Application Report
SPRA625 - February 2000

GSM Adaptive Multi-Rate Voice Coding on the TMS320C62x™ DSP

Peter Dent — *Wireless Communications*

ABSTRACT

This document describes the GSM adaptive multi-rate (AMR) voice coding implementation on the TMS320C62x™ generation DSP. It discusses the system requirements and provides two levels of detail for the subroutines—one for programmers who want to use the code in a basic GSM architecture, and the other for programmers who want to use the code as a basis for optimizing code or modifying other algorithms.

Contents

1 Introduction . . . 3
1.1 Implementation Functions . . . 3
1.2 Typical Implementation . . . 5
2 System Requirements . . . 5
2.1 Memory Allocation . . . 5
2.2 Channel Switching and Interrupts . . . 6
2.3 Execution Time Benchmarks . . . 6
2.4 Program Memory Requirements . . . 6
2.5 Data Memory Requirements . . . 7
3 Top-Level Routines . . . 7
3.1 main . . . 7
3.2 AMR_Assign_GSM . . . 7
3.3 AMR_SpeechEncoder1...5 . . . 8
3.4 AMR_SpeechDecoder1...4 . . . 8
3.5 Set_Channel . . . 8
3.6 Return_Channel . . . 9
3.7 AMR_ResetEnc . . . 9
3.8 AMR_ResetDec . . . 9
3.9 Multichannel Setup Example . . . 9
4 Linker Notes (amr_Vocoder.cmd) . . . 10
4.1 Partial_Link_AMR.cmd . . . 10
4.2 GSM_AMR_TI.cmd . . . 10
4.3 Code Composer Make Files . . . 10
4.4 Cont_Dual.gsm . . . 10
5 Software . . . 11
5.1 Primary Optimizations . . . 11
6 Hand Assembly Routines . . . 12

TMS320C62x is a trademark of Texas Instruments.

6.1 Standard Loops . . . 12
6.2 New Multichannel Functions . . . 12
6.2.1 Set_Channel (Address); . . . 12
6.2.2 Return_Channel ( ); . . . 12
6.2.3 Handle_Ints( ); . . . 13
6.3 Hand-Optimized Part Functions . . . 13
6.4 Hand-Optimized Complete Functions . . . 13
6.5 New Multichannel Functions in eXpress DSP Mode . . . 14
6.5.1 algControl . . . 15
6.5.2 Extended eXpress DSP Functions . . . 15
6.6 New Multichannel Functions With Old Half-Rate/Full-Rate Code . . . 16
6.6.1 Set_Channel (Address); . . . 16
6.6.2 Return _Channel ( ); . . . 16
6.6.3 Handle_Ints( ); . . . 16
6.6.4 int** AMR_Assign_GSM(int Channels); . . . 16

7 Code Compliance/Status . . . 17

8 Vocoder Variables Available and Needed for Subsections . . . 17

References . . . 17

List of Figures

1 GSM AMR Speech Vocoder . . . 4
2 GSM AMR Speech Decoder . . . 4
3 Typical GSM Base Station Implementation . . . 5

List of Tables

1 Execution Times . . . 6
2 Data Memory Requirements . . . 7
3 Weight of Optimization Considerations . . . 11
4 Hand-Optimized Part Functions . . . 13
5 Hand-Optimized Complete Functions . . . 14
6 Standard eXpress DSP Functions . . . 14
7 eXpress DSP Control Commands . . . 15
8 Extended eXpress DSP Functions . . . 15
9 Compliance Standards . . . 17
10 Vocoder Variables and Subsections . . . 17

List of Examples

1 Using a Subroutine . . . 9
2 Defining Variables for Memory Allocation . . . 10
3 Cont_Dual.GSM File . . . 11
4 int** AMR_Assign_GSM(int Channels); . . . 16

1 Introduction

This document describes the Global System for Mobile Communications (GSM) adaptive multi-rate (AMR) voice coding implementation on the TMS320C62x™ generation digital signal processor (DSP). 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 modification, for using it as a basis to optimize code for other algorithms, or for porting to enhanced members of the TMS320C62x DSP generation.

