.

P&D: Embedded Systems and Multimedia

Sub-band Speech Coding

2015-2016

Mohit Dandekar
John O’Callaghan

Contents

List of Tables

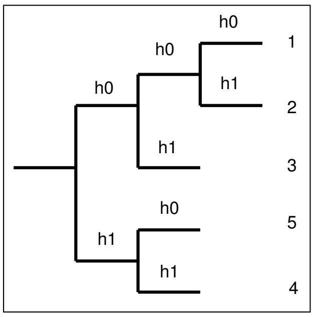
List of Figures

1 Introduction

A speech codec scheme is required to compress stereo audio at a bit rate of 24 kbit/s per channel for a sampling frequency of 8 kHz. The design specifications require a polyphase implementation of a QMF tree-structured filterbank with adaptive quantization of each subband signal. The concept will be implemented first in MATLAB, later converted to C and finally optimised for implementation on a DSP.

1.1 Filter Bank Tree and Bit Allocation

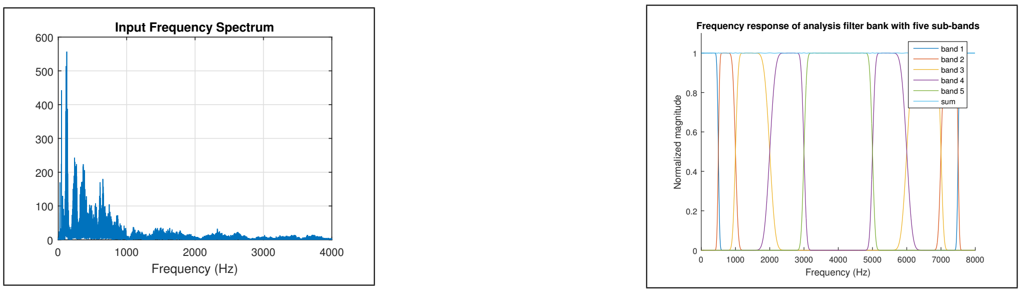
The subband decomposition is chosen as described in [?] that decomposes the input 0 – 4 kHz band into five subbands through a non uniform tree of recursive two-band QMF decompositions. The filter tree structure used on the analysis filter bank is shown in Table ?? below.



Band	Band Edges (Hz)	Bits
1	0 – 500	5
2	500 – 1000	5
3	1000 – 2000	4
4	2000 – 3000	3
5	3000 – 4000	0

Table 1: Sub-band decomposition and bit allocation

The band edges and bit allocation used for each sub-band are shown in Table ?? [?] [?]. This bit allocation was chosen because most of the energy in typical speech signals is contained in the 0 – 1 kHz region, as can be seen from the FFT of the words_m input file shown in Figure ??. The signal content in the 3 – 4 kHz band is least important for speech perception, and it was decided not to allocated any bits for this frequency band and instead allocate more bits to lower frequency bands.



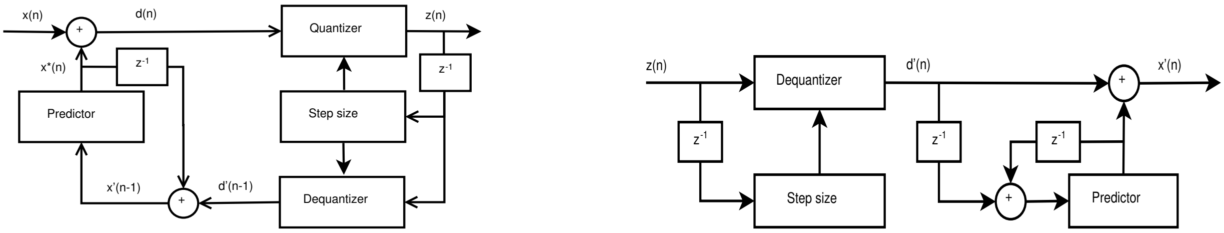
(a) FFT of the words_m input file.

(b) Analysis Filter Bank Frequency Response.

Figure 1: Typical Input Speech Spectrum and Filter Bands

1.2 ADPCM Coder-Decoder

The ADPCM codec was chosen as described in [?]. The codec block diagrams are shown in Figure ?? (encoder) and Figure ?? (decoder).



(a) ADPCM encoder block diagram.

(b) ADPCM decoder block diagram.

Figure 2: ADPCM Encoder and Decoder block diagrams

Let the input PCM data be $x(n)$, the prediction be $x^*(n)$, the prediction error be $d(n)$, the quantization scale be $\Delta(n)$, the scaled and quantized output be $z(n)$ and the dequantized prediction error be $d'(n)$. Then, each iteration of the encoder can be described by the following set of equations:

$$\begin{aligned} d'(n-1) &= z(n-1) \cdot \Delta(n-1) \\ x^*(n) &= \mu\left(x^*(n-1) + d'(n-1)\right) \\ d(n) &= x(n) - x^*(n) \\ \Delta(n) &= \Delta(n-1) \cdot F\left(|z(n-1)|\right) \\ z(n) &= Q\left(\frac{d(n)}{\Delta(n)}\right) \end{aligned} \tag{1}$$

For the decoder, let the ADPCM input be $z(n)$, the dequantizer scale be $\Delta(n)$, the dequantized ADPCM input be $d'(n)$, the predictor output be $x^*(n)$ and the reconstructed PCM output be $x'(n)$, then each iteration of the decoder can be described by the following set of equations:

$$\begin{aligned} \Delta(n) &= \Delta(n-1) \cdot F\left(|z(n-1)|\right) \\ d'(n) &= z(n) \cdot \Delta(n) \\ x^*(n) &= \mu\left(x^*(n-1) + d'(n-1)\right) \\ x'(n) &= x^*(n) + d'(n) \end{aligned} \tag{2}$$

The differential coder works on the principle of encoding only the difference between each input sample and a previously computed prediction of the sample. The first-order prediction is simply the previous sample scaled by a precomputed factor μ. The choice of this coefficient is a design optimization task. The step-size used by the quantizer is adaptively adjusted based on the signal level of the previous quantized output. This step-size, the dequantized difference signal, and an approximation of the original input signal are then recovered by the decoder provided that the initial step value, and the step size computation logic are replicated exactly in both encoder and decoder. [?]

1.3 Design and Development Approach

The first stage of the implementation is carried out in MATLAB with arithmetic computations in full precision (floating point), and without the quantization of the final ADPCM output. This simple test ensures perfect reconstruction and verification that the ADPCM encoder is performing the adaptive scaling of the PCM input of each band to the desired dynamic range. Then the algorithm is verified with quantization applied only to the coder output and the performance is measured (PESQ and SNRSEG). This is followed by the complete fixed point implementation of each computation block by first, selecting the correct Q format for each data stream through the algorithmic flow and modifying the arithmetic operations to adhere to the selected Q formats.

The MATLAB code thus achieved serves as the golden reference for the development to follow. All the (PC) C kernels are to be verified against this reference. This part gets tricky as unlike an interpreted programming environment like MATLAB the C kernels are difficult to test/debug on the fly → difficult to develop in isolation. Hence a hybrid approach is followed. The C kernels are developed in a bottom-up modular manner with unit testing of each module in a co-testing setup. This setup has a MATLAB script that implements the target functionality and processes the input data to produce the reference output. The input data and reference output are then written out to testvectors (data files). The C kernel is then tested with a testwrapper around it that loads the testvectors, invokes the C kernel under test and verifies the output against the expected output. This is done automatically. The automation also removes laborious manual data comparison and debugging.

Such a co-testing approach enables a very robust and incremental test driven development of the reference C kernel modules in a bottom-up manner. As the fixed point issues have already been considered at MATLAB level, the C kernel modules are tested with a stringent bitexactness criteria against the MATLAB modules.

2 MATLAB Implementation

The first stage of implementation is carried out in MATLAB by encapsulating the QMF 2 bank analysis/synthesis filter and the ADPCM encoder/decoder into MATLAB functions stored in the '`./MATLAB/src`' directory in the project folder. These implementations work with either floating or fixed point computation, that can be selected at runtime by passing a 'mode' parameter into the function. The '`./MATLAB/tests`' folder contains various tests including the reference implementation of the full subband codec. Some utility functions can be found under directory '`./MATLAB/utils`'. Further details on the m-files can be found in section **??**.

