Application Report
SPRA565B - February 2000

GSM Enhanced Full Rate Speech Coder: Multichannel TMS320C62x Implementation

Min Wang
Xiangdong Fu
C6000 Applications

ABSTRACT

This document provides a detailed description of the implementation of the GSM enhanced full rate (EFR) speech encoder and decoder (codec) on the Texas Instruments (TI™) TMS320C62x digital signal processor (DSP). Topics include program structure, code writing rules, data/program memory requirement, and performance evaluation. Issues on multichannel implementation and interrupts are also addressed.

Contents

List of Figures

List of Tables

1 Introduction

This document provides a detailed description of the implementation of the GSM enhanced full rate (EFR) speech encoder and decoder (codec) on the TMS320C62x DSP. Issues on multichannel implementation and interrupts are also addressed.

The EFR speech codec is defined by the European Telecommunications Standards Institute (ETSI) specifications GSM 06.51, 06.60, 06.61, 06.62, and 06.82. A reference C program is provided in GSM 06.53, along with a set of testing sequences. The code implemented on the C62x is fully bit compatible with the reference code on all testing sequences.

The GSM EFR speech codec uses the algebraic code excited linear prediction (ACELP) algorithm, which is an analysis-by-synthesis algorithm and belongs to the class of speech coding algorithms known as code excited linear prediction (CELP). Therefore, the speech decoder is primarily a subset of the speech encoder.

For every 20-ms speech frame, the EFR encoder extracts 57 parameters, as well as VAD information. These parameters include short-term (LP) parameters, adaptive excitation (LAG) parameters, and algebraic code excitation (CODE) parameters.

Each speech frame is equally divided into four subframes. Except for LP parameters, which are extracted once per frame and therefore are frame parameters, the parameters are all subframe parameters. These parameters are quantized into 244 bits, resulting in a transmission rate of 12.2 kbits/s. In the EFR decoder, the quantized parameters are decoded and a synthetic excitation is generated by adding the adaptive and innovative codevectors scaled by their respective gains. The synthetic excitation is then filtered through the LP synthesis filter to generate the synthetic speech waveform, which is processed by a so-called post-filter to further improve the speech quality.

Refer to ETSI specification GSM 06.60 [1] for a detailed description of GSM EFR vocoder.

2 Multichannel System

The multichannel system can run more than one algorithm at the same time. Any algorithm compliant with the eXpressDSP™ Algorithm Standard[5][6][7][8][9] is capable of multichannel processing. An XDAIS compliant algorithm requires three functional modules; initialization, freeing and kernel. The kernel module performs the algorithm processing while the initialization and freeing module initializes and frees the algorithm context data[2].

The XDAIS application programming interface (API) for the initialization is defined as

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

The XDAIS application programming interface (API) for freeing is defined as

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

where the first parameter of type IALG_Handle is a pointer that contains the current context data memory location. During framework initialization, the framework calls the algorithm initialization functions. All the initialization functions perform the same tasks: initialize context data and store to the desired memory location. After the initialization function is completed, the system repeatedly calls an algorithm(s) until all the frames 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 done.