This voice coder (vocoder) is available as C-callable C code with hand-optimized assembly code.

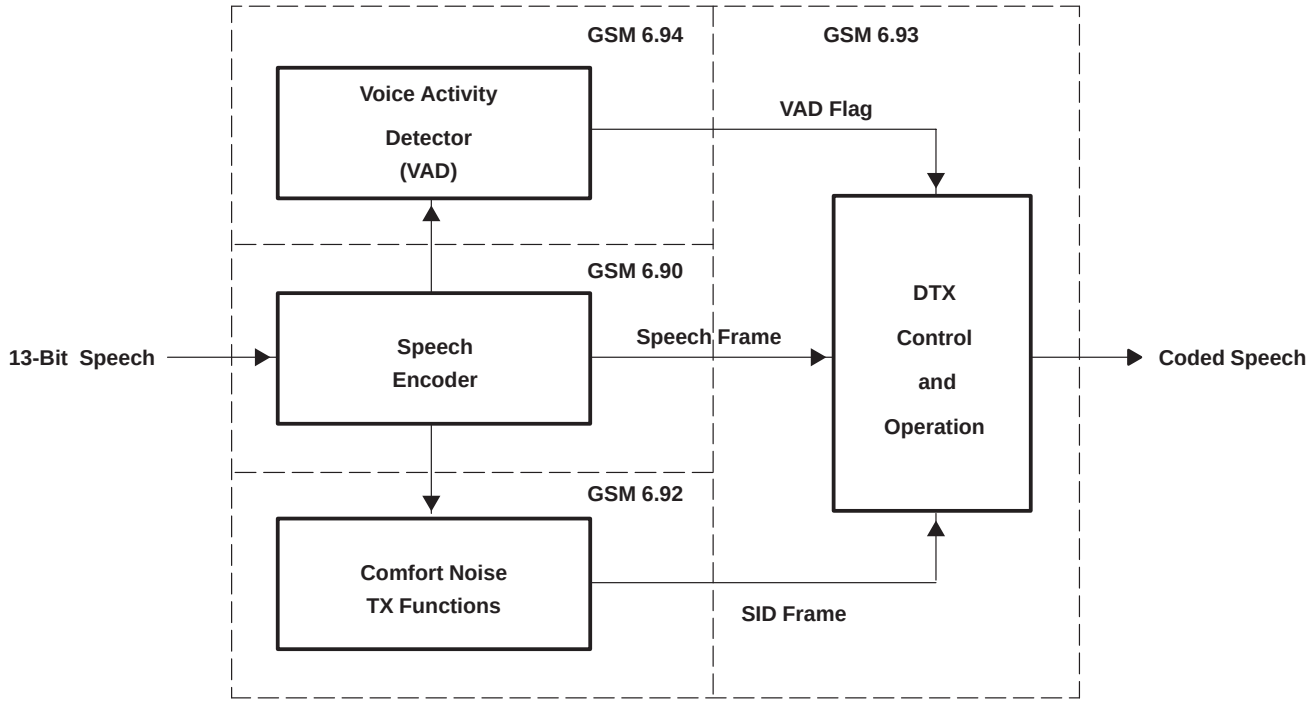
Wherever possible, the nomenclature in the code is that used in the standard C code provided by the European Telecommunications Standards Institute (ETSI). Most variable names are maintained.

This document does not describe the GSM AMR voice coding algorithm. For more detailed information on the voice-coding algorithm, see the relevant ETSI documents EN 301 703…708.[1]