As the final application is to be real time, i.e. the input voice PCM data is to be received in chunks of a fixed number of samples in a corresponding unit of time, the subband coding scheme has to be implemented in a block-processing manner. This online block wise processing has to be equivalent to offline processing where all the input data is available at the beginning. This adds an overhead in the implementation: for the QMF filter bank tree, each QMF filter has to be implemented as an overlap-add system and the ADPCM coder/decoder have to be implemented to run across consecutive blocks with feed forward self initialization.

2.1 Subband Tree

The subband tree structure chosen requires 3 stages with a total of 4 QMF banks as shown in figure **??**.

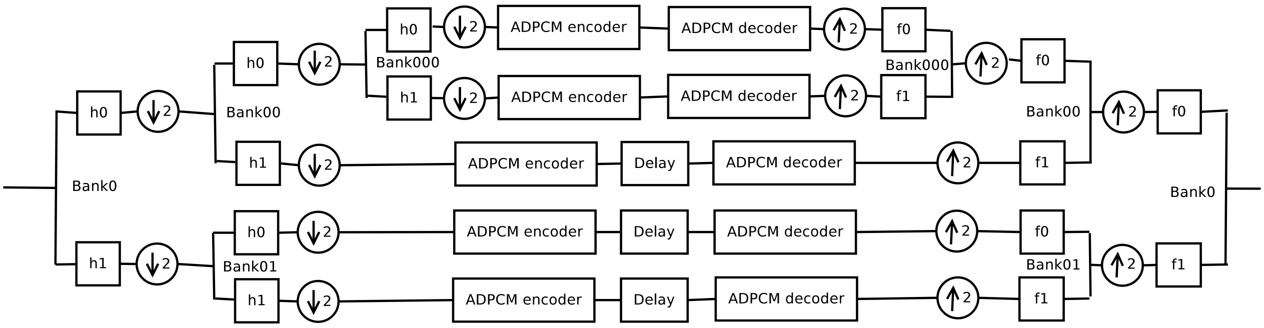


Figure 3: Block Diagram for the full CODEC

The input chunk of blocksize L that passes through the analysis filter bank yields 5 streams at decimation level 1/8, 1/8, 1/4, 1/4, 1/4 respectively. Thus the QMF bank0 accepts block size L, bank00 and bank01 accept block size L/2 and the bank000 accepts block size L/4 for the analysis side and output mentioned block size on the synthesis side.

There are two considerations for the implementation of the QMF. First, as the QMF output is decimated by 2 and owing to the structure of the analysis/synthesis filters, a two phase decomposition of the filter bank followed by a trivial two point DFT yields an efficient implementation. Second, as the decomposition tree is non uniform, i.e. all the output streams are not at the same decimation rate, the delays suffered in perfect reconstruction branches at different decimation rates are different and hence must be compensated for during reconstruction as shown in figure **??**.

2.1.1 Polyphase QMF Bank

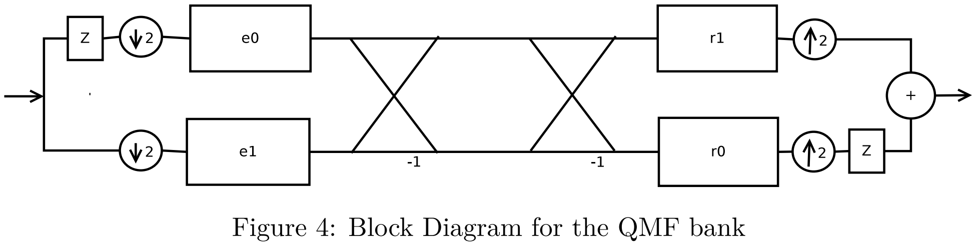


Figure 4: Block Diagram for the QMF bank

The design of the analysis or synthesis QMF bank is realized efficiently using a polyphase decomposition [**?**]. This reduces the number of MAC operations as well as the memory requirement of the overlap-add system for the FIR. The typical polyphase QMF filter bank has been slightly modified in the implementation for efficient realization in code (both at MATLAB level and C) by selecting the odd phase in the decimation rather than the traditional even phase. This requires the use of an advancer in one of the phase input/output rather than a traditional delay element. The logic of perfect reconstruction remains the same. This approach simplifies the addressing logic for the polyphase overlap-add block FIR and is given by equation:

$$X'(z) = \begin{bmatrix} 1 & z \end{bmatrix} \begin{bmatrix} R_1(z^2) & 0 \\ 0 & R_0(z^2) \end{bmatrix} F_2 F_2^{-1} \begin{bmatrix} E_0(z^2) & 0 \\ 0 & E_1(z^2) \end{bmatrix} \begin{bmatrix} z \\ 1 \end{bmatrix} X(z)$$

Where $E_0(z)$ and $E_1(z)$ are the even/odd polyphase components of the analysis filter and $R_0(z)$ and $R_1(z)$ are the even/odd polyphase components of the synthesis filter. F_2 and F_2^{-1} are the size 2 FFT and IFFT matrix respectively.

2.1.2 Delay

Each QMF polyphase perfect reconstruction filter bank adds a delay of (M-2) where M is the length of the analysis/synthesis filter. Hence the total delay from input to output is:

$$\begin{aligned} \text{total delay} &= 2 * (2 * (M-2) + (M-2)) + (M-2) \\ &= 7 * (M-2) \\ &= 7 * (40-2) = 266 \text{ Samples} = 33.2 \text{ ms at 8 ksps} \end{aligned}$$

This however is the total delay of the 5 band non uniform filter bank tree. The two branches at highest decimation rate of 1/8 have the additional delay of (M-2) compared to the remaining three bands that are at decimation rate 1/4. Hence prior to synthesis the data samples in these three branches must be delayed by (M-2) as shown in figure ??. At the MATLAB level this is achieved simply appending (M-2) zeros in the top 3 bands of the final analysis filter bank, i.e. to the total data vector (NOT to each block), before passing it to the ADPCM encoder. This ensures no compensation is needed at the synthesis side. The C implementation however handles this in a block wise manner by employing circular delay line read/write buffer at the bitpacking stage detailed in section ??.

2.2 ADPCM Encoder/Decoder

The ADPCM codec is implemented in the `ADPCM_coder.m` and `ADPCM_decoder.m` functions. For a real time implementation, the data needs to be processed as it arrives. This requires buffering an appropriate number of input samples (called a block) and then processing blockwise.

To ensure the iterative blockwise processing is equivalent to the non-blockwise case, after the processing of a block has been completed, the history of three elements needs to be maintained and passed to the next block as chunk variables. This additional scratch memory requirement stores the last computed step value, quantised output $z(n)$, and $x^*(n)$.

2.2.1 Optimization of Predictor Coefficients

The design parameters, (i.e. the prediction coefficient μ of each of the five sub-bands) need to be chosen such that the prediction error (and hence the number of quantization bits needed to represent it) is minimised. The coefficients were obtained using MATLAB's `fmincon` function, using the SNRseg result as the optimization criterion. The `fmincon` solver was chosen for this optimization problem as the speech coder is a nonlinear multivariable function with constraints on the values of μ (namely, a lower and upper bound of 0 and 1, respectively).

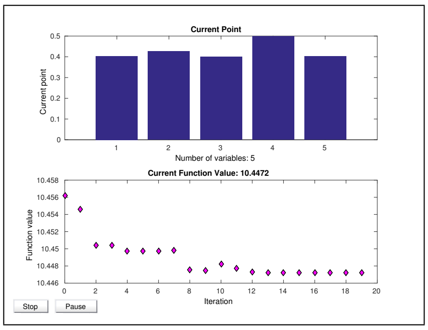
The optimization training data used was a concatenation of all eight audio files. This new file, `allspeech`, had the periods of silence removed (for example, between sentences and at the beginning and end of the training data files), to ensure that the final values of μ are based on real speech only. The optimization was run with different starting points ($\mu = 0.4, 0.5, 0.9$). There are many local minima in the function, but the difference in SNR between the different minima found below was small.

