AN2151
Rev. 1, 1/2005

ITU-T G.729A Implementation on the StarCore™ SC140/SC1400 Cores

By Razvan Ungureanu, Bogdan Costinescu, and Costel Ilas

This application note reflects the activity flow and the results of the ITU G.729A implementation on the StarCore™ SC140/SC1400 cores. This project was implemented according to the methodology described in application note AN2094, *ITU-T G.729 Implementation on the StarCore SC140/SC1400 Cores* [3]. Thus, this application note also reflects the viability of the methodology.

CONTENTS

1 G.729A Basics 2
1.1 Encoder Differences Between G.729A and G.729 .2
1.2 Decoder Differences Between G.729A and G729 ..3
1.3 Data Table Differences Between G.729A and G729 4
1.4 Processing Load Estimation 4
2 Implementation Methodology 5
2.1 Porting ITU G.729A Reference Code to the SC140 6
2.2 Project-Level Optimizations 6
2.3 Algorithmic Changes 8
2.4 Function-Level C Optimizations 10
2.5 Assembly Implementation 10
3 Results 12
4 Conclusions 15
5 References 16

freescale™ semiconductor

1 G.729A Basics

The ITU G.729A 8 kbit per second speech codec is a reduced-complexity version of the ITU G.729 CS-ACELP speech vocoder. However, the G.729A vocoder is bit-stream compatible and interoperable with the G.729 vocoder. It operates on 10 ms speech frames with 5 ms look-ahead for linear prediction (LP) analysis. Compared to G.729, the G.729A vocoder requires much less processing power; however, the reconstructed speech quality may not be as good in some cases. **Table 1** shows the differences between the two vocoders in terms of the Mean Opinion Score (MOS), a commonly used test to assess speech quality. The MOS rating ranges from 0 to 5; a MOS of 4 is considered 'toll' quality. The general description of G.729A is similar to the G.729 [3], so the following sections describe only the algorithmic differences between the two vocoders.

Table 1. MOS Results

Vocoder	MOS Result (In Clean Conditions)
ITU G.729	4.125
ITU G.729A	3.7

1.1 Encoder Differences Between G.729A and G.729

The major differences between the two vocoders can be found in the LP analysis quantization and interpolation, perceptual weighting, pitch analysis, adaptive codebook search, and fixed codebook search modules, and the memory update function.

Table 2. Summary of Encoder Differences Between G.729A and G.729

Module	Description
LP analysis quantization and interpolation	The LP to LSP conversion performed by the Az_lsp() function is simplified in two ways—the number of points in which the function is evaluated is reduced from 60 to 50, and the number of intervals for root search is reduced by half. In other respects, the Az_lsp() function is similar to the one in G.729. Interpolation of the LSP coefficients performed in the encod_ld8a() is also similar to G.729 except that only the quantized LP coefficients are interpolated
Perceptual weighting	The code for this functional block is located in the encod_ld8a() function. It uses the Weight_Az(), Residu() and Syn_filt() functions, which are identical to those used in G.729 except that the number of filtering operations is reduced.
Pitch analysis	The open loop pitch lag is computed in the `Pitch_ol_fast()` function, which reduces the complexity of the search for the best adaptive-codebook delay.
Adaptive codebook search	The computation of the impulse response of weighted synthesis filter and the target vector for pitch search, both encoded in `encod_ld8a()`, are performed in the `Syn_filt()` function as in the G.729, but they are simplified. The closed-loop fractional pitch search is performed in the `Pitch_fr3_fast()` function, which simplifies the process of finding the adaptive codebook by reducing the search range to a limited interval.
Fixed codebook search	The pulse positions search, performed by the `d4i40_17_fast()` function, is greatly accelerated. An iterative depth-first, tree search is used instead of the exhaustive nested-loop search as in G.729. This approach reduces complexity by limiting the number of candidate pulse position combinations.
Memory update	After the gain quantization stage, the states of the weighted synthesis filter are updated in order to perform target signal computation in the next subframe. This update, performed by the `encod_ld8a()` function, is simplified by eliminating a filtering operation.