1.1 Implementation Functions

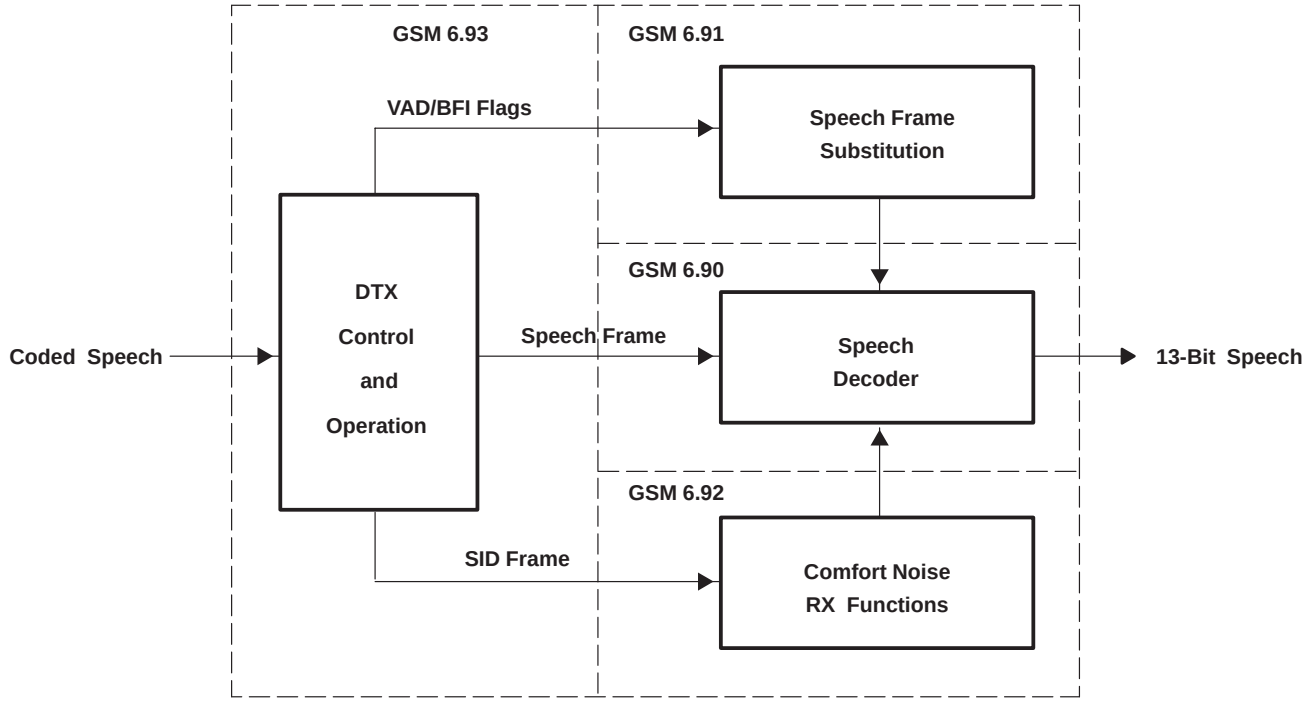
This implementation includes all of the AMR voice-coding functions for the GSM AMR vocoder and decoder. These are GSM specifications 6.90, 6.91, 6.92, 6.93, and 6.94. Figure 1 and Figure 2 illustrate the vocoder and decoder, respectively, and show how these modules interact to provide the voice coding and decoding functions.

The code is based on the ANSI C code in GSM 6.73 optimized for the TMS320C62x DSP generation and has been verified with the tests specified in GSM 6.74.



NOTES: A. DTX = Discontinuous transmission
B. SID = Silence descriptor
C. TX = Transmit