Table ?? below shows the final values of μ for the three different starting points. The best SNRseg result of 19.55dB was found when a search starting point of $\mu = 0.4$ was used. The final optimal μ values found by the optimizer are highlighted in bold in Table ??. These are the values used in the encoder.

Figure ?? shows the final result of the fmincon function for the $\mu = 0.4$ starting point. The function value to be minimised was $f = 30 - SNRseg$. The final value shown in the figure is for an SNR of 19.55dB. Also shown are the PESQ and SNRseg results obtained after finding the optimal μ predictor coefficients across all input files.

Start Point	μ_1	μ_2	μ_3	μ_4	μ_5	SNRseg
0.4	**0.4035**	**0.4259**	**0.4004**	**0.4016**	**0.4993**	**19.55**
0.5	0.5022	0.4851	0.4710	0.4586	0.5000	19.44
0.9	0.7443	0.1250	0.1992	0.5227	0.5251	19.08

Table 2: Results of μ search with fmincon function for different starting points.



Input file	PESQ	SNRseg
belasting	3.645	19.455
bir	4.108	20.870
f116	3.458	15.884
f216	3.397	17.687
m116	4.066	15.427
m216	4.017	15.643
words_f	3.854	20.526
words_m	4.153	20.155

Table 3: Left: Result of fmincon optimizer. Right: PESQ and SNRseg results for fixed-point coder on 8 input files using optimal μ values.

2.2.2 Quantizer

The output of the quantizer z(n) is the error signal divided by a step size. For each nbit-quantized output z(n), the corresponding step-size multiplier is looked up and the step-size, Δ, for the next sample is computed by multiplying the previous step-size by this function of the quantized error signal. Table **??** shows the step-size multipliers used for each coder type, nbit. This function has the property of calling for fast increases and slow decreases of step-size. [**?**]

nbit	Step Size Multiplier
2	0.8, 1.6
3	0.9, 0.95, 1.5, 2.75
4	0.9, 0.9, 0.9, 0.9, 1.2, 1.6, 2.0, 2.4
5	0.9, 0.9, 0.9, 0.9, 0.95, 0.95, 0.95, 0.95, 1.2, 1.5, 1.8, 2.1, 2.4, 2.7, 3.0, 3.3

Table 4: Step-size multipliers for different nbit settings.

2.3 Fixed-Point Issues and Solutions

The input wav file data is floating point with dynamic range ±1. As the expected word size of the DSP is 16 bit (for the datapath), the input data Q format is chosen to be q15. The input data is thus transformed to Q15 by multiplying by 32768 and saturating in a signed 16 bit range. The analysis/synthesis filters are also normalized and converted to a q15 format.

2.3.1 FIR Filter in QMF

The FIR filtering operation in the QMF primarily employs a MAC operation. The input to MAC are the filter coefficient and data sample. The MAC thus is assumed to be 40 bit accumulator for q15 input multiplicands. Upon accumulation the final result is rounded and packed down to signed 16 bits by shifting down by 15. Let W be the 40 bit accumulator and x and y be the q15 input multiplicands then:

$$\begin{aligned}
&(\text{step1}) : W = 0 \\
&(\text{step2}) : \sum_{0}^{M-1} W = W + x_k \cdot y_k \\
&(\text{step3}) : out_n = ((W + 16384) >> 15) + overlap\_add\_history_n
\end{aligned}$$

The history samples are also packed down to q15 before storing. Ideally the history samples must be stored in full precision (40 bit), but considering that the DSP may not have 40 bit read/write option or it may be slow compared to the standard 16 bit data read/write, this compromise is made. For large block size > filter length, the effect of the low precision of the overlap add history is minimal.

2.3.2 ADPCM Encoder/Decoder

The ADPCM encoder takes in q15 subband filtered PCM data sample. The constituent variables in equation **??** and **??** are put in respective q formats:

- x (input PCM stream) : q15
- x^* (predicted input value) : q15
- d (prediction error) : q15
- Δ (quantization scale): q15
- d' (dequantized prediction error): q15
- z (quantized prediction error (output)): q15.0 with saturation to required number of bits
- $F(|z|)$ (step adaptation) : q2.13 to meet the range in table **??**

The multiplication for the equations **??** and **??** are performed with rounding and packing to meet the correct output format. The Q format for the decoder variables are essentially the same.

2.3.3 $\frac{1}{x}$ Computation via Taylor Series

The ADPCM encoder requires a step of division in equation set **??**. For floating point implementation this is not a problem, however for the fixed point computation division presents a problem. As all the input numbers have to be in proper format, using integer division for step 5 of equation set **??** will yield in severe performance degradation. Instead the numerator is multiplied with the high precision reciprocal of the denominator. Towards this end the reciprocal $\left(\frac{1}{\Delta}\right)$ has to be computed. The input number Δ is in range(q15)/$\{0\}$. Hence at one extreme the maximal precision output (16 bit) is q15.0 for input ± 0x0001 and q1.14 for input in the range ± [0x4000 , 0x7fff]. Hence the reciprocal function is implemented to yield maximal precision output along with the applicable scale factor via the taylor series approximation to $\frac{1}{x}$ around a point x_0:

$$f(x) = \frac{1}{x} \approx \frac{1}{x_0} + \left(-\frac{1}{x_0^2}\right) \cdot (x - x_0)$$

Let the input number be xin and let x0 be a set of 32 values in range 0.5–1 at midpoints of the bins, then

- normalize the input number to range 0.5–1 and remember shift factor S
- compute the 5 bit index using bits 13 downto 9 of the normalized number (to select the closest available approximation point)
- compute $\delta = (x - x_0)$ as (xin & 0x00FF) − (1 << 8)
- compute the reciprocal as:

  $$y = \text{one\_by\_x0[index]} + (\text{one\_by\_x0\_sq[index]} \cdot \delta + 16384) >> 15$$

  where one_by_x0 is 32 entry table of first term of Taylor series computed at set x0 and one_by_x0_sq is 32 entry table of the second Taylor series term computed at set x0 both in format q2.13
- return y and the shift factor S

The output is thus in format q(2+S).(13-S). Hence multiplication of a q15 number with y should be rounded down by (15+13-S) to yield output in q15.0. This yields the result z in equation **??** in the correct format.

2.4 Testbench

The MATLAB testbench `codec_BlocProc_POC.m` in folder '`./MATLAB/tests`' is the main testbench for the full codec implementation. This script implements the block processing subband decomposition, encoder-decoder and synthesis blocks and generates performance measures: PESQ and SNRSEG for the input wav file. The script can be run in two modes by setting the '`mode`' variable equal to: '1' → floating point or '0' → fixed point. For the floating point mode, as the computation are in full precision the expected result is a PESQ score of 4.5 (when top band has non zero bit allocation otherwise slightly lower) implying perfect reconstruction. For mode 0, as all the intermediate computation is fixed point with the encoded output of each band quantized to the specified number of bits, the results are, though data dependent, in acceptable range of PESQ score. The block size is fixed to the minimum size of 24 and can be changed. The testbench verifies the golden reference MATLAB implementation of the QMF filter bank and the ADPCM encoder decoder modules and can be used to develop the next phase, i.e. in C.

3 C Implementation

With the MATLAB functions as reference the C development is taken up in a bottom up modular manner. The development is broken in three levels. The Level 1 modules are the leaf modules that are combined to form the modules at higher level. Each higher level encapsulates a particular functionality. The QMF analysis/synthesis filter, the ADPCM encoder/decoder and the Bit packer/depacker constitute the level 1 modules. The full subband analysis/synthesis filter tree that includes its constituent QMF filters is at level 2, and at level 3 the full subband coder/decoder module constitutes ADPCM encoder/decoder and Bit packer/depacker from level 1 and the full subband analysis/synthesis filter tree from level 2.

