Texas Instruments

GSM Half-Rate Voice Coding on the TMS320C62xx DSP

Application Report

September 1999

Wireless ASP Products

SPRA582

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.

Contents

List of Figures

List of Tables

List of Examples

GSM Half-Rate Voice Coding on the TMS320C62xx DSP

ABSTRACT

This document describes the GSM half-rate voice coding implementation on the TMS320C62xx processor (also called the vocoder). It discusses the system requirements and gives two levels of detail for the subroutines. The first level is designed for programmers who want to use the code in a basic GSM architecture. The second level is for programmers who want to see how the code is implemented for modifying it, using it as the basis for optimizing code for other algorithms, or for porting it to enhanced members of the TMS320C62xx processor family.

1 Introduction

Two versions of the software are available: one with hand-optimized assembly code and one without hand-optimized assembly code. Unless otherwise stated, this document discusses the hand-optimized version of the code, although data on the C code with intrinsics version is included for comparison purposes.

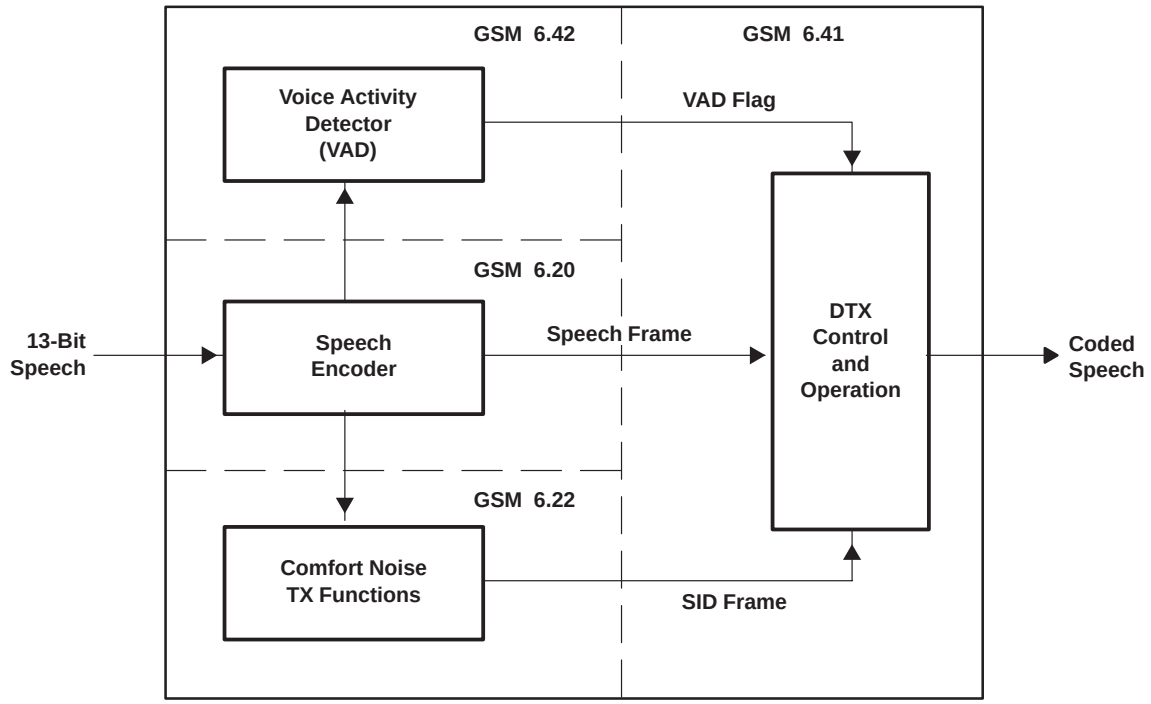
Wherever possible, the nomenclature in the code is that used in the standard C code provided by the European Telecommunications Standards Institute (ETSI). Most variable names are maintained. Some small structures have been split into their component parts to simplify passing parameters; in these cases, the variable name is made up of both parts of the original name.

This document discusses implementation when there are significant optimizations in the TMS320C62xx implementation of the standard ETSI version. It does not describe the Global System for Mobile (GSM) half-rate (HR) voice-coding algorithm. For more detailed information on the voice-coding algorithm, see the relevant ETSI documents, ETSI 300 581–(1…8).