1.2 Decoder Differences Between G.729A and G729

The decoder is almost the same as the one in G.729 except for the Post-Processing block, in which the long-term post-filter, short-term post-filter tilt, compensation filter and adaptive gain control are simplified. The post-processing block is implemented with four new functions: `post_filter()`, `pit_pst_filt()`, `preemphasis()` and `agc()`, but uses the `Syn_filt()` and `Residu()` functions as in G729. Because of these modifications there are some minor differences between the `decod_ld8a()` function in G.729A and the `decod_ld8k()` function in G.729. **Table 3** compares the functions used in the G.729A vocoder with those in the G.729 vocoder.

Table 3. G.729A Functions

Encoder Functions		Common Functions		Decoder Functions	
Function Name	**Comparison with G.729**	**Function Name**	**Comparison with G.729**	**Function Name**	**Comparison with G.729**
ACELP_Code_A	few diff.	ClearOverflow	same	agc	new
Autocorr	same	Copy	same	D_lsp	same
Az_lsp	few diff.	Gain_predict	same	Dec_gain	same
Chebps_10	same	Gain_update	same	Dec_lag3	same
Chebps_11	same	Get_lsp_pol	same	Decod_ACELP	same
Cor_h	same	GetOverflow	same	Decod_ld8a	few diff.
Cor_h_X	same	Int_qlpc	same	Gain_update_erasure	same
Corr_xy2	same	Inv_sqrt	same	Lsp_iqua_cs	same
D4i40_17_fast	different	Log2	same	Pit_pst_filt	new
Dot_product	new	Lsf_lsp2	same	Post_Filter	new
Enc_lag3	same	Lsp_Az	same	Post_Process	same
Encod_ld8a	different	Lsp_expand_1_2	same	Preemphasis	new
G_pitch	same	Lsp_get_quant	same	Random	same
g729a_encode	similar	Lsp_prev_compose	same	g729a_decode	similar
g729a_encode_initialize	similar	Lsp_prev_extract	same	g729a_decode_initialize	similar
Gbk_presel	same	Lsp_prev_update	same		
Get_wegt	same	Lsp_stability	same		
Lag_window	same	Pow2	same		
Levinson	same	Pred_lt_3	same		
Lsp_expand_1	same	Qua_lsp	same		
Lsp_expand_2	same	Residu	same		
Lsp_get_tdist	same	Set_zero	same		
Lsp_last_select	same	Syn_filt	same		
Lsp_lsf2	same	Weight_Az	same		
Lsp_pre_select	same				
Lsp_qua_cs	same				
Lsp_select_1	same				
Lsp_select_2	same				
Parity_Pitch	same				
Pitch_fr3_fast	new				
Pitch_ol_fast	new				
Pre_Process	same				
Qua_gain	same				
Qua_lsp	same				
Relspwed	same				
test_err	same				
update_exc_err	same				

1.3 Data Table Differences Between G.729A and G729

Table 4 lists the differences between the global data tables used in G.729A and G.729 reference codes. As the table shows, the total table data space in G.729A is 584 bytes smaller than in the G.729 reference code.

Table 4. Data Table Comparison

Data Table Name	Comparison	Comment
Word16 grid[]	Different elements and smaller dimensions in G.729A	Table reduced by 20 bytes
Word16 inter_3[]	Unused in G.729A	Saves 26 bytes
Word16 tab_hup_s[]	Unused in G.729A	Saves 56 bytes
Word16 tab_hup_l[]	Unused in G.729A	Saves 224 bytes
Word16 table[]	Unused in G.729A	Saves 130 bytes
Word16 slope[]	Unused in G.729A	Saves 128 bytes

1.4 Processing Load Estimation

Because only a few functions are different and many identical functions were optimized in the G.729 project, the identical functions can be used to obtain a faster version of G729A vocoder, as shown in **Table 5**. Adapted code means that the initial ITU C reference code was modified to support multiple channels and use the intrinsic functions and processor flags. In addition, the 32-bit operations are inlined. These modifications are all similar to those performed in the G.729 project and they are requested by the optimized functions.

