
   cl6x –@options
2. Create a library module of the .obj files.
   ar6x @aroptions
3. Compile and link with main to create a stand-alone version of G.723.1.
   cl6x –@opttest
4. Copy the library to the multichannel system directory. (Note that the path may vary!)
   Copy g723.lib c:\ti\myprojects\system
5. Change to the system directory:
   cd c:\ti\myprojects\system
6. Compile and link the library with the system and other algorithms.
   cl6x –@options

The code directory of the G.723.1 coder contains the files necessary to perform the compilation, assembly, and link with the appropriate linker command files lnk.cmd in the coder and system directories. It builds all the object files (.obj) in the code directory and links them, producing a .map file (codec723.map), which contains the memory-mapping information. The stand-alone executable file is codec723.out. It also builds all the object files (.obj) in the system directory and links them, producing a .map file (dsp.map) and a C6x .out file for the Windows™ GUI program to load on the EVM. The compiling and linking options presented in the options and lnk.cmd files assume the 3.00 C6x Code Generation Tools.

6.1.1 Encoder

The encoder is called from the file lbccodec.c, for the stand-alone version, or the file component.c, for the multichannel system version. The calling function must initialize the encoder before calling the Encoder_G723() function to start the encoding process of a new channel.

The function Init_EncodG723( ) is called one time before the encoder, to initialize the static variables and arrays for each channel. The rate information is provided within the parameter data structure; hence, there is no need for a separate initialization routine for the 5.3 or 6.3 rate encoder.

To start the encoding process of a channel, refer to the *Wrapper API* section of this application report.

6.1.2 Decoder

The decoder is called from the file lbccodec.c for the stand-alone version, or the file component.c for the multichannel system version. The calling function must first initialize the decoder before calling the Decoder_G723() function to start the decoding process of a new channel.

The function Init_DecodG723( ) is called one time before the decoder to initialize the static variables and arrays for each channel. The rate information is provided within the parameter data structure; hence, there is no need for a separate initialization routine for the 5.3 or 6.3 rate decoder.

To start the decoding process of a channel, refer to the *Wrapper API* section of this application report.

6.2 Test and Validation

The code was fully tested against the test vectors provided by the ITU specification.

6.3 Multichannel Demonstration

The TMS320C62x DSP is capable of processing a number of channels of this speech coder, as well as several other coders. The multichannel vocoder technology demonstration kit (MCV TDK), which is a multichannel/algorithm real-time system, has been built and runs on the TMS320C62x EVM to demonstrate this algorithm and others running in real time. To learn more about this system, see the application reports written by Xiangdong Fu and Zhaohong Zhang [3],[4],[5].

The Demonstration program displays the icon for the 5.3 rate version of the G.723.1 coder. To gain access to the alternate rate (6.3 kbps), you need to reconfigure the channel(s) via a dialog box that shows both rates [6].

Windows is a registered trademark of Microsoft Corporation.

7 Performance / Code Size Results

Table 9 summarizes the estimated number of bytes that is required for the program memory and data memory of the G.723.1 vocoder. In this table, all the numbers are in bytes, and N represents the number of channels running on the same device.

Table 9. Program and Data Memory Requirements of G.723.1

Function	Context Data	Stack 5.3 rate	Stack 6.3 rate	Tables	Program Memory
Encoder	1476 x N	NA	NA	NA	NA
Decoder	420 x N	NA	NA	NA	NA
Common				NA	NA
Total	**1896 x N**	**8176**	**3328**	**18992**	**68800**

Seventeen assembly functions, which were substituted for the equivalent functions in the original C code, are listed in Table 10.

Table 10. Assembly Functions and Their Associated C Module

ASM functions	C Module	ASM functions	C Module
Estim_pitch	Exc_lbc	Upd_ring	Lpc
Find_acbk	Exc_lbc	Comp_Lpc	Lpc
Comp_info	Exc_lbc	Error_Wght	Lpc
G723Cor_h_X†	Exc_lbc	Lsp_svq	Lsp
Filt_Lpf	Exc_lbc	AtoLsp	Lsp
Get_Rez	Exc_lbc	Comp_En	Util_lbc
D4i64_LBC	Exc_lbc	Ser2Par	Util_lbc
Comp_ir	Lpc	Vec_Norm	Util_lbc
Sub_ring	Lpc		

† The name of Cor_h_X was changed to G723Cor_h_X to avoid conflicting with functions of the same name in other ITU coders.

The implementation of the assembly functions above yielded the performance shown in Table 11 and Table 12. The assembly optimizations included the following:

- Improved register usage
- Improved multiplier usage
- Read constants once
- FIR and IIR values kept in registers to avoid memory update
- Improved functional unit utilization
- Optimized search with dual computations in parallel.

Table 11 shows the relative cycles required for the encoder versus the decoder at either rate. The lower-bit-rate coder shows a 3:1 ratio of encoder versus the decoder, while the higher-bit-rate coder shows a 7:1 ratio of encoder-to-decoder cycles. This cycle count is based on the G.723.1 coder being compiled with version 3.00 of the code generation tools and runs on the simulator.

Table 11. Cycle Estimations for G.723.1 Coder

	Simulator Cycles	
Function	**5.3 rate**	**6.3 rate**
Encoder	281796	737424
Decoder	99435	101290
Total	**381231**	**838714**

Table 12 shows the results of running the current implementation of the coder in the multichannel system.

Table 12. G.723.1 Performance Observed on Multichannel System

Encoder + Decoder	Number of Channels	MHz
5.3 rate (ACELP)	10 @ 87% of 160MHz C6x	13.9
6.3 rate (MP-MLQ)	6 @ 87% of 160MHz C6x	23.1

The performance level shown here is certainly not the limit for this coder on the TMS320C62x. Performance improvements will continue to be made on the G.723.1 coder. Contact Texas Instruments for the latest implementation available for this coder.

8 References

1. *ITU-T Recommendation G.723.1 – Dual Rate Speech Coder for Multimedia Communications Transmitting at 5.3 and 6.3 kbit/s*, March 1996.
2. *ITU-T Recommendation G.723.1 – Annex A: Silence Compression Scheme*, November 1996.
3. Xiangdong Fu and Zhaohong Zhang, *A Multichannel/Algorithm Implementation on the TMS320C6000 DSP*, SPRA556.
4. Xiangdong Fu and Zhaohong Zhang, *TMS320C6000 Multichannel Vocoder Technology Demonstration Kit Host Side Design*, SPRA558.
5. Xiangdong Fu, *TMS320C6000 Multichannel Vocoder Technology Demonstration Kit Target Side Design*, SPRA560.
6. *DSP Algorithm Integration Standard (XDAS) (Rules and Guidelines)*, SPRU352.
7. *eXpressDSP Algorithm Standard (API Reference)*, SPRU360.
8. Stig Torud, *Making DSP Algorithms Compliant With the eXpressDSP Algorithm Standard*, SPRA579.
9. Carl Bergman, *Using the eXpressDSP Algorithm Standard in a Static DSP System*, SPRA577.
10. Carl Bergman, *Using the eXpressDSP Algorithm Standard in a Dynamic DSP System*, SPRA580.

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE ("CRITICAL APPLICATIONS"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.