1.1 Implementation Functions

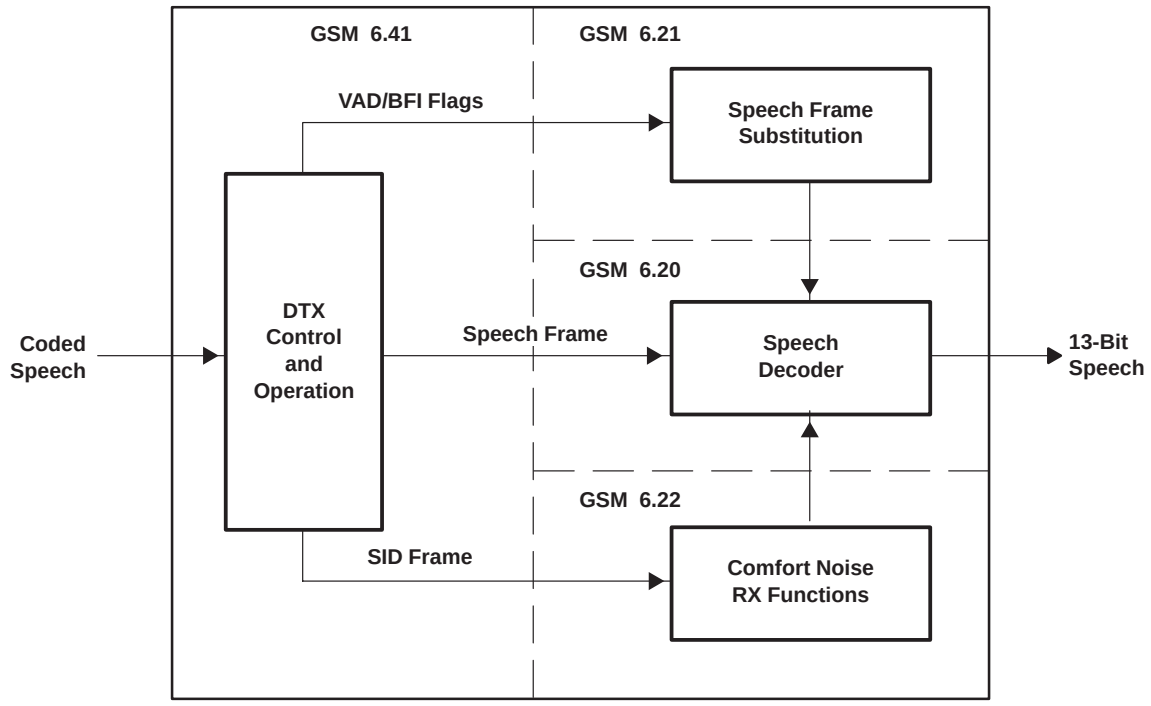
This implementation includes all of the half-rate voice coding functions for the GSM half-rate voice coder and decoder and supports these GSM specifications: 6.02, 6.20, 6.21, 6.22, 6.41, and 6.42. Figure 1 and Figure 2 illustrate the voice coder and decoder, respectively, and show how these modules interact to provide the voice coding and decoding functions.

The voice code is based on the ANSI C code in GSM specification 6.06 that is optimized for the TMS320C62xx processor and has been verified with the tests specified in GSM specification 6.07.



NOTES: A. SID = Discontinuous transmission
B. DTX = Discontinuous transmission

Figure 1. GSM Half-Rate Speech Coder



NOTE: BFI = Bad frame indication

Figure 2. GSM Half-Rate Speech Decoder

1.2 Typical Implementation

A typical implementation in a GSM base station is shown in Figure 3.

The TMS320C6202 processor performs all of the vocoding (coding + decoding) for a full GSM half-rate radio frequency (RF) of 16 timeslots. Alternatively, it may be grouped with other GSM voice coders to produce a multistandard base station. The frame format for inputting and outputting GSM- and PCM-coded (pulse code modulation) data depends on the application and is not covered in this document. For more information on typical applications using the serial and direct memory access (DMA) ports for the vocoder, see the TI *TMS320C6201/6701 Peripherals Reference Guide*.