As a rule only level 1 module object structures have memory buffer arrays statically declared. Level 2 and 3 modules instantiate level 1,2 modules inside their respective object structures and in turn have access to their buffers via dereferencing internal pointers. The figure **??** and **??** capture the object hierarchy. The arrows capture the internal pointers set up by the initialization (from right to left) and the the order of execution (from left to right).

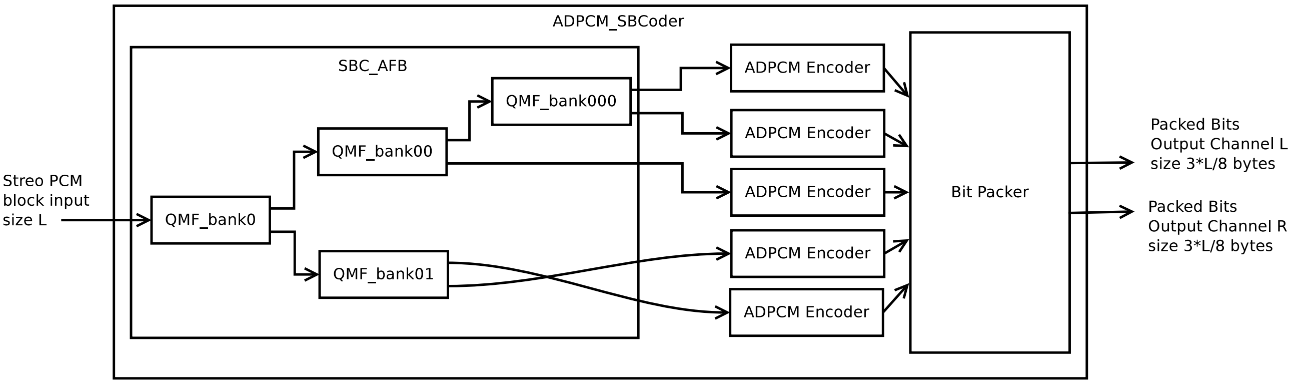


Figure 5: Code Architecture for ADPCM Subband Encoder

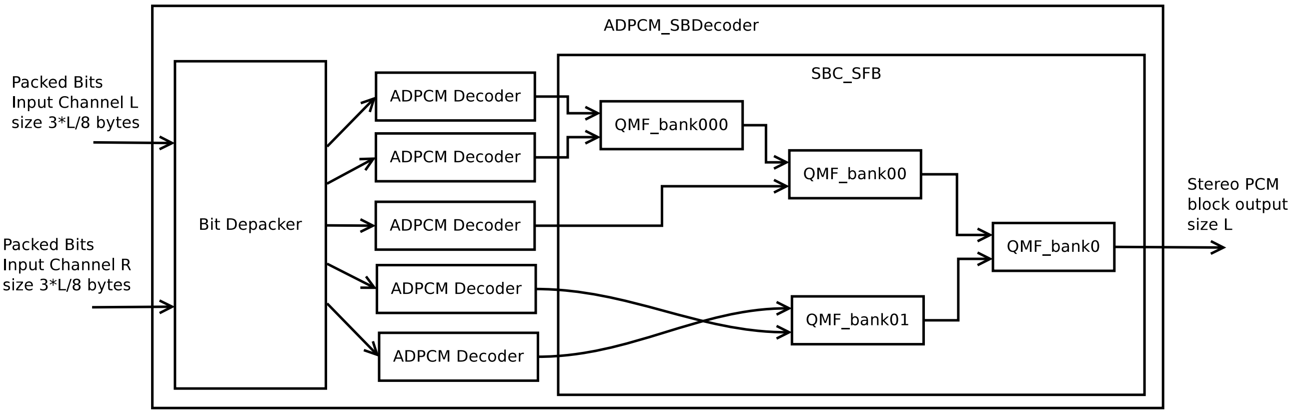


Figure 6: Code Architecture for ADPCM Subband Decoder

This approach has two benefits: first, the bottom up approach allows unit test driven development of the modules. Second, the object oriented development results into simple integration of complex functionality going upwards in the design. Each module then comes with a `init` routine and `process` routine. The `init` routine sets up the internal data structures by connecting the pointers to outside arrays to the internal pointers and initializing arrays to 0. Hence the `init` routine takes as an argument the pointer to the module object, the set of output pointers, other parameters. The `process` routine takes exactly one argument, i.e. the pointer to the module object.

For level 2,3 modules the robust method of initialization is from right to left, i.e. the module last to be written into (invoked) must be initialized first. This to make sure all the internal pointers are set up and point to the correct internal modules/arrays. Once the initialization is complete (done once in the beginning of the program) the process call can be invoked in the logical order (left to right in figure above) with no arguments required.

3.1 Level 1 Modules

The level 1 modules implement the bottom most level of functionality. Level 1 modules thus, may contain array buffers to store temporary data such has history buffers required to implement overlap-add FIR.

3.1.1 QMF Analysis/Synthesis Bank

The QMF analysis/synthesis filters are implemented with the following object structure. The table **??** and **??** describe the constituent elements of the module object structure and the `init` and `process` routines.

QMF Analysis class (QMFA_bank_t)						
Members			Functions			
type	name	description		type	name	description
aud_data_t	history	scratch / history array buffer	QMFA_bank_init	QMFA_bank_t *	QMFA_bank_obj	QMF Analysis object
aud_data_t	data_in_buffer	input buffer		aud_data_t *	data_out_LO	data out low branch ptr
aud_data_t *	data_out_LO_ptr	pointer to data out low branch		aud_data_t *	data_out_HI	data out high branch ptr
aud_data_t *	data_out_HI_ptr	pointer to data out high branch		sint16 *	filter	pointer to filter
sint16 *	flt_ptr	pointer to filter		sint32	block_len	block length
sint32	blk_len	block length	QMFA_process	QMFA_bank_t *	QMFA_bank_obj	QMF Analysis object

Table 5: QMF Analysis class

The `QMFA_bank_init` function initializes the `history` buffer to 0 and connects the output array pointers `data_out_LO` and `data_out_HI` provided to the internal pointers. Thus the internal pointers now point to the target buffers. It also initializes the filter pointer to point to the analysis filter array and sets the block length variable. The `QMFA_bank_process` call thus processes one block of input data updates its history buffer and writes out the bank outputs to the arrays pointed by the internal data out pointers. It is assumed that the internal buffer has been written into and input data is ready before the process call. To run the test enter directory '`./dev/co_testing/unit_tests/QMF_Analysis`' and follow the instructions in the `README` for unit testing.

QMF Synthesis class (QMFS_bank_t)						
Members			Functions			
type	name	description		type	name	description
aud_data_t	history	scratch / history array buffer	QMFS_bank_init	QMFS_bank_t *	QMFS_bank_obj	QMF Synthesis object
aud_data_t	data_in_buffer_LO	input buffer: low branch		aud_data_t *	data_out_ptr	pointer to data out
aud_data_t	data_in_buffer_HI	input buffer: high branch		sint16 *	filter	pointer to filter
aud_data_t *	data_out_ptr	pointer to data out		sint32	blk_len	block length
sint16 *	flt_ptr	pointer to filter	QMFS_process	QMFS_bank_t *	QMFS_bank_obj	QMF Synthesis object
sint32	blk_len	block length				

Table 6: QMF Synthesis class

The `QMFS_bank_init` function initializes the `history` buffer to 0 and connects the output array pointer `data_out_ptr` provided to the internal pointer. Thus the internal pointer now point to the target buffer. It also initializes the filter pointer to point to the synthesis filter array and sets the block length variable. The `QMFS_bank_process` call thus processes one block of input data updates its history buffer and writes out the output to the array pointed by the internal data out pointers. It is assumed that the internal buffer has been written into and input data is ready before the process call. To run the test enter directory '`./dev/co_testing/unit_tests/QMF_Synthesis`' and follow the instructions in the `README` for unit testing.

3.1.2 ADPCM Encoder/Decoder

The ADPCM Encoder and Decoder are implemented with the following object structure. The table ?? and ?? describe the constituent elements of the module object structure and the `init` and `process` routines.