Figure 1. GSM AMR Speech Vocoder

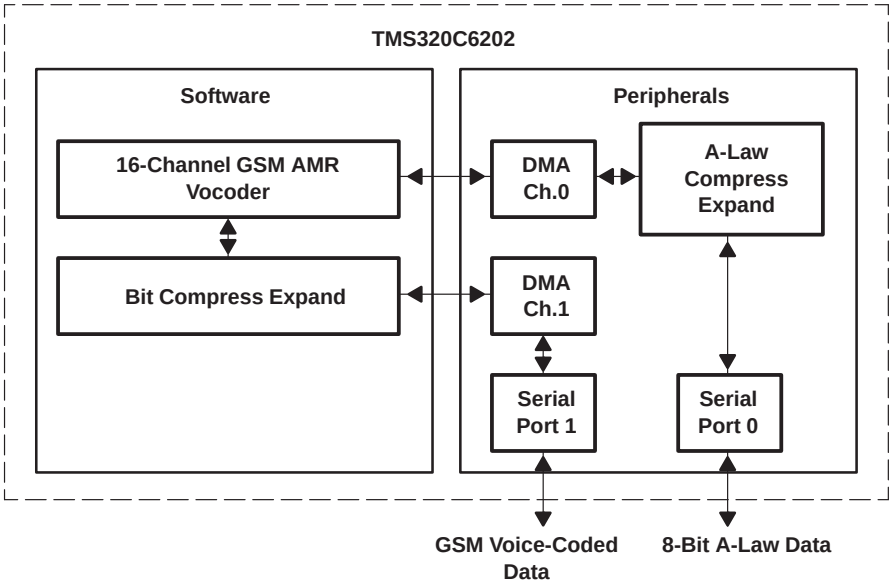


NOTES: D. BFI = Bad frame indication
E. RX = Receive

Figure 2. GSM AMR Speech Decoder

1.2 Typical Implementation

A typical implementation of the voice coder in a GSM base station is shown in Figure 3.



NOTES: F. DMA = Direct memory access

Figure 3. Typical GSM Base Station Implementation

The TMS320C6202™ DSP performs all the 16-channel voice coding of the GSM AMR vocoder. 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 pulse code modulation (PCM) coded data depends on the application and is not covered in this document. Further information on typical methods for using the serial and direct memory access (DMA) ports is available in the TI *TMS320C6000 Peripherals Reference Guide*.[3]

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.[2]

C Compiler version 3.01 supports options –mtw, –mh2147483647, –mr1, and –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 3.01.

2.1 Memory Allocation

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

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

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

TMS320C6202 is a trademark of Texas Instruments.

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 available for either the array of pointers or the channel data space, then some or all of the pointers are going to be NULL. You must ensure that there is sufficient heap memory at compile time or check the values returned for error conditions. The example code allows enough heap space for the channel input/output (CIO) and 16 channels of GSM AMR voice coding.

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 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 in 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

The execution time benchmarks shown in Table 1 were obtained with Compiler 3.01 and Code Composer Studio™ 1.0 C6202 Simulator. They are measured from the label AMR_SpeechEncoder1 or AMR_SpeechDecoder1 to the return address from the last subpart. All data and code are stored in internal memory. These benchmarks do not include the CIO 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 CIO were used.

Table 1. Execution Times

Processor	Channels	MHz
TMS320C6202 C+asm	1	9.9
TMS320C6202 C+asm	16	158.4

2.4 Program Memory Requirements

The program size of the GSM AMR vocoder subroutines is 126,400 bytes. 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. The program fits inside the memory space of the TMS320C6202 DSP and does not require any external program memory.

Code Composer Studio is a trademark of Texas Instruments.

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 2). Other areas are based on the sample code shipped with the algorithm and are implementation-dependent. A 16-channel vocoder fits into the internal data memory of the TMS320C6202 DSP.

Table 2. Data Memory Requirements

Function	Size (Bytes)
Stack	8192
Constants	29424
Channel variables	5664

3 Top-Level Routines

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

3.1 main

The C program *main* is used to apply the test patterns via Code Composer Studio to the GSM AMR 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 AMR_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 AMR routine:

```
int**= AMR_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 cod_amr.c file. If a NULL value is returned in any of the pointers, it means that there is insufficient heap memory available.

3.3 AMR_SpeechEncoder1...5

The following subroutines handle the encoding of a speech frame to the GSM AMR standard:

```
AMR_SpeechEncoder1 (*Channel, *Input, *Output[0]);
AMR_SpeechEncoder2 (*Channel, *Input, *Output[0]);
AMR_SpeechEncoder3 (*Channel, *Input, *Output[0]);
AMR_SpeechEncoder4 (*Channel, *Input, *Output[0]);
AMR_SpeechEncoder5 (*Channel, *Input, *Output[0]);
```

Where:

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

Input is the address of 160 samples of 13-bit 2s complement linear speech input. In addition, the requested AMR voice-coding mode is passed as the 161st input data sample.

Output is the address to put the encoded frame of GSM variables. All three parameters should be the same for each call to the same channel.

Set_Channel and Return_Channel are handled within this routine and should not be called. In single-channel applications, Channel should be __bss__ (a TMS320C62x DSP compiler constant). The code can be found in cod_amr.c. The GSM AMR parameters will be in an uncompressed array of bits. Data packing is not performed and interrupts must be disabled before these routines are called.

3.4 AMR_SpeechDecoder1...4

The following subroutines handle the decoding of a speech frame to the GSM AMR standard:

```
AMR_SpeechDecoder1 (*Channel, *Input, *Output);
AMR_SpeechDecoder2 (*Channel, *Input, *Output[40]);
AMR_SpeechDecoder3 (*Channel, *Input, *Output[80]);
AMR_SpeechDecoder4 (*Channel, *Input, *Output[120]);
```

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 TMS320C62x DSP compiler constant). The code can be found in dec_amr.c. The GSM AMR parameters are in an uncompressed array of bits. Data unpacking is not performed. Interrupts must be disabled before calling these routines.

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 discontinuous transmission (DTX) mode is required. The following subroutine may be called before channel-specific modes are examined or changed.

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

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

This subroutine is found in the channels.asm file and it is defined in the gsm.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 gsm.h file.

3.7 AMR_ResetEnc

This required subroutine resets the encoder at the start of a call. The call for this subroutine is:

```
AMR_ResetEnc(int* Channel,int DTXenable)
```

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 TMS320C62x DSP compiler constant). The code is found in the cod_amr.c file.

3.8 AMR_ResetDec

This required subroutine resets the decoder at the start of a call:

```
AMR_ResetDec(int* Channel)
```

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 TMS320C62x DSP compiler constant). The code is found in the dec_amr.c file.

3.9 Multichannel Setup Example

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

Example 1. Using a Subroutine

```
        if (!strcmp(DTXmode,”dtx”))
       {
            AMR_ResetEnc(Channels[ChCount],1);
/* enable DTX for VAD tests */
            AMR_ResetDec(Channels[ChCount]);
        }
        else
        {
            AMR_Reset(Channels[ChCount],0);
            /* disable DTX for non–VAD tests */
            AMR_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 (amr_Vocoder.cmd)

Two variables are defined by the linker to allow allocation of the memory for external variables (see Example 2). These variables must be defined before and after the .bss sections for the subparts of the vocoder variables; otherwise, memory for additional channels will be incorrectly allocated. A list of the modules required between these variables is in each of the example link control files.

Example 2. Defining Variables for Memory Allocation

```
_AMR_ram_Start = .;      /* used to allocate heap for
                      additional channels */
  cod_amr.obj (.bss)
  dec_amr.obj (.bss)
  .
  .
  .
 _AMR_ram_End = .; /* used to allocate heap for
                      additional channels */
```

Each module should be compiled separately using the –mt –mh –o3 options.

4.1 Partial_Link_AMR.cmd

This link command file should be used in implementations with multiple voice coders that use the same standard as the older half-rate (HR) and full-rate (FR) vocoders. This command links the routines together into a single module that can be linked again later. The global definitions are reduced to the minimum required. Only global definitions beginning with AMR_ remain. This command can also be used to link code from multiple sources.

4.2 GSM_AMR_TI.cmd

This link command file should be used in implementations where multiple voice coders are used with the new eXpress DSP™ algorithm standard. It links the routines together into a single module that can be linked again later.

4.3 Code Composer Make Files

Three Code Composer™ make files are included: CC_GSM_AMR.MAK, CC_GSM_AMR_PL.MAK, and GSM_AMR_TI.MAK. These three files can be used to build a standalone full-rate voice coder, a reusable library module compatible with the older FR and HR vocoders, or a reusable library module compatible with the new eXpress DSP standard, respectively.

4.4 Cont_Dual.gsm

This file tells the CIO how to handle all of the test pattern files. For each test sequence, there are 6 or 7 lines, as shown in Example 3.

Code Composer and eXpress DSP are trademarks of Texas Instruments.