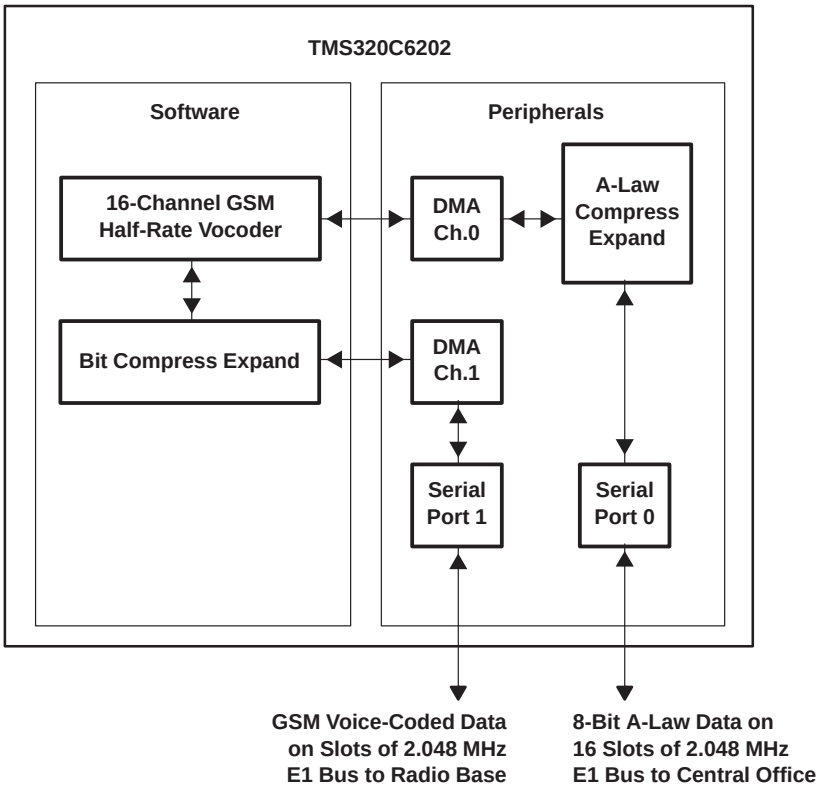


Figure 3. Typical GSM Base Station Implementation

2 System Requirements

C code from all the C modules must be compiled with the –mt –mh –o3 options. In addition, the correct compiler option must be used to force the compiler to do data page pointer (DP) relative addressing for array variables. The C compiler versions supported by this system are shown in Table 1.

Table 1. C Compiler Versions Supported

C Compiler Version	Option
1.0.2 beta	–mt –mh –o3 –mc
1.0.1	–mt –mh –o3 –mc
2.1 beta	–mt –mh –o3
2.1	–mt –mh –o3
3.0	–mtw –mh2147483647 –mr1 –o3

Some previous versions of the C compiler do not support some of the functions used by this C code and are therefore obsolete. The code has not been tested with versions above 3.0.

2.1 Memory Allocation

Memory is allocated at link time for a single channel of vocoding (coding + decoding). Additional memory can be allocated on the heap for additional channels by calling the following routine:

```
int** address[channels] HR_Assign_GSM(channels)
```

Where:

The input parameter is the total number of channels required, and it returns a pointer to an array of channel context addresses.

These addresses may be used by the routine Set_Channel (address) to switch context to that channel before any of its variables are called, reset, or changed. The first channel allocated in this array is the one linked to the .bss section. If only one channel is required, this routine need not be called, and all calls to Set_Channel (address) and Return_Channel () can be removed.

If there is insufficient heap memory for either the array of pointers or the channel data space, some or all of the pointers will be NULL. You must ensure that there is sufficient heap space at compile time or check the values returned for error conditions. The example code allows enough heap space for the C I/O and 16 channels of GSM at half rate.

2.2 Channel Switching and Interrupts

Channel switching for special operations is done by calling the routines Set_Channel (address) and Return_Channel (). Channel switching is handled automatically by the main coder and decoder routines. It is, however, necessary to handle cache enabling/freezing and interrupt disabling/enabling around the coder and decoder routines. As context is switched via the DP register, interrupts must be disabled because the DP register may point to a memory block other than that expected by a C interrupt service routine. All hand-coded assembly routines assume that interrupts are already disabled. The Handle_Ints routine may be added anywhere into the C code to provide an interrupt window. (This routine adds a few cycles each time that it is called).

2.3 Execution Time Benchmarks

TI recommends that the C code with hand-optimized assembly code be used on either the TMS320C6202 or TMS320C6203 platform. This approach allows a single TMS320C6202 processor to handle a full GSM timeslot of 16 half-rate channels. Table 2 shows calculated loading for some other configurations.

The execution time benchmarks were obtained with Compiler 2.1/Simulator 2.0 and are measured from the label HR_SpeechEncoder1 or HR_SpeechDecoder1 to the return address from the last subpart. The cache is enabled during speech vocoding but is disabled during file I/O. These benchmarks do not include the C I/O file input or output because in a real system, the serial ports working in conjunction with the DMA channels would probably handle this, and the cycle count would be much less than if C I/O were used.

Table 2. Execution Times

Processor	Channels	MHz	Notes
TMS320C6202 C+asm	1	12.50	1
TMS320C6202 C only	1	21.67	2
TMS320C6201 (Sequence A) C+asm	16	193	3
TMS320C6201 (Sequence B) C+asm	16	188	4
TMS320C6202 C+asm	16	180	1
TMS320C6202 C only	8	174	2

NOTES:
1. TMS320C6202 C+asm has all memory internal.
2. TMS320C6202 C uses the C code without hand-optimized assembly code.
3. TMS320C6201 (Sequence A) is in cached SBSRAM. Data is in internal data memory. The code and decode sequences are called alternately.
4. TMS320C6201 (Sequence B) is in cached SBSRAM. Data is in internal data memory. First the code sequences for all channels are called and then the decode sequences for all channels are called. This gives a lower cache miss rate.