ADPCM Encoder class (ADPCM_enc_t)						
Members			Functions			
type	name	description		type	name	description
aud_data_t	step	history for step value	ADPCM_enc_init	ADPCM_enc_t *	ADPCM_enc_obj	ADPCM encoder object
aud_data_t	x_star	history for x* value		adpcm_data_t *	data_out	pointer to data output
adpcm_data_t	z_N	history for quantized output		sint16	mu	mu value
aud_data_t	data_in_buffer	PCM input data array		sint16	nbit	number of output bits
sint16	mu	mu value		sint16	blk_len	block length
sint16	nbit	number bits for output	ADPCM_enc_process	ADPCM_enc_t *	ADPCM_enc_obj	ADPCM encoder object
sint16	blk_len	block length				
sint16 *	stepTable_ptr	pointer to step table				
adpcm_data_t *	data_out_ptr	pointer to adpcm output				

Table 7: ADPCM Encoder class

The `ADPCM_enc_init` function initializes the `z_N` and `x_star` history variables to 0 and the `step` value to 16384. The output array pointer `data_out_ptr` provided is connected to the internal pointer. Thus the internal pointer now points to the target buffer. It also initializes the step table pointer to point to the correct table of step multipliers depending on the number of output bits required. The `ADPCM_enc_process` call thus processes one block of input data and implements the ADPCM encoder operations as described in section ??, and saturating the output depending on `nbit`. The inverse step operation is performed by a `RECIP` function as described in section ??. The process call then updates the history variables for the next block to use and writes out the output to the array pointed by the internal data out pointer. It is assumed that the internal buffer has been written into and input data is ready before the process call. The encoder delay handling is performed entirely in the bit packer function. To run the test enter directory '`./dev/co_testing/unit_tests/ADPCM_Enc`' and follow the instructions in the `README` for unit testing.

ADPCM Decoder class (ADPCM_dec_t)						
Members			Functions			
type	name	description		type	name	description
aud_data_t	step	history for step value	ADPCM_dec_init	ADPCM_dec_t *	ADPCM_dec_obj	ADPCM decoder object
aud_data_t	x_star	history for x* value		aud_data_t *	data_out	pointer to data output
aud_data_t	z_N	history for quantized output		sint16	mu	mu value
adpcm_data_t	data_in_buffer	adpcm input data array		sint16	nbit	number of output bits
sint16	mu	mu value		sint16	blk_len	block length
sint16	nbit	number bits for output	ADPCM_dec_process	ADPCM_dec_t *	ADPCM_dec_obj	ADPCM decoder object
sint16	blk_len	block length				
sint16 *	stepTable_ptr	pointer to step table				
aud_data_t *	data_out_ptr	pointer to PCM data output				

Table 8: ADPCM Decoder class

The `ADPCM_dec_init` function initializes the history variables and connects the output array and step table pointers in a similar way to the `ADPCM_enc_init` function. The `ADPCM_dec_process` call thus processes one block of ADPCM input data and implements the ADPCM decoder operations as described in section ??. The process call then updates the history variables for the next block to use and writes out the output to the array pointed by the internal data out pointer. To run the test enter directory '`./dev/co_testing/unit_tests/ADPCM_Dec`' and follow the instructions in the `README` for unit testing.

3.1.3 Bit Packer/Depacker

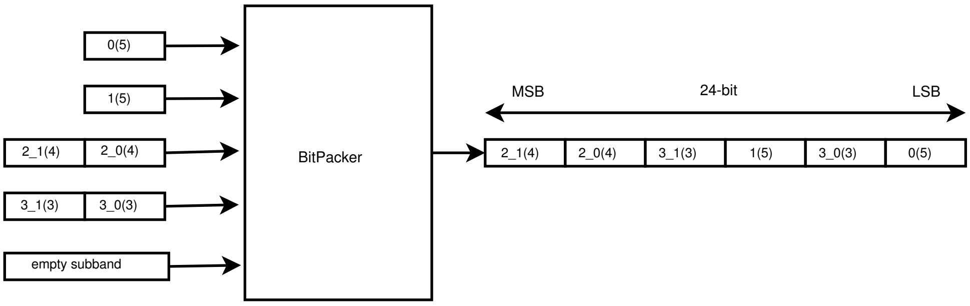


Figure 7: Bit Packer Data Output Format

The bit packer and depacker are implemented with the following object structure. The table ?? and ?? describe the constituent elements of the module object structure and the `init` and `process` routines.

The bit packer function is used to combine the outputs of the four enabled subbands into three bytes per channel. The non-uniform QMF tree structure in use means that the lower two subbands are decimated a factor of 2 more than the upper three subbands. Therefore the output of the ADPCM encoders on subbands 2 and 3 will be at twice the frequency of those on the lower two subbands. The bit packer function implements delay buffers on subbands 2 and 3 in order to compensate for this delay as mentioned in section ??. Each of these bands is delayed by $M-2$ samples, where M is the filter length. As a result, each 24-bit output from the bit packer consists of one output from each of subbands 0 and 1 and two outputs from subbands 2 and 3 as shown in figure ??.

Bit Packer class (Bitpack_t)						
Members			Functions			
type	name	description		type	name	description
adpcm_data_t	inp_band0	Subband 0 ADPCM input array	BitPacker_init	BitPack_t *	BitPack_obj	Bit Packer object
adpcm_data_t	inp_band1	Subband 1 ADPCM input array		bitstream *	bitsout_L	
adpcm_data_t	inp_band2	Subband 2 ADPCM input array		bitstream *	bitsout_R	
adpcm_data_t	inp_band3	Subband 3 ADPCM input array				
adpcm_data_t	inp_band4	Subband 4 ADPCM input array				
adpcm_data_t	delay_buffer2	Delay buffer array for subband 2				
sint16	delay_buffer2_wr					
sint16	delay_buffer2_rd					
sint16	delay_buffer2_lim					
adpcm_data_t	delay_buffer3	Delay buffer array for subband 3				
sint16	delay_buffer3_wr		BitPacker_process	BitPack_t *	BitPack_obj	Bit Packer object
sint16	delay_buffer3_rd					
sint16	delay_buffer3_lim					
bitstream *	out_ptr_L	Left channel output				
bitstream *	out_ptr_R	Right channel output				

Table 9: Bit Packer class

The BitPack class contains 5 input `adpcm_data_t` stereo ADPCM types, two delay buffers, and pointers to 3 output bytes per channel. The `Bitpacker_init` function initializes the input arrays and delay buffers to zero. The `Bitpacker_process` call copies data from the input buffers into the delay buffers for subbands 2 and 3, constructs three output bytes for each channel as described in figure ??, and then writes them out to the bitpacker object. To run the unit test enter directory '`./dev/co_testing/unit_tests/Bitpacker`' and follow the instructions in the `README` for unit testing.

Bit Depacker class (BitDepack_t)						
Members			Functions			
type	name	description		type	name	description
bitstream	inp_buffer0	Left channel packed data input	BitDepacker_init	BitDepack_t *	BitDepack_obj	Bit Depacker object
bitstream	inp_buffer1	Right channel packed data input		adpcm_data_t *	band0_out	Subband 0 output pointer
adpcm_data_t *	out_band0_ptr	Subband 0 ADPCM output		adpcm_data_t *	band1_out	Subband 1 output pointer
adpcm_data_t *	out_band1_ptr	Subband 1 ADPCM output		adpcm_data_t *	band2_out	Subband 2 output pointer
adpcm_data_t *	out_band2_ptr	Subband 2 ADPCM output		adpcm_data_t *	band3_out	Subband 3 output pointer
adpcm_data_t *	out_band3_ptr	Subband 3 ADPCM output		adpcm_data_t *	band4_out	Subband 4 output pointer
adpcm_data_t *	out_band4_ptr	Subband 4 ADPCM output	BitDepacker_process	BitDepack_t *	BitDepack_obj	Bit Depacker object

Table 10: Bit Depacker class

3.2 Level 2 Modules

3.2.1 Subband Analysis/Synthesis Tree

The Subband filter bank tree is constructed in the `SBC_AFB` and `SBC_SFB` object structures respectively. Each instantiates the four QMF analysis/synthesis filter banks respectively required to form the tree structure.