The APIs for the called algorithm, encoder and decoder, is defined as

```
extern DAIS_Int8 EFRE_TI_encode(IEFRENC_Handle handle, DAIS_Int16* in,
DAIS_Uint16* out);
extern DAIS_Int8 EFRD_TI_decode(IEFRDEC_Handle handle, DAIS_Uint6* 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. Since XDAIS does not specify standard API's for algorithm processing, these two APIs are algorithm specific.

3 Algorithm Description

The complete speech codec is implemented in mixed C and TMS320C2x assembler. Each basic math operation defined in basic_op2.c either is replaced by its corresponding TMS320C62x intrinsic or is static inlined. All of the 32-bit functions in oper_32b.c are expressed in intrinsics and are static inlined.

The codec is multichannel enabled. After processing one frame of one channel, the codec can process one frame of another channel. Additional effort is required to make the codec interruptible within a frame processing to allow the decoder higher priority than the encoder and/or to timely deliver the parameters to the next stage of the whole process.

Features, including voice activity detection (VAD), discontinuous transmission (DTX), error concealment, and comfort noise generation are implemented.

TI's implementation of this algorithm is fully bit-compatible with the bit-exact Reference C code, version 5.1.0 (GSM 06.53) on all testing sequences.

Both encoder input and decoder output are 13-bit PCM values.

The interface is a 244-bit stream to the preliminary channel coding stage.

3.1 Encoder Structure

The GSM EFR speech encoder is divided into five submodules:

- Submodule 1 contains pre-processing, the LPC analysis, and LPC to LSP conversion, including Autocorr(), Lag_window(), Levinson(), and Az_lsp().
- Submodule 2 calls Q_plsf_5() to conduct LSP quantization.
- Submodule 3 generates the interpolated LPC parameters, computes weighted speech, and finds the open-loop pitch, including Int_lpcs(), Int_lpc(), Weight_Ai(), Residu(), Syn_filt(), and Pitch_ol().
- Submodule 4 performs the adaptive codebook search, calling Pitch_fr6(), Enc_lag6(), Pred_lt_6(), Convolve(), G_pitch() and q_gain_pitch().

- Submodule 5 performs the innovative codebook search and filter memory update, calling code_10i40_35bits(), G_code(), q_gain_code(), and Syn_filt().

Submodules 1 through 3 are for frame processing and should be done once per frame. Submodules 4 and 5 are for subframe processing and should be repeated four times per frame.

3.2 Decoder Structure

The GSM EFR speech decoder is divided into two submodules.

- Submodule 1 deals with frame parameter recovering, which should be done once per frame.
- Submodule 2 contains subframe parameter recovering, speech synthesizing, and post-filtering. Submodule 2 should be repeated four times per frame.

3.3 Main Program

Figure 1 shows the file tree of the GSM EFR speech codec software package.

```
main  ---------  code --------- common ---------------- functions
  |                  |                           |
             |                |----- encoder              |------  intrinsics, inlined ma
             |                |                                       tables & constant
             |                |----- decoder
             |
             |----- test  --------- enc
                              |---- dec
                              |---- run_evm
```

Figure 1. File Tree of the GSM EFR Speech Coder Software Package

The code directory contains the codec.c file that is the top file of the whole codec; the linker command file, codeclnk.cmd; the make file, makefile; and the simulator initialization file, siminit.cmd.

The common directory contains the include files, *.h; the table files, *.tab; and the source files (in *.c or *.asm) common to both the encoder and decoder.

The encoder directory contains the encoder top file, encoder.c, and the include files, *.h, that are only used in the encoder. The program follows the structure of the encoder described above. The files specific to one submodule are stored in the corresponding subdirectory.

The decoder directory contains all of the files used in the decoder only.

The test directory contains some test files to run the encoder and the decoder, with a simulator or an EVM (evaluation module).

4 Coding Guidelines

This program is a mixed C and assembly implementation.

4.1 Variable, Array, and Pointer Names

We use the same name conventions for variables, arrays, and pointers as in the reference C code.

4.2 Math Operations

All math operations defined in basicop2.c and oper_32b.c in the reference C program are replaced by TMS320C62x intrinsics, if possible. In addition to the usage of intrinsics, basic operations div_s and negate and all the math functions defined in oper_32b.c are inlined. Their expressions in intrinsics or inlined versions reside in basic_op.h and oper_32b.h, respectively. For a description and usage of TMS320C62x intrinsics, see the *TMS320C62x/67x Programmer's Guide* (SPRU198) [10].

4.3 Tables and Functions

The names of tables are the same as in the reference C program, except for a few tables named *t* in the reference C program. To eliminate ambiguity and help the memory management of and accessibility to these tables, a unique name is provided for each of these tables. For example, the table used in function Inv_sqrt is renamed t_inv_sqrt. See for the names of all of the tables in this codec.

The functions are sorted in the following three categories based on their functionality in the vocoder: encoder, decoder, and common functions.

- All of the functions belonging to the encoder only are located in the encoder directory.
- All of the functions belonging to the decoder only are located in the decoder directory.
- All the functions used by both the encoder and the decoder are placed in the common directory.
- Each function is either in C or assembly. The name of a function closely matches its name in the reference C program. For a function named * in the reference C program:
  - If it is in C, the function is named *.c in this implementation.
  - If it is in hand-optimized assembly, the function is named *_ho.asm in this implementation.

4.4 Interrupt Issues

The codec is multichannel enabled in the sense that, after the process of one frame of one channel, the code can process one frame of another channel. That is, the codec is interruptible at the frame process boundary. Additional study is required to make the codec interruptible within a frame process.

4.4.1 Interrupts at the Frame Boundary

To ensure access to the correct memory of each channel, the static variables and arrays with values that must be kept from one frame to the next are sorted into one of the two structures, EFR_Sta_Encoder (for the encoder) and EFR_Sta_Decoder (for the decoder). Both of these two structures are defined in EFR_StaMem.h. The static variables and arrays of the codec of each channel are initialized with the standard IALG initialization functions efreInit() and efrdInit(), respectively.

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