2.4 Program Memory Requirements

The GSM half-rate program is 84576 bytes (C is only 87616). In addition, 17248 bytes are used by the sample C code that reads and writes test vectors using the simulator C I/O functions. Of the GSM half-rate code, 11680 bytes are hand-optimized assembler code (approximately 14%). In terms of millions of instructions per second (MIPS), this equates to approximately 57%. Further optimization of the remaining C code is possible, but is not currently planned. This code is fully re-entrant, and no increase in code size for the algorithm itself is required, although the control program is more complicated. The control program fits inside the memory space of the TMS320C6202 processor and requires no external program memory.

2.5 Data Memory Requirements

The space required by the data memory constants is independent of the number of channels implemented, whereas the space required for channel variables is a multiple of the number of channels required (see Table 3). Other areas are based on the sample code shipped with the algorithm and are implementation-dependent. A 23-channel vocoder fits in the internal data memory of a TMS320C6202 processor; an 8-channel vocoder fits into a TMS320C6201 processor.

Table 3. Data Memory Requirements

Function	Size (Bytes)
Stack	8192
Constants	16129
Channel variables	4576*n
C I/O variables (from sample code)	3800

3 Top-Level Routines

The top-level routines that you need to control the GSM half-rate voice coder/decoder using the GSM half-rate code are discussed in this section.

3.1 main

The C program main is used to apply the test patterns via the simulator to the GSM half-rate voice coder. This program can be used as an example to generate real application code or to verify the code. It can be found in the gsm.c file. This particular code is for a 4-channel voice coder/decoder, which automatically runs all the test vector files on a limited number of channels. The number of channels is determined from #define ChMax near the beginning of the file. I/O-specific routines for this form of I/O are in the host.c file. Applications that do not use file I/O can safely remove the host.c file from that application because this code is not called elsewhere. All code not in the gsm.c file or in the host.c file is part of the standard and must be compiled and linked for all applications.

3.2 HR_Assign_GSM

This subroutine defines and assigns memory for the voice coders and decoders. If more than one channel is being used, then the following subroutine must be called before any other GSM half-rate routine:

```
int**= HR_Assign_GSM(Channels);
```

Where:

Channels is the number of channels required, and int** is of type int* *ChannelArray.

On return, ChannelArray contains an array of address pointers the same size as that requested. In single-channel applications, this call can be skipped. The code is found in the homing.c file. When a NULL value is returned in any of the pointers, then there is insufficient heap memory available.

3.3 HR_SpeechEncoder1...5

The following subroutines handle the encoding of a speech frame to the GSM half-rate standard:

```
HR_SpeechEncoder1 (*Channel, *Input, *Output[0]);
HR_SpeechEncoder2 (*Channel, *Input, *Output[6]);
HR_SpeechEncoder3 (*Channel, *Input, *Output[9]);
HR_SpeechEncoder4 (*Channel, *Input, *Output[12]);
HR_SpeechEncoder5 (*Channel, *Input, *Output[15]);
```

Where:

Channel is the address of the data space for this channel of voice coding and decoding.

Input is the address of a frame of expanded GSM variables.

Output is the address to put the decoded frame of 160 samples of 13-bit 2s complement linear speech output.

Set_Channel and Return_Channel are handled within this routine and should not be called. In single-channel applications, Channel should be __bss__ (a TMS320C62xx compiler constant). The code is found in the sp_dec.c file. The GSM half-rate parameters are in an uncompressed array of shorts. Data unpacking is not performed, and interrupts must be disabled before these routines are called.

3.4 HR_SpeechDecoder1...4

The following subroutines handle the decoding of a speech frame to the GSM half-rate standard:

```
HR_SpeechEncoder1 (*Channel, *Input, *Output[0]);
HR_SpeechEncoder2 (*Channel, *Input, *Output[40]);
HR_SpeechEncoder3 (*Channel, *Input, *Output[80]);
HR_SpeechEncoder4 (*Channel, *Input, *Output[120]);
```

3.5 Set_Channel

This subroutine sets the DP register to point to a specific channel. It is not usually required except at initialization, when some initialization of mode-specific variables relating to the DTX mode is required. The following subroutine may be called before channel-specific modes are examined or changed:

```
Set_Channel (*Channel);
```

Where:

Channel is the address of the data space for this channel of voice coding/decoding.

This subroutine is found in the homing.c file and is defined in the homing.h file. Interrupts must be disabled before this subroutine is called, and they must remain disabled until after Return_Channel is called.