Subband Analysis Filter Bank class (SBC_AFB_t)						
Members			Functions			
type	name	description		type	name	description
QMFA_bank_t	QMF_bank0	bank0 instance	SBC_AFB_init	SBC_AFB_t *	SBC_AFB_obj	Analysis Filter Bank object
QMFA_bank_t	QMF_bank00	bank00 instance		aud_data_t *	out_buffer0_0_0	points to band0 encoder input
QMFA_bank_t	QMF_bank01	bank01 instance		aud_data_t *	out_buffer0_0_1	points to band1 encoder input
QMFA_bank_t	QMF_bank000	bank000 instance		aud_data_t *	out_buffer0_1	points to band2 encoder input
aud_data_t *	inp_buffer	points to QMF_bank0 input		aud_data_t *	out_buffer1_0	points to band4 encoder input
sint32	inp_blk_len	input block length		aud_data_t *	out_buffer1_1	points to band3 encoder input
				sint16 *	filter	
				sint32	inp_blk_len	
			SBC_AFB_process	SFC_AFB_t *	SBC_AFB_obj	Analysis Filter Bank object

Table 11: Subband Analysis Tree class

The `SBC_AFB` object constitutes of four instances of the QMF analysis banks. The `SBC_AFB_init` function initializes the constituent banks beginning with the rightmost (the bank last written to), i.e. `bank000` and works its way towards `bank0`. This initialization connects the output pointers to the correct input buffers to form the analysis filter bank tree. The `SBC_AFB_process` call thus processes one block of input data and invokes the constituent QMF banks in their logical order to produce 5 bands of subband decomposed output that are written into the buffers pointed by the output pointers. It is assumed that the internal buffer has been written into prior to the process call of the module. To run the test enter directory '`./dev/co_testing/unit_tests/QMF_full_FB`'.

Subband Synthesis Filter Bank class (SBC_SFB_t)						
Members			Functions			
type	name	description		type	name	description
QMFS_bank_t	QMF_bank000	bank000 instance	SBC_SFB_init	SBC_SFB_t *	SBC_SFB_obj	Synthesis Filter Bank object
QMFS_bank_t	QMF_bank00	bank00 instance		aud_data_t *	data_out	points to data output
QMFS_bank_t	QMF_bank01	bank01 instance		sint16 *	filter	points to filter
QMFS_bank_t	QMF_bank0	bank0 instance		sint32	out_blk_len	output block length
aud_data_t *	in_buffer0_0_0	points to input buffer0 of bank000	SBC_SFB_process	SBC_SFB_t *	SBC_SFB_obj	Synthesis Filter Bank object
aud_data_t *	in_buffer0_0_1	points to input buffer1 of bank000				
aud_data_t *	in_buffer0_1	points to input buffer1 of bank00				
aud_data_t *	in_buffer1_0	points to input buffer0 of bank01				
aud_data_t *	in_buffer1_1	points to input buffer1 of bank01				
sint32	out_blk_len	output block length				

Table 12: Subband Synthesis Tree class

The `SBC_SFB` object constitutes of four instances of the QMF synthesis banks. The `SBC_AFB_init` function initializes the constituent banks beginning with the rightmost (the bank last written to), i.e. `bank0` and works its way towards `bank000`. This initialization connects the output pointers to the correct input buffers to form the synthesis filter bank tree. The `SBC_SFB_process` call thus processes one block of input data and invokes the constituent QMF banks in their logical order to combine 5 bands to the synthesised output that is written into the buffer pointed by the output pointer. It is assumed that the internal buffers have been written into prior to the process call of the module. To run the test enter directory '`./dev/co_testing/unit_tests/QMF_full_FB`'.

3.3 Level 3 Modules

3.3.1 ADPCM Subband Coder/Decoder

The level 3 module captures the full functionality of the ADPCM Subband coder/decoder. The structure is captured in figure **??** for the subband encoder and figure **??** for the subband decoder.

ADPCM Subband Coder class (ADPCM_SBCoder_t)						
Members			Functions			
type	name	description		type	name	description
SBC_AFB_t	SBC_AFB	analysis filterbank object	ADPCM_SBCoder_init	ADPCM_SBCoder_t *	ADPCM_SBCoder_obj	ADPCM Subband Coder object
ADPCM_enc_t	ADPCM_ENC_band0	instantiate band0 coder		bitstream *	bitsout_L	left channel ADPCM output
ADPCM_enc_t	ADPCM_ENC_band1	instantiate band1 coder		bitstream *	bitsout_R	right channel ADPCM output
ADPCM_enc_t	ADPCM_ENC_band2	instantiate band2 coder		sint32	inp_block_len	input block length
ADPCM_enc_t	ADPCM_ENC_band3	instantiate band3 coder	ADPCM_SBCoder_process	ADPCM_SBCoder_t *	ADPCM_SBCoder_obj	ADPCM Subband Coder object
ADPCM_enc_t	ADPCM_ENC_band4	instantiate band4 coder				
BitPack_t	BitPack	instantiate bitpacker				
aud_data_t *	inp_buffer	points to input buffer				
sint32	blk_len	block length				

Table 13: ADPCM Subband Coder class

The `ADPCM_SBCoder` object constitutes of the `SBC_AFB` filter bank tree and five `ADPCM_Encoder` objects and the `BitPacker` object. The `ADPCM_SBCoder_init` function initializes the constituent modules from right to left starting with the bitpacker followed by the ADPCM encoders and then the filter bank tree object. The output pointer points to the two bitbuffers provided as argument to the init function. The `ADPCM_SBCoder_process` call thus processes one block of input data and produces 2 streams of packed encoded bits and writes out to the output buffers pointed by the output pointers. It is assumed that the internal data buffer has been written into and the input data is read before the process call. To run the test enter directory '`./dev/co_testing/CODEC`' and run the full test following the instructions in `README`.

The `ADPCM_SBDecoder` object constitutes of the `SBC_SFB` filter bank tree and five `ADPCM_Decoder` objects and the `BitDepacker` object. The `ADPCM_SBDeoder_init` function initializes the constituent modules from right to left starting with the synthesis filter bank tree followed by the ADPCM decoders and then the bit depacker object. The output pointer points to the output buffer provided as argument to the init function. The `ADPCM_SBDecoder_process` call thus processes one block of input data and produces synthesised and writes out to the output buffer pointed by the output pointer. It is assumed that the internal data buffers have been written into and the input data is read before the process call. To run the test enter directory '`./dev/co_testing/CODEC`' and run the full test following the instructions in `README`.

ADPCM Subband Decoder class (ADPCM_SBDecoder_t)						
Members			Functions			
type	name	description		type	name	description
SBC_SFB_t	SBC_SFB	synthesis filterbank object	ADPCM_SBDecoder_init	ADPCM_SBDecoder_t *	ADPCM_SBDecoder_obj	ADPCM Subband Decoder object
ADPCM_dec_t	ADPCM_DEC_band0	instantiate band0 decoder		aud_data_t	data_out_ptr	points to output
ADPCM_dec_t	ADPCM_DEC_band1	instantiate band1 decoder		sint32	out_block_len	output block length
ADPCM_dec_t	ADPCM_DEC_band2	instantiate band2 decoder	ADPCM_SBDecoder_process	ADPCM_SBDecoder_t *	ADPCM_SBDecoder_obj	ADPCM Subband Decoder object
ADPCM_dec_t	ADPCM_DEC_band3	instantiate band3 decoder				
ADPCM_dec_t	ADPCM_DEC_band4	instantiate band4 decoder				
BitDepack_t	BitDepack	instantiate bit depacker				
bitstream *	inp_buffer_L	left channel input				
bitstream *	inp_buffer_R	right channel input				
sint32	blk_len	block length				

Table 14: ADPCM Subband Decoder class

3.4 Data Structure Size

Module	Instances	Total Size in Bytes
DSP_ADPCM_SBCoder_t	1	**15948**
|– DSP_SBC_AFB_t	1	12936
|– DSP_QMFA_t	4	12928
|– DSP_ADPCM_enc_t	5	1280
|– DSP_BitPacker_t	1	1184
ADPCM_SBDecoder_t	1	**14204**
|– DSP_BitDePacker_t	1	276
|– DSP_ADPCM_dec_t	5	980
|– DSP_SBC_SFB_t	1	12936
|– DSP_QMFS_t	4	12912

Table 15: Data Structure Size for Block Size = 336 Stereo Samples

3.5 Full Codec

3.5.1 Test Case

The `main.c` located in the `'./co_testing/CODEC/'` directory instantiates the encoder and decoder objects `ADPCM_SBCoder` and `ADPCM_SBDecoder`, loads the required block size number of input samples from file, and fills the subband coder input buffer accordingly before calling the `ADPCM_SBCoder_process`. The bitstream from the encoder is then used to fill the decoder input buffer and the `ADPCM_SBDecoder_process` is called. When this is completed, the decoded PCM output is read out to the reconstructed buffer, and then written to file using the `wavpcm_output_write` function.