Example 3. Cont_Dual.GSM File

```
PCM Data Input File
GSM Data Output File
GSM Data Input File
PCM Data Output File
dtx| nodtx
MR122| MR102| MR795| MR74| MR67| MR59| MR515| MR475| AMR
(optional)
```

The first four lines are the filenames of the input and output files. The fifth line indicates whether that code sequence is to be run with voice activity and discontinuous transmission enabled or disabled (dtx or nodtx, respectively). The sixth line indicates the GSM data rate; the rate is either the fixed MR rate or the AMR rate. If the rate is AMR, there is an additional seventh line containing a filename that lists the mode to be used on a frame-by-frame basis.

This structure is repeated for the number of test sequences that need to be run; at the end of the sequence, the C program exits.

5 Software

Optimization is always a compromise between different objectives. The primary considerations and the approximate weight of each is shown in Table 3.

Table 3. 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%

† MIPS = Million instructions per second

5.1 Primary Optimizations

- All calls to ETSI functions defined in the mathhalf.c file are replaced with TMS320C62x C code intrinsics, for example, sadd (x,y) => _sadd(x<<16,y<<16)>>16. In practice, when combined with other optimizations, this code often becomes _sadd(total, y<<16) and the total is realigned outside the loop.
- All functions in the mathdp31.c file are replaced with inline subroutines except divide_s, which is in hand-optimized assembler. These are all very short subroutines that are called regularly. Because these subroutines use few variables, they provide savings in the subroutine call and return, and better register usage in the calling routine.
- All look-up ROMs are forced to the constants (.const) section for compiler absolute addressing. This forces the compiler to use absolute addressing instead of DP (.bss) addressing. This use of absolute addressing is primarily advantageous for multichannel implementations because constants are global to all channels and .bss remains available for variables.

- All channel-specific context variables make the DP relatively addressable for online context switching by changing the DP to compiler DP offset addressing. This is done primarily for multichannel operations.
- All local temp variables are stack based and use register or stack pointer (SP) offset addressing. This is done primarily for multichannel operations.
- Local shorts are redefined to ints 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 realigned outside the loop.
- Typedefs.h file is redefined so that word lengths match the TMS320C62x DSP C code instead of ETSI-assumed C sizes. In particular, long is 32 bits in ETSI C, but long is 40/64 bits in TMS320C62x DSP C code.
- Hand-optimized versions of standard loops are created.
- Hand optimization of specific critical functions is performed.

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

6 Hand Assembly Routines

This section describes subroutines that have been rewritten or altered from their original C form. These subroutines are provided so that you can implement any modifications that occur due to a change in the standard. With the exception of the new multichannel function, users who do not understand the algorithm or do not need to change the algorithm can skip this section because external programs would not normally call these routines. New multichannel functions are those that add multichannel support to the single-channel ETSI C code.

6.1 Standard Loops

The copy function has been rewritten as a standard hand-optimized function. This function can be found in the copy_ho.asm file as two versions: copy_ho and copy_ho_a. Both these subroutines are called identically, but copy_ho_a is faster, has more memory restrictions, requires longer arrays, and is word-aligned data.

6.2 New Multichannel Functions

The new multichannel functions are explained in this section.

6.2.1 Set_Channel (Address);

The Set_Channel (Address) routine points the DP register to the local (.bss) section.

6.2.2 Return_Channel ( );

The Return_Channel ( ) routine restores the DP register to the default (.bss) section.

6.2.3 Handle_Ints( );

The Handle_Ints( ) routine is a NULL function that restores the DP and handles interrupts in the middle of voice coding code. Although not actually used, it may be inserted into C code to allow more interrupts, if needed.

6.3 Hand-Optimized Part Functions

These functions in the .asm files are hand-optimized versions of a part of one or more ETSI C functions, as shown in Table 4.

Table 4. Hand-Optimized Part Functions

Function Name	New Location	Old Location
Build_code1035_ho	build_code1035_ho.asm	c1035.c
ReorderLPC_ho	ReorderLPC_ho.asm	New function to allow different data optimizations for different functions operating on the same data
Search3_ho	search3_ho.asm	c3_14pf.c
Search4_ho	search4_ho.asm	c4_17pf.c
Search_8i40_ho Search_10i40_ho	search_10i40_ho.asm	s10_8pf.c
vq_subvec_ho	vq_subvec_ho.asm	q_plsf.c
vq_subvec_s_ho	vq_subvec_s_ho.asm	q_plsf.c

Search_10i40_ho.asm can produce code to search for either 8 or 10 codebook vectors. Because both rates are used by different vocoder rates, Search_10i40_ho.asm should be included twice, once assembled with *search* assigned a value of 8 and once with *search* assigned a value of 10. The overall assembly file hand_opt.asm handles this and all assembly.