3.6 Return_Channel

The following subroutine complements Set_Channel by providing a means to restore the DP register to its C defaults after channel-specific modes are examined or changed:

```
Return_Channel ();
```

This subroutine is an inline compilation and can be found in the homing.h file.

3.7 HR_ResetEnc Encoder

Typically, the encoder is reset when a special homing frame that automatically resets it is sent. You may bypass this mechanism by calling the following subroutine directly:

```
HR_ResetEnc(int* Context,int DTXenable)
```

Where:

Channel is the address of the data space for this channel of voice coding/decoding.

DTXenable selects if DTX is enabled (1) or disabled (0).

Set_Channel and Return_Channel are handled within this routine and should not be called. In single-channel applications, Channel should be __bss__ (a TMS320C62xx compiler constant). This code is found in the homing.c file and is defined in the homing.h file. Use of the homing frame ensures that both ends of the vocoding link are reset at the same point.

3.8 HR_ResetDec Decoder

Typically, the decoder is reset when a special homing frame that automatically resets it is sent. You may bypass this mechanism by calling the following subroutine directly:

```
HR_ResetDec(int* Context)
```

Where:

Channel is the address of the data space for this channel of voice coding/decoding.

Set_Channel and Return_Channel are handled within this routine and should not be called. In single-channel applications, Channel should be __bss__ (a TMS320C62xx compiler constant). This code is found in the homing.c file and is defined in the homing.h file. Use of the homing frame ensures that both ends of the vocoding link are reset at the same point.

3.9 Multichannel Setup

Example 1 shows how to use some of the previously discussed routines.

Example 1. Using a Subroutine

```
if (!strcmp(DTXmode,"dtx"))
{
   HR_ResetEnc(Channels[ChCount],1);
   HR_ResetDec(Channels[ChCount]);
}
else
{
   HR_ResetEnc(Channels[ChCount],0);
   HR_ResetDec(Channels[ChCount]);
}
```

This code comes from the gsm.c file. It performs the initialization of the DTX mode for a new test vector file and resets the encoder and decoder at the start of a new test pattern sequence.

4 Linker Notes

When you compile and link code, note that the combinations for the best optimization results all involve compromises. In particular, using the –pm (program mode) flag produces a smaller program that executes in fewer cycles on the fast simulator; the cycle count is higher on the slow cache accurate simulator. This is due to an increase in cache misses because of overlapping program areas, and is solved by compiling without the –pm option and by the linker allocating modules in a more efficient order. This approach is recommended for the TMS320C6201 processor.

The TMS320C6202 processor, however, has more on-chip program space. Because the entire program fits into internal memory, it runs faster if the –pm option is used with default linking.

4.1 Ink.cmd (TMS320C6201 Optimum)

The linker defines two variables that allow allocation of the memory for external variables (see Example 2). These variables must be defined before and after the .bss section of the homing.obj file; otherwise, memory for additional channels is incorrectly allocated.

Example 2. Allocating Memory for External Variables (TMS320C6201)

```
_homing_start = .;  /* used to allocate heap for additional
channels */

homing.obj (.bss)

_homing_end = .; /* used to allocate heap for additional channels */
```

The order in which the .text sections are specified has been optimized to reduce the risk of cache misses with the TMS320C6201 processor. The benefits of specifying addresses that are less likely to conflict in the cache to the linker outweigh the gain obtained by using –pm with the compiler. Each module should be compiled separately using the –mt –mh –o3 options.

4.2 Partial_Link_HR.cmd

This link command file should be used in implementations with multiple voice coders. This command links the routines together in a single module that can be linked again later. The global definitions are reduced to the minimum required. Only global definitions beginning with HR_ remain. This command can also be used to link code from multiple sources.

4.3 Code Composer Make Files

The CC_GSM_HR.MAK and CC_GSM_HR_PL.MAK code composer make files can be used to build a standalone half-rate voice coder or a reusable library module, respectively.

5 Simulator Notes

This section briefly discusses the simulator functions:

- Simfast.simcmd

  This simulator command file is used for vector verification.

- Simslow.simcmd

  This simulator command file is used for benchmark evaluation.

- Simprofile.simcmd

  This simulator command file is used for profiling and to determine where to optimize. If this simulator command file is used, the code must also be compiled with the –mg option.

- Cont_Dual.gsm

  This simulation file tells the C I/O how to handle all of the test pattern files. For each test sequence, there are six lines; the first four lines are the filenames of the input and output files, and the fifth line indicates if that code sequence is to be run with discontinuous transmission (DTX) enable or disabled; (dtx or nodtx, respectively). The last line indicates the GSM data rate; this code is always HALF.

  ```
  PCM Data Input File
  GSM Data Output File
  GSM Data Input File
  PCM Data Output File
  dtx | nodtx
  HALF
  ```

  This structure is repeated for the number of test sequences that must be run; when the test sequences finish, the C program exits.