3.5.2 Results

The final PESQ results for the C implementation are listed in table ??. The PESQ score has been computed with the block size set to various sizes from the minimum of 48 to 1024. There is as such no maximum block size and would be constrained by the application latency requirements and also the crypto input block size. The PESQ scores increases marginally with block size as the effect of low precision history in overlap add reduces with increasing block size.

BLOCK_SIZE	48	80	256	528	1024
	PESQ				
belasting	3.619	3.632	3.627	3.627	3.620
bir	4.090	4.089	4.090	4.090	4.089
f116	3.484	3.451	3.447	3.454	3.443
f216	3.415	3.413	3.432	3.430	3.433
m116	4.039	4.048	4.059	4.062	4.059
m216	4.014	4.014	4.012	4.043	4.013
words_f	3.818	3.810	3.823	3.814	3.813
words_m	4.119	4.128	4.133	4.116	4.127

Table 16: Final PESQ results for C implementation

3.6 Co-Testing Framework

The co-testing framework is a set of makefiles, `MATLAB` scripts and C testwrappers that work in congruence to perform automated testing of the modules via unit test and the final codec test. The framework consists of a `MATLAB` script that implements the functionality of the module under test using the `MATLAB` functions created before and produces the expected outputs and writes the input/output pair to test vector data files. The C test wrapper initializes a testcase by loading the testvectors and invokes the C module under test and verifies the outputs with the expected outputs. The criteria for '`TEST PASS`' is bitexactness with respect to the `MATLAB` fixed point outputs. The hierarchy of makefiles enable invoking unit tests one at a time or in a automated run over all unit tests and documents the results. The testing framework also contains a test for the full codec. This test can take in a input wav file as argument and run the full coder decoder and produce the output for which the PESQ score is measured. The PESQ measurement is done via PESQ reference software [?] compiled to form a standalone binary that can measure PESQ score given two input wav files to compare. The details of using the testing framework is provided in detail in the `README` in the top folder.

3.6.1 Software Package for C and MATLAB Co-Implementation

```
PROJECT DIRECTORY STRUCTURE

dev
|-- co_testing     // Top directory for testing
|   |-- makefile      // Top level makefile to be used as described above
|   |-- CODEC        // Main subband codec testcase
|   |-- unit_tests
|       |-- ADPCM_Dec      // unit test case to test the adpcm decoder
|       |-- ADPCM_Enc      // unit test case to test the adpcm decoder
|       |-- BitPacker      // unit test case to test the bit packing and depacking
|       |-- QMF_Analysis   // unit test case for QMF analysis filter
|       |-- QMF_Synthesis  // unit test case for QMF synthesis filter
|       |-- QMF_2bank_FB   // unit test case to test the 2 bank perfect reconstruction filter bank
|       |-- QMF_full_FB    // unit test case to test the full 5 band perfect reconstruction filter bank
|
|-- C_src
|   |-- ADPCM [.c/.h]     // ADPCM encoder and decoder implementation
|   |-- ADPCM_SBC [.c/.h] // Full Subband coder decoder with bitpacking and depacking implementation
|   |-- BitPacker [.c/.h] // bit packer and depacker
|   |-- common.h          // common #defines and macros
|   |-- globals.h         // common #defines for codec test code
|   |-- QMF [.c/.h]       // QMF analysis and synthesis filter implementation
|   |-- SBC [.c/.h]       // Full subband analysis and synthesis filter implementation
|   |-- wavpcm_io [.c/.h] // wav file read/write utility
|
|-- MATLAB
|   |-- src
|   |   |-- ADPCM_coder.m   // reference implementation for ADPCM coder (floating and fix point)
|   |   |-- ADPCM_decoder.m // reference implementation for ADPCM decoder (floating and fix point)
|   |   |-- QMF_analysis.m  // reference implementation for QMF analysis filter bank
|   |   |-- QMF_synthesis.m // reference implementation for QMF synthesis filter bank
|   |-- tests
|   |   |-- codec_BlockProc_POC.m // main full blockwise SB-ADPCM coder decoder test case
|   |-- utils // miscellaneous utilty matlab functions
|
|-- scripts          // common makefile
|-- training_data // wav files for testing
|-- ITU_PESQ      // PESQ measurement software as provided by ITU
|-- README        // instructions towards running tests
|-- tags          // Tags file can be loaded into vim to navigate the code
```

4 Integration With Encryption

The C reference code for the full codec example mentioned above was extracted out to build a standalone test case (new test wrapper only) to integrate with the crypto functions to yield a unified test case for subband coding with encryption. The crypto code has been taken from Crypto-Group-6. The code received implements similar high level API for encryping/decrypting the bitpacked subband-ADPCM data block. The block size was adapted to fit the block size requirement of the crypto implementation. The combined code was submitted as a standalone package with all unit tests and co-testing framework removed.

The encryption-decryption processing was thus successfully added between the encoder and decoder. Thus the encryption received bitpacked subband ADPCM coded bits for both channels resulting into encrypted bits, forming the transmitter side. The encrypted bits are decrypted to yield the original bitpacked data block that is passed to the ADPCM decoder and results are found to be bit-exact with respect to the flow without encryption.

5 DSP Implementation

With the verified C reference available, the next phase of development is to port the C code to the specific DSP platform. The objective of the C reference was to get the kernel functionality and develop data structures/objects and function call hirerarchies. As only level-1 modules above implement leaf node processing, i.e. the QMF filter bank or ADPCM encoder/decoder, only the level 1 modules are taken up for optimizations. The level 2 and higher modules do not require optimizations as such as they do not implement a major signal processing task but join the lower level modules together to achieve full functionality.

5.1 Test Setup

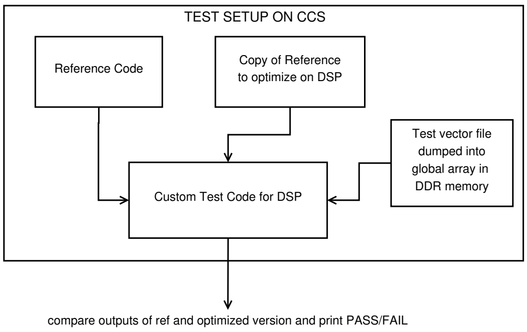

TEST SETUP ON CCS
Reference Code
Copy of Reference to optimize on DSP
Test vector file dumped into global array in DDR memory
Custom Test Code for DSP
compare outputs of ref and optimized version and print PASS/FAIL

Figure 8: Optimization and Testing setup in Code Composer Studio

The source code minus the test framework was imported into a Code-Composer-Studio project with the TI-DSP C674x platform as the target architecture. The code base was included in duplicate with the names of the data structures and associated functions prefixed with 'DSP_' to differentiate from the reference copy. The names of the internals of the data structures need not be changed as they lie in different scopes. The test code instantiates the reference and duplicate copy of the data structure for the encoder and decoder and initializes them respectively. It thus loads both the reference and duplicate implementation input buffers with identical data in each iteration and calls the processing functions. The output of each iteration of the reference is compared with the output of the duplicate copy with strict bit exactness criteria (even 1 bit error is considered as a TEST FAIL). Optionally a top level flag is provided to dump data into file to measure PESQ on the processed data and verify offline the correctness of the code. The test also includes a testvector of stereo samples of one of the test wav files compiled in as a static const array placed in DDR memory, this is to speed up the simulation.