Table 5. G.729A Quick Speed-Up

Code Version	Encoder Speed (MCPS)	Decoder Speed (MCPS)	Total Speed (MCPS)
Adapted code	8.99	2.11	11.1
Adapted code with functions optimized in G.729 (asm if available)	6.05	1.19	7.24

Although speed improvement is significant, overall performance is still unacceptable. Based on the experience gained in the G.729 project, analysis of the initial code led to the estimated speed improvements listed in **Table 6**.

Table 6. Speed Estimation

Encoder Speed	Decoder Speed	Total Speed
4.8 MCPS[1]	0.97 MCPS	5.77 MCPS
NOTES: 1.Millions of Cycles Per Second.		

Before the estimation, a profiling session was performed to select the most time-consuming functions, those that took up to 80 percent of the encoder and up to 80 percent of decoder time. The functions already optimized in G.729 were included in this group of functions. Speed improvements in the other functions were estimated taking the following factors into account:

- the type of the code: DSP code or control code
- factors which affect data access, including data alignment and addressing mode
- code flow in loops

2 Implementation Methodology

The implementation methodology used for the G.729A speech codec project was the same as in the G.729 speech codec for SC140, as described in [3]. Thus, the G.729A project provided an opportunity to confirm the G.729 methodology. This section presents the results obtained at each step as well as comparisons to G.729 in terms of the impact of that step on the code speed and other related issues. **Table 7** summarizes the main stages of this methodology. For a complete description and examples for each stage, refer to [3].

Table 7. Methodology Steps

Development Stage	Description
Porting to SC140	Data type definitions, introduction of intrinsic functions, StarCore adaptations.
Project-level optimizations	Inlining, data alignment, multichannel transformations.
Algorithm changes	Platform-independent and platform-dependent changes in algorithms.
Function-level C optimizations	C optimization techniques (multisample, loop unrolling, split summation), better use of intrinsic functions.
Function implementation in assembly	Implementation of selected functions in assembly for best optimization.

The goal of the G.729 project was to obtain the maximum speed without concern for memory consumption (code, tables and stack). In the G.729A project, speed was also the primary concern, but special attention was also paid to memory consumption. Therefore, the results show a reduction in memory consumption as well as an increase in speed. One of the requirements of the ITU G.729A recommendation is that all implementations preserve bit-exactness with the reference code. To verify bit-exactness, the ITU provides a set of test vectors for both the encoder and decoder. These tests are listed in **Table 8**. In addition to ITU test vectors, the code was also tested with a set of Freescale Semiconductor internal test vectors that sum up to 56000 frames.

Table 8. ITU G.729A Test Vectors

Encoder Inputs	Encoder Outputs and Decoder Inputs	Decoder Outputs
algthm.in	algthm.bit	algthm.pst
fixed.in	fixed.bit	fixed.pst
lsp.in	lsp.bit	lsp.pst
pitch.in	pitch.bit	pitch.pst
speech.in	speech.bit	speech.pst
tame.in	tame.bit	tame.pst
	erasure.bit	erasure.pst
	overflow.bit	overflow.pst
	parity.bit	parity.pst

The tests were performed on StarCore SC140 simulator as well as the following hardware platforms:

- SC140 Software Development Platform (SDP) equipped with an SC140 core
- MSC8101 Application Development System equipped with an MSC8101 DSP.

The software environment for developing the project on the SC140 platform was Metrowerks® CodeWarrior® IDE Release 1. All results (MCPS and bytes) presented here were measured with the latest available tools. As tools evolve, one can expect improvement in these figures. A PC-based compiler was used to check the modifications of the C code and generate test vectors in the unit-testing phase.