6 Software

Optimization is always a compromise between different objectives. The primary considerations and the approximate weight of each is shown in Table 4.

Table 4. Weight of Optimization Considerations

Consideration	Weight
Minimum MIPS	60%
Minimum per-channel data memory	30%
Minimum program memory	5%
Minimum all-channel data memory	5%

6.1 Primary Optimizations

The primary optimizations include:

- Replace all calls to ETSI functions defined in mathhalf.c with TMS320C62xx C code intrinsics. For example, sadd (x,y) => _sadd(x<<16,y<<16)>>16. In practice, when combined with other optimizations, this often becomes _sadd(total, y<<16) with the total realigned outside the loop.
- Replace all functions in mathdp31.c with inline subroutines except divide_s which is in hand-optimized assembly code. These are all very short subroutines that are called regularly. Saving is mainly in the subroutine call and return, with a better register in the calling routine because it uses very few variables.
- All look-up ROMs are forced to the .const section. This forces the compiler to use absolute addressing instead of DP (.bss) addressing. The primary advantage of this is in multichannel implementations because constants are global to all channels, which leaves .bss for variables.
- All channel-specific context variables are relocated to the homing.c .bss section for online context switching by changing the DP register. The compiler uses DP offset addressing. This is primarily for multichannel operation.
- All local temp variables are stack-based register or stack pointer (SP) offset addressing. This is primarily for multichannel operation.
- Local shorts are redefined as int. to reduce compiler masking of unused bits. This takes two forms, depending on whether the variable can saturate or not. If the variable cannot saturate, it compiles best in the default data size. If the variable can saturate, then it may be better to saturate it in the upper 16 bits; for example, sadd (x,y) => _sadd(total, y<<16) with the total being realigned outside the loop.
- Type defs.h is redefined so that word lengths match the TMS320C62xx C code, not ETSI-assumed C sizes. In particular, long is 32 bits in ETSI C code, but it is 40/64 bits in the TMS320C62xx C code.
- Hand-optimized versions of standard loops
- Hand optimization of specific critical functions

In general, hand-optimized versions of functions occur in the *2.asm files of the original *.c file. Exceptions to this are some tight-loop bottom-level functions that have been added.

6.2 Program Hierarchy

The hierarchy of subroutines within the optimized GSM half-rate modules is shown in Figure 4.

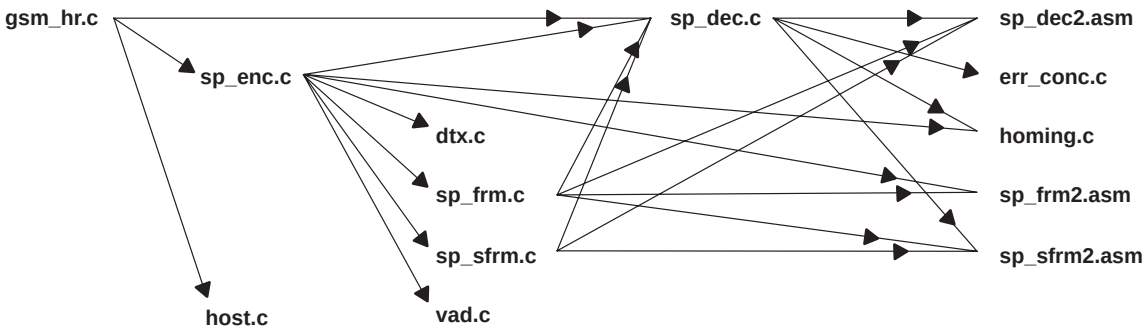


Figure 4. C+ Assembly-Optimized Program Hierarchy

The equivalent of Figure 4 for the pure C code version is shown in Figure 5.