With this setup verified via the PASS/FAIL print on the console and the verification of the dumped data (into txt files), the duplicate copy can be taken up for optimization on the target platform. During this optimization the reference copy is NOT modified and always serves as a golden reference that the optimized kernel is tested against implicitly by the test code.

5.2 Optimization Strategy

The optimizations are carried out with the Texas Instruments documentation for the C674x processor as references [?], [?], [?]. After the first profiling of the reference code it was observed that first kernel for optimization was DSP_QMFA_process. This implements the polyphase FIR analysis filter bank. The original implementation

Function Under Optimization		Inclusive Count Total cycle count[1]				
Encoder	#	Initial	Round 1	Round 2	Round 3	% reduction
DSP_ADPCM_SBCoder_process	1	**103.6k**	**84.3k**	**81.5k**	**34.2k**	**-66.99%**
|– DSP_SBC_AFB_process	1	56.2k	36.9k	34.1k	25.3k	-54.98%
|– DSP_QMFA_process	4	56.2k	36.9k	34.1k	25.2k	-55.16%
|– DSP_ADPCM_enc_process	5	46k	46k	46k	8k	-82.6%
|– DSP_BitPacker_process	1	1.26k	1.26k	1.26k	751	-40.39%
Decoder	#	Initial	Round 1	Round 2	Round 3	% reduction
DSP_ADPCM_SBDecoder_process	1	**63.6k**	**63.6k**	**63.6k**	**27.4k**	**-56.91%**
|– DSP_BitDePacker_process	1	8.26k	8.12k	8.12k	1.44k	-82.56%
|– DSP_ADPCM_dec_process	5	2.3k	2.3k	2.3k	2.3k	0%
|– DSP_SBC_SFB_process	1	53.1k	53.1k	53.1k	23.7k	-55.36%
|– DSP_QMFS_process	4	53k	53k	53k	23.6k	-55.47%

Table 17: Optimization Results for block size 208 stereo samples

was done to have minimal number of multiply-accumulates. It was observed that the low level data structure to encapsulate the stereo data samples was unnecessary inside the kernel as the compiler would treat the constituents separately leading to inefficiency. Hence, it was decided to typecast the pointer to the stereo data structure (of native size 32 bits with two 16 bit signed constituents) as a single 32 bit element. This enabled loading the stereo samples in one cycle and the use of C intrinsics that perform dual multiply. Similar strategy works for the `DSP_QMFS_process`. The core loop count in the implementation was variable (minimal multiply accumulate). It was observed as the loop count logic is fixed once the buffer size and filter length are fixed, the inner loop counts could be precomputed at initialization and stored in the object structure and loaded at run time. This yields significant savings in the core loop condition.

The second kernel to be optimized was the ADPCM encoder. The encoder included the computation of the reciprocal of the step size. In addition to following a similar strategy of accessing the data arrays after typecasting to 32 bit types and use of intrinsics thereof, the recip computation was placed inline manually to cut the overhead of function call in the core loop. Special intrinsics like '`_norm`' were used to compute the leading zeros in the step value in the normalization step described in section **??**.

The third kernel to be optimized was the `DSP_BitDePacker_process`. The core loop involved a sign extension of the depacked bits which was done via a function. The function call for such a basic operation was contributing large cycles in the core loop and hence the `SGN_EXT` function was inlined.

Miscellaneous optimizations included the use of `restrict` local pointers to access the data arrays and `#pragma MUST_ITERATE()` and compiler flags such as `-mt`. The `-mt` compiler option was used to assume no bad alias or loop behavior.

5.3 Observations

- The first bug encountered was that the TI compiler did not align the `aud_data_t` type arrays to 4 byte boundaries. This was done by adding a compiler directive to align the array to 4 byte boundary. The misalignment resulted in poor performance and also errors due to data being written out at an offset.
- When running on the actual TI board, an initialization bug was encountered. The init functions take care of initializing the buffers in the data structures to zero. In two places, namely `DSP_BitPacker_t` and the `DSP_QMFS_t` structure, arrays were not initialized properly due to a typo in the coding of the init functions.
- These bugs were not caught on the simulator as the simulator mimics the RAM as a data structure that happens to get initialized at the start of the simulation. On the real hardware board the RAM may wake up in a random state, causing an uninitialized array to cause unpredictable errors.
- The core loop of the QMF filter bank performs $\frac{M}{2} \cdot \frac{N}{2}$ multiplies per filter per stereo channel, where M is the filter length and N is the block size. The computation is as follows assuming $N > M$

$$
\begin{aligned}
\text{Total MACs in polyphase FIR } &= 2\cdot\left(\sum_{i=1}^{\frac{M}{2}-1} i + \sum_{i=1}^{\frac{N}{2}-\frac{M}{2}+1}\frac{M}{2} + \sum_{i=1}^{\frac{M}{2}-1}(\frac{M}{2}-i)\right)\\
&= 2\cdot\left(\frac{M}{2}\right)\cdot\left(\frac{N}{2}\right)
\end{aligned}
$$

This implies the cycles/mac performance of the FIR kernel is given as

$$
\begin{aligned}
\left[\frac{\text{cycles per call}}{\frac{M}{2}*\frac{N}{2}*(\text{number of phases})*(\text{stereo})}\right] &= \frac{10475}{20*104*2*2}\\
&= 1.25 \text{ cycles per MAC}
\end{aligned}
$$

5.4 Real Time Audio Testing

For the real time audio implementation with integrated encryption, the `init_audio` function contains the `DSP_ADPCM_SBCoder_init` and `DSP_ADPCM_SBDecoder_init` coder and decoder initialization calls as well as the key establishment function from the crypto group. The `process_audio` function is called repeatedly with pointers to new input samples from line in. The function first fills the subband coder input buffer with the new samples from input_buf and then calls the subband coder process `DSP_ADPCM_SBCoder_process`. The `AES128_Encrypt` and `AES128_Decrypt` crypto functions are called to encrypt and decrypt the packed ADPCM data. The decrypted output is then written into the subband decoder input buffer and the `DSP_ADPCM_SBDecoder_process` is called. The decrypted output is then written to the stereo output buffer.

The real time demo was successfully completed and presented to the Teaching Assistants with speech input from training data files as well as speech and music from YouTube. The quality was observed to be good for both inputs including music with some minor suppression of high frequencies in the case of music. The setup was also tested without encryption for different block sizes and was seen to function across a selection of different block sizes.

6 Conclusions

The stereo speech codec was implemented to meet the required output bitrate specification, while achieving high PESQ score. The transition between the different development environments was identified as a challenge, as from MATLAB → C → DSP, the programming environment becomes progressively hostile. This makes it difficult to debug or make core modifications at later stages. This was identified and a strategy was developed to mitigate the challenge:

- First, a lot of effort was put on getting the functionality correct with floating point and its corresponding fixed point version including the implementation of reciprocal, as integer division was identified as a potential problem (section **??**). A significant amount of time was spent on this stage to ensure a solid foundation was available to avoid running into problems in less forgiving development environments
- Second, the C implementation was carried out via a unit testing driven development with **automated** co-testing against MATLAB reference as described in section **??**. Special emphasis was put on data structure hierarchies and object orientation, anticipating DSP platform restrictions
- Third, the DSP implementation was done via first creating a duplicate copy of the code and verifying the test setup, and then optimizing one of the copies (named with prefix DSP_) while comparing the output at each optimization stage. Also the reconstructed audio was written out to file for external verification via PESQ score

This development approach had the aim of proactively preempting potential bugs, via systematic bottom up code structuring and a corresponding application programming interface (API) as described in section **??**. This approach also removed any efforts to refactor/restructure the code for DSP environment. This has been achieved via ad-hoc tools and scripts enabling heterogeneous programming environments to work together. Such development could be made much simpler and systematic if there were a unified environment to allow algorithmic exploration, C coding and platform simulation, all under one high level programming paradigm.