```
DAIS_Int8 EFRE_TI_encode(IEFRENC_Handle handle,DAIS_Int16* in,
DAIS_UInt8* out)
```

where the pointer handle denotes the start address of the context memory block that contains the structure defined as EFR_Sta_Encoder 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.

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

```
DAIS_Int8 EFRD_TI_decode(IEFRDEC_Handle handle,DAIS_UInt8* in,
DAIS_Int16* out)
```

where the pointer handle denotes the start address of the context memory block that contains the structure defined as EFR_Sta_Decoder 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.

4.4.2 Interrupts at the Submodule Boundary

You can interrupt the codec anywhere other than software pipelined loops. (Of course, you can even choose not to interrupt the codec until the end of a frame process as long as the overall system design requirement is met.) If interrupts are necessary, we suggest that, in general, interrupts occur in certain places, like the end of each submodule, so that the increases in data memory and cycle counts are kept minimal.

5 Memory Requirements

The data memory is divided into three groups:

- Context data
- Tables
- Local variables and arrays

5.1 Memory for Context Data

The context data are the static variables and arrays with values that must be kept from one frame to the next. and summarize the encoder/decoder context data names and their sizes. The context data for the GSM EFR codec are sorted into one of the two structures, EFR_Sta_Encoder and EFR_Sta_Decoder.

Table 1. EFR Encoder Context Data Structure (EFR_Sta_Encoder)

Array	Size (Bytes)
gain_code_old_tx	56
lsp_old	20
lsp_old_q	20
lsf_old_tx	140
lsp_old_q	20
lsf_p_CN	20
L_sacf	108
L_sav0	144
mem_err	20
mem_syn	20
mem_w0	20
old_A	24
old_CN_mem_tx	12
old_exc	308
old_speech	480
old_wsp	286
past_qua_en	8
past_r2_q	20
pred	8
pre_pro	12
rvad	18
static_vad	24
Total	**1780**

Table 2. EFR Decoder Context Data Structure (EFR_Sta_Decoder)

Array	Size (Bytes)
gain_code_old_rx	56
gbuf	10
lsp_old_dec	20
lsf_old_rx	140
lsf_new_CN	20
lsf_old_CN	20
lsf_p_CN_dec	20
mem_syn_dec	20
mem_syn_pst	20
old_exc_dec	308
past_qua_en_dec	8
past_r2_q_dec	20
past_lsf_q	20
pred_dec	8
pbuf	10
Total	**700**

5.2 Memory for Tables

Table 3 summarizes the tables in the source code. Tables common to both encoder and decoder take 8298 bytes of data memory. If you run encoder only, 9668 bytes of data memory are required for the tables. If you run decoder only, 8670 bytes of data memory are required. If you run both encoder and decoder on the same C62x chip, 9914 bytes of data memory are required. Notice that the memory for tables is independent of the number of channels running on the chip.

Table 3. List of GSM-EFR Tables

Table	Size (Bytes)	Encoder	Decoder
ab	10	x	
cdown	14		x
cos	130	x	x
dhf_mask	114		x
dgray	16		x
dico1_lsf	1024	x	x
dico2_lsf	2048	x	x
dico3_lsf	2048	x	x
dico4_lsf	2048	x	x
dico5_lsf	512	x	x
f_gamma1	20	x	
f_gamma2	20	x	
f_gamma3	20		x
f_gamma4	20		x
gray	20	x	
grid	122	x	
inter_6	50	x	
inter_factor	48		x
t_inv_sqrt	98	x	x
lag_h	20	x	
lag_l	20	x	
t_log2	66	x	x
lsp_old	20	x	x
mean_lsf	20	x	x