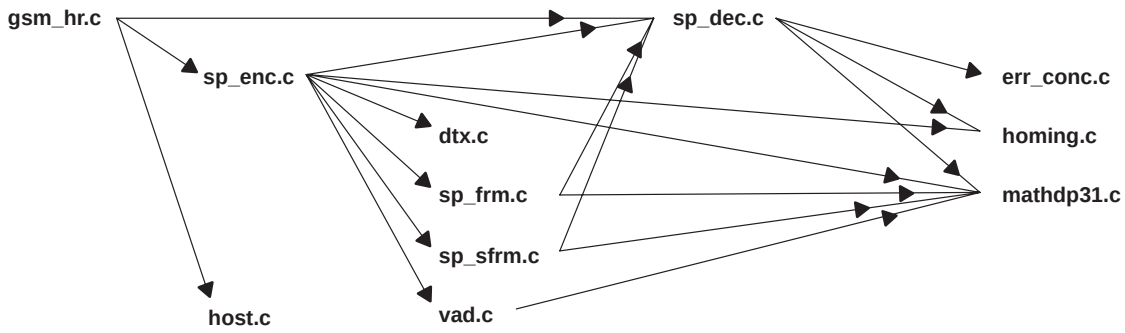


Figure 5. Pure C Only Program Hierarchy

7 Hand Assembly Routines

This section describes subroutines rewritten or altered from their original C form and is provided so that any modification due to changes in the standard can be implemented. With the exception of the new multichannel functions, you may skip this section if you are not dealing with the algorithm because external programs do not usually call these routines. The new multichannel functions provide multichannel support to the single-channel ETSI C code.

7.1 Standard Loops

Some common loops have been replaced with hand-optimized assembly subroutines that are placed in sp_sfrm2.asm (see Table 5). These subroutines can be called from any routine.

Table 5. Hand-Optimized Assembly Subroutines

Subroutine	Description
int SquareAccumulate (short* Array[length], int length);	Returns the sum of the squares of the elements in Array
movehalf_a (short* Input]length], short* Output[length], length);	Copies an array from one location to another. Input and Output must be word-aligned, even length, and not backward overlapping. (You can copy an array if it overlaps and if Output begins before Input). This subroutine is more efficient and restrictive than movehalf_u.
movehalf_u (short* Input[length], short* Output[length], length);	Copies an array from one location to another. Input and Output are not backward overlapping. (You can copy an array if it overlaps and Output starts before Input). There are no alignment restrictions, and there is no even length restriction; this version is slower than movehalf_a, which should be used if alignment is known.
Scale (short* Input[length], short* Output[length],scale, length);	Scales an array down by scale/65536. Input and Output must be word-aligned, of even length, and not backward overlapping. (You can scale an array down if it overlaps and if Output begins before Input).

7.2 New Multichannel Functions

Table 6 describes the new multichannel functions.

Table 6. New Multichannel Functions

Function	Description
Set_Channel (Address);	Points the DP register to the local (.bss) section
Return_Channel ();	Restores the DP register to the default (.bss) section
HANDLE_INTS();	NULL function that restores the DP register and handles interrupts in the middle of voice-coding code. It is not actually used but may be inserted in C code to allow more interrupts if needed.

7.3 Hand-Optimized Part Functions

These functions in the .asm files are hand-optimized versions of part of the equivalent C functions (see Table 7). They are named OldName_PartName, where OldName is the name of the original function, and part name is an additional name to distinguish it from the original function.

Table 7. Hand-Optimized Part Functions

Function	Description	Location
_flat3	Performs one iteration of the flat recursion from within the flat C function	sp_frm2.asm
_v_srch_mac	Performs the 3-dimensional multiply accumulate function from within the v_srch C function	sp_sfrm2.asm
_v_srch_search	Performs the 2-dimensional vector codebook search from within the v_srch C function	sp_sfrm2.asm

7.4 Hand-Optimized Complete Functions

These functions in the .asm files are hand-optimized versions of all of the equivalent C functions (see Table 8). They are named OldName, where OldName is the name of the original function. Their functionality is exactly the same as the old ETSI C function of the same name.

Table 8. Hand-Optimized Complete Functions

Function	Description	Location
_divide_s	Performs the same functions as the original C function divide_s	sp_dec2.asm; originally in mathdp31.c
_findBestInQuantList	Performs the same functions as the original C function findBestInQuantList	sp_frm2.asm
_fnBest_CG	Performs the same functions as the original C function fnBest_CG	sp_frm2.asm
_g_corr2	Performs the same functions as the original C function g_corr2	sp_sfrm2.asm
_getNextVec	Performs the same functions as the original C function getNextVec. The method of addressing vectors has been changed from 16-bit addressing with a separate upper and lower byte pointer to a single byte-addressed pointer. This also affects setupPreQ and setupQuant, inline C modules defined in sp_frm.h.	sp_frm2.asm
_hnwfilt	Performs the same functions as the original C function hnwfilt	sp_sfrm2.asm
_lpcFir	Performs the same functions as the original C function lpcFir	sp_dec2.asm
_lpclir	Performs the same functions as the original C function lpclir	sp_dec2.asm
_lpclrZslir	Performs the same functions as the original C function lpcIrZslir	sp_dec2.asm
_lpcZilir	Performs the same functions as the original C function lpcZiIir	sp_dec2.asm
_lpcZslir	Performs the same functions as the original C function lpcZslir	sp_dec2.asm
_lpcZsFir	Performs the same functions as the original C function lpcZsFir	sp_dec2.asm
_lpcZslirP	Performs the same functions as the original C function lpcZslirP	sp_dec2.asm
_quantLag	Performs the same functions as the original C function quantLag	sp_frm2.asm
_r0Quant	Performs the same functions as the original C function r0Quant	sp_frm2.asm

7.5 Hand-Optimized Multiple Functions

These functions in the .asm files are hand-optimized versions of all the C functions. They also include either some of the code that surrounds the call to that function or inline functions within the function (see Table 9).