6.4 Hand-Optimized Complete Functions

These functions in the .asm files are hand-optimized versions of all of the equivalent C functions. Their functionality is arithmetically the same as the old ETSI C functions with similar names.

Set_sign8and10_.asm can produce code to set the sign in either 8 or 10 codebook vectors. Because both rates are used by the different vocoder rates, it needs to be included twice, once assembled with search assigned to a value of 8 and once with search assigned to a value of 10. The overall assembly file, hand_opt.asm, handles this operation.

Table 5. Hand-Optimized Complete Functions

Function Name	New Location	Old Location
Convolve_ho	Convolve_ho.asm	Convolve.c
Cor_h_ho	cor_h_ho.asm	cor_h.c
Cor_h_x_ho	cor_h_x_ho.asm	cor_h.c
e_homing_ho	e_homing_ho.asm	e_homing.c
Inv_sqrt_ho	inv_sqrt_ho.asm	inv_sqrt.c
Log2_ho	log2_ho.asm	log2.c
Lsp_az_ho	lsp_az_ho.asm	lsp_az.c
Lsp_lsf_ho	lsp_lsf_ho.asm	lsp_lsf.c
Pow2_ho	pow2_ho.asm	pow2.c
Pre_proc_ho	pre_proc_ho.asm	pre_proc.c
Reorder_ho	reorder_ho.asm	reorder.c
Residu_ho	residu_ho.asm	residu.c
Set_sign8_ho Set_sign10_ho	set_sign8and10_ho.asm	Set_sign.c
Set_sign_ho	set_sign_ho.asm	Set_sign.c
Syn_filt_ho	syn_filt_ho.asm	syn_filt.c

6.5 New Multichannel Functions in eXpress DSP Mode

The new multichannel functions in eXpress DSP mode are explained in this section. Table 6 describes the standard eXpress DSP functions.

Table 6. Standard eXpress DSP Functions

Function	Implementation	Comments
*implementationId	Mandatory–Implemented	See the TI *eXpress DSP Algorithm Standard API Reference.*
AlgActivate	Optional – Not Implemented	NULL
AlgAlloc	Mandatory – Implemented	parentFxns field is ignored, 2 memTabs
AlgControl	Optional – Implemented	Performs channel reset and mode selection (See Section 6.5.1, *algControl.)*
AlgDeactivate	Optional – Not Implemented	NULL
AlgFree	Mandatory–Implemented	See the TI *eXpress DSP Algorithm Standard API Reference.* 2 memTabs
algInit	Mandatory–Implemented	See the TI *eXpress DSP Algorithm Standard API Reference.* Resets coder (DTX disabled) + decoder
AlgMoved	Optional – Implemented	See the TI *eXpress DSP Algorithm Standard API Reference.* Changes algorithm history pointer(s)
AlgNumAlloc	Optional – Implemented	Returns 2

6.5.1 algControl

This function implements algorithm control functions. Status is unused; the command word is divided into bit fields, as shown in Table 7.

Table 7. eXpress DSP Control Commands

Bit Field	Value	Function
1 … 0	00 01 10 11	No action No action Reset encoder, DTX disabled Reset encoder, DTX enabled
2	0 1	No action Reset decoder
31 … 3	0	Unused