t_pow2	66	x	x
pred_6	122	x	x
qua_gain_code	64	x	x
qua_gain_pitch	32	x	x
slope	128	x	
window_160_80	480	x	
window_232_80	480	x	
Total	**9914**	**9668**	**8670**

5.3 Memory for Local Variables and Arrays

The local variables and arrays are stored in the stacks and consume about 4970 bytes and 1376 bytes of data memory space for the encoder and decoder, respectively.

5.4 Program Memory

The codec is mixed C and assembly. The overall hand coded assembly routines, total of 19 routines, require 12576 bytes of program memory. All of these 19 routines are called in the encoder and only 8 of them, requiring 2208 bytes, are called in the decoder. With 3.00 compiler, the overall code sizes of the rest C routines of the encoder and the decoder takes 32976 bytes and 17632 bytes of program memory, respectively. The overall code size of the GSM EFR codec with the compiler version 3.00 release, is 56832 bytes.

Table 4 summarizes the estimated number of bytes required for the GSM EFR vocoder program memory and data memory. All of the numbers are in bytes. The *N* in the table represents the number of channels running on the same device.

Table 4. Program and Data Memory Requirements of the GSM-EFR

Function	Static RAM	Dynamic RAM	ROM Table	Program Memory (3.00 Compiler)
Encoder	1780 x *N*	4970	9668	45572
Decoder	700 x *N*	1376	8670	19842
Total	**2480 x *N***	**4970**	**9914**	**56832**

6 Performance Results

Because the current implementation of EFR is a mixed C and C62x assembly, the cycle count depends on the compiler used.

Table 5 gives the worst-case cycle count of one GSM EFR channel with the compiler version 3.00 release. The testing sequence is the test20 in the GSM EFR testing vectors, with both dtx and post-filter enabled. The cycle count presented in the table is obtained by adding the worst-case cycle count with the version 2.00 fast simulator and 10% overhead caused by possible memory bank hits. (Because the program memory required by GSM EFR is approximately ~56Kbytes, the code can fit into any C62x on-chip program memory.)

Table 5. Performance Estimation of GSM-EFR With C62x DSP Compiler Version 3.00

Function	MHz
Encoder	9.09
Decoder	1.60
Total	**10.69***

7 References

1. European Telecommunications Standards Institute (ETSI), *GSM—Enhanced Full Rate Specifications 06.51, 06.60-63and 06.82*.
2. Xiangdong Fu and Zhaohong Zhang*, A Multichannel/Algorithm Implementation on the TMS320C6000,* SPRA556.
3. Xiangdong Fu and Zhaohong Zhang*, TMS320C6000 Multichannel Vocoder Technology Demonstration Kit Host Side Design,* SPRA558.
4. Xiangdong Fu, *TMS320C6000 Multichannel Vocoder Technology Demonstration Kit Target Side Design*.
5. *eXpressDSP Algorithm Standard (Rules & Guidelines)*, SPRU352.
6. *eXpressDSP Algorithm Standard (API Reference)*, SPRU360.
7. Stig Torud, *Making DSP Algorithms Compliant to the eXpressDSP Algorithm Standard*, SPRA579.
8. Carl Bergman, *Using the eXpressDSP Algorithm Standard in a Static DSP System,* SPRA577.
9. Carl Bergman, *Using the eXpressDSP Algorithm Standard in a Dynamic DSP System*, SPRA580.
10. *TMS320C62x/67x Programmer's Guide*, SPRU198.