These functions are generally named after the highest level complete function name but may include a postfix; for example, aflatRecursionLoop contains all of aflatRecursion and some of the code that previously surrounded all calls to aflatRecursion.

Table 9. Hand-Optimized Multiple Functions

Function	Description	Location
_aflatRecursionLOOP	Performs the same functions as the original C function aflatRecursionLoop but includes the loop that calls it from either sp_frm.c or dtx.c and an inline version of getNextVec	sp_frm2.asm
_decorr	Performs the same functions as the original C function decorr. Divide_s has been hand in-lined within this function.	sp_sfrm2.asm
_filt4_2nd	Performs the same functions as the original C functions filt4_2nd and iir_d	sp_frm2.asm

8 New C Routine: int** HR_Assign_GSM(int Channels);

This routine works with Set_Channel if there is insufficient space on the heap memory for all channels. This C routine assigns an array of pointers on the heap memory to the channel spaces. It assigns the existing .bss channel to the first address and assigns space on the heap memory for any additional channels. Then, it returns a pointer to the array of pointers.

This routine assigns the maximum number of channels and sets the pointer of the remaining channels to NULL. If there is insufficient space for the array of pointers, the routine returns a NULL value.

```
ChannelPointers = HR_Assign_GSM (NoOfChannels);
For (Channel=0,Channel<NoOfChannels, Channel++)
{
Set_Channel (ChannelPointers[Channel]);
/* Channel specific “.bss” processing */
Return_Channel ();
}
```

You must ensure that there is sufficient memory at link time, or check the return values for NULL at run time.

9 Compliance/Status

All of the C code has been compiled with version 2.1 of the TMS320C6x C compiler for the PC and simulated with version 2.0 of the simulator for the PC. All of the GSM half-rate voice coding DTX and VAD test vectors have passed this simulation and are designed to meet the standards described in Table 10.

Table 10. Compliance With GSM Standards

Standard	Title	Description
GSM 06.02 v4.0.2	ETS 300 581–1	Half-Rate Overview
GSM 06.20 v4.2.1	ETS 300 581–2	Half-Rate Transcoding
GSM 06.21	ETS 300 581–3	Half-Rate Lost Frames
GSM 06.22	ETS 300 581–4	Half-Rate Comfort Noise
GSM 06.41	ETS 300 581–5	Half-Rate DTX
GSM 06.42	ETS 300 581–6	Half-Rate VAD

The C code is also in compliance with a translation and optimization of GSM 06.06 v4.0.3, ETS 300 581–7, Half-Rate ANSI–C, January 1996.

It has been verified to GSM 06.07 v4.0.3, ETS 300 581–8, Half-Rate Test Sequence, January 1996.

10 Vocoder Variables Available for Subsections

Table 11 shows the voice coder variables that are required and available for each subsection of the voice coder/decoder.

Table 11. Vocoder Variables and Subsections

Parameter	Encode Available	Decode Required
R0	1	1
LPC[1..3]	1	1
INT_LPC	1	1
MODE	1	1
Lag(1)_Code1(1)	2	1
Code(1)/Code2(1)	2	1
GSP(1)	2	1
Lag(2)_Code1(2)	3	2
Code(2)/Code2(2)	3	2
GSP(2)	3	2
Lag(3)_code1(3)	4	3
Code(3)/Code2(3)	4	3
GSP(3)	4	3
Lag(4)_Code1(4)	5	4
Code(4)/Code2(4)	5	4
GSP(4)	5	4
VAD/SP	1	
BFI/UFI/SID/TAF		1

11 Pure C Code Version

The pure C code version is produced by back-fitting C from previous versions into the latest C+asm-optimized code. The code is not pure ETSI C because it contains TMS320C62xx intrinsics. No attempt is made to take advantage of ongoing compiler improvements after a module is hand-optimized. This code is verified against the same test sequences as the hand-optimized code. Pure ANSI C code that is not processor-specific is also available from ETSI.

12 References

ETS 300 581 1-8 Half-Rate Speech

TMS320C6x Optimizing C Compiler (SPRU187E)

TMS320C6201/6701 Peripherals Reference Guide (SPRU190C)