6.5.2 Extended eXpress DSP Functions

Table 8. Extended eXpress DSP Functions

GSM_SpeechEncoder1	Code Frame Parameters for Vocoder
GSM_SpeechEncoder1	Code frame parameters for vocoder
GSM_SpeechEncoder2	Code subframe 1 parameters for vocoder
GSM_SpeechEncoder3	Code subframe 2 parameters for vocoder
GSM_SpeechEncoder4	Code subframe 3 parameters for vocoder
GSM_SpeechEncoder5	Code subframe 4 parameters for vocoder
GSM_SpeechDecoder1	Decode frame and subframe 1 parameters for vocoder
GSM_SpeechDecoder2	Decode subframe 2 parameters for vocoder
GSM_SpeechDecoder3	Decode subframe 3 parameters for vocoder
GSM_SpeechDecoder4	Decode subframe 4 parameters for vocoder

In all cases, the format of these functions is the same. For example:

```
Void (*GSM_SpeechVocoder)(IGSM_Handle handle, Short pswIn[], Short pswOut[]);
```

Where:

GSM_SpeechVocoder is the partial encoder or partial decoder.

handle is the same as that used in the standard eXpress DSP functions.

pswIn is the pointer to the first data word of input data for the frame.

pswOut(coders) is the pointer to the first data word of output data for the frame.

pswOut(decoders) is the pointer to the first data word of output data for the subframe.

6.6 New Multichannel Functions With Old Half-Rate/Full-Rate Code

The new multichannel functions to use with old HR and FR code are explained in this section.

6.6.1 Set_Channel (Address);

This function points the DP register to the local (.bss) section.

6.6.2 Return _Channel ( );

This function restores the DP register to the default (.bss) section.

6.6.3 Handle_Ints( );

This is a NULL function that restores the DP and handles interrupts in the middle of voice-coding code. Although not actually used, this function may be inserted in C code to allow more interrupts, if needed.

*6.6.4 int** AMR_Assign_GSM(int Channels);*

This function assigns an array of pointers to channel spaces on the heap memory, then assigns the existing .bss channel to the first address and assigns space on the heap memory for any additional channels, then returns a pointer to the array of pointers.

This function works with Set_Channel if there is insufficient heap memory for all channels. This function assigns the maximum number of channels and sets the pointer of the remaining channels to NULL. If sufficient heap memory exists for the array of pointers, this function returns a NULL value.

Example 4. int AMR_Assign_GSM(int Channels);**

```
ChannelPointers = AMR_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 else check the return values for NULL at run time.

7 Code Compliance/Status

All of the code has been compiled with version 3.01 of the TMS320C6x™ DSP generation C Compiler for the PC and simulated with version 1.0 of Code Composer Studio for the PC, and on an EVM. All of the GSM AMR Voice Coding DTX, VAD, and homing test vectors have passed this simulation. This code is designed to meet the standards listed in Table 9.

Table 9. Compliance Standards

Standard	ETSI No	Description
GSM 06.71	EN 301 703 v7.0.1	AMR overview
GSM 06.90	EN 301 704 v7.1.0	AMR transcoding
GSM 06.91	EN 301 705 v7.0.1	AMR lost frames and DTX
GSM 06.92	EN 301 706 v7.1.0	AMR comfort noise
GSM 06.93	EN 301 707 v7.1.0	AMR DTX
GSM 06.94	EN 301 708 v7.1.0	AMR VAD

The C code is also in compliance with a translation and optimization of GSM 6.73 EN 301 712 v7.1.0 AMR ANSI-C, September 1999.

This code has been verified to GSM 6.74 AMR Test Sequence, October 1999.

Changes to algorithms or vector specifications on or after October 18, 1999 are NOT incorporated in this application report.

8 Vocoder Variables Available and Needed for Subsections

Table 10 shows the voice coder variables needed and available for each subsection of the voice coder/decoder.

Table 10. Vocoder Variables and Subsections

Parameter	Encode Available	Decode Needed
Frame	1	1
Subframe1	2	1
Subframe2	3	2
Subframe3	4	3
Subframe4	5	4
VAD/SP	1	
TAF†/SID/BFI		1

† TAF = Time alignment flag

References

1. EN 301 703–8 Adaptive Multi-Rate Speech
2. TI *TMS320C6x Optimizing C Compiler* (SPRU187E)
3. TI *TMS320C6000 Peripherals Reference Guide* (SPRU190C)

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 acknowledgment, 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.

Customers are responsible for their applications using TI components.

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.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265