2.1 Porting ITU G.729A Reference Code to the SC140

The original ITU C reference code can be easily compiled and run on the SC140 platform. However, this code is very inefficient because it does not use many of the available compiler extensions, and all DSP fractional operations are simulated with integer add, integer multiply or shift operations. Thus, in order to obtain a true SC140 ported version, the reference code was modified as follows:

- Word16, Word32, Flag and other data types were redefined in `typedef.h` to comply with SC140 architecture (see [3] and [11]).
- The intrinsic functions defined in the compiler's `prototype.h` replaced the DSP emulation functions from `basic_op.c`.
- The Boolean overflow and carry flags used by emulated fractional operations were removed because the intrinsic functions rely on the corresponding processor flags. However, because the overflow flag must be tested in some functions, two assembly routines that give access to the processor overflow flag were added to `basic_op.c`: `GetOverflow()` and `ClearOverflow()`.

Table 9 summarizes the results of the porting phase.

Table 9. Performance Results After Porting to the SC140

Speed	ROM		RAM	
	Program	Tables	Channel Data	Stack
15.07 MCPS	33.42 KB	5340 bytes	3144 bytes	2592 bytes

2.2 Project-Level Optimizations

In this development stage, three types of changes were applied to the ported code:

- changes that modified the number of C files and the structure of the code
- changes that affected memory structures
- changes that affected certain function prototypes

In the first step, almost all C files (except for `basic_op.c` and `oper_32b.c`) were split into smaller files, each containing only one function. The files were named with the name of the contained function. This action helps in the optimization process, especially in the unit testing phase and assembly implementation phases, and enables the functions to be grouped in separate compilation classes— those optimized for speed and those optimized for size.

The second step was to alter the code to accommodate multi-channel systems. This step was performed in the same way as in the G.729 project—the global and the static variables from each C module (except for the constant tables from `tab_ld8a.c`) were moved in special data structures that form the so-called ‘channel data’. The memory for these structures is allocated on the caller stack and is now initialized with dedicated functions. In addition, pointers to the appropriate channel data structures were added to the function prototypes.

The library calls must be compliant with the Application Binary Interface (ABI) as specified in [3]. The ABI defines the standard calling convention and other rules for the calling and called functions. Code Listing 1 shows the ABI-compliant C functions of the external interface to the vocoder.

Code Listing 1. G.729A Vocoder Software Interface

```
void g729a_encode_initialize(G729A_ENCODER_CHANNEL_INFO_T *enc_info);
void g729a_encode(Word16 *signal, Word16 *prm,
                  G729A_ENCODER_CHANNEL_INFO_T *enc_info);
```

```
void g729a_decode_initialize(G729A_DECODER_CHANNEL_INFO_T *decoder_channel_info);
void g729a_decode(Word16 *prm, Word16 *synth,
                  G729A_DECODER_CHANNEL_INFO_T *dec_info);
```

Table 10 summarizes the data structures required by the software interface.

Table 10. Data Structures Required by Vocoder Software Interface

Data Structure	Description
G729A_ENCODER_CHANNEL_INFO_T	8-byte aligned type defined by MDCR
G729A_DECODER_CHANNEL_INFO_T	8-byte aligned type defined by MDCR
signal	8-byte aligned array of eighty 16-bit signed fractions
prm	8-byte aligned array of eleven 16-bit integers that stores the analysis parameters
synth	8-byte aligned array of eighty 16-bit signed fractions that stores the synthesized speech

Another change was the introduction of fast 32-bit operations. Because the ITU code was originally designed for 16-bit processors, the 32-bit operations are performed using a non-standard representation of 32-bit double-precision numbers called double precision format (DPF). Two 16-bit portions that were originally processed in two separate DALU registers were combined into a single 32-bit value, using only one DALU register. This enables the usage of the processor's 32-bit capabilities, speeding-up the 32-bit operations in `oper_32b.c`. The prototypes of several functions that originally received two 16-bit parameters were modified to receive only one 32-bit DPF parameter. In addition, the original DPF vectors stored as two 16-bit vectors were combined and stored as 32-bit vectors, thus increasing the efficiency of memory operations.

The StarCore C compiler provides efficient support for writing fast DPF operations based on intrinsics. Appendix A lists the new form of the DPF operations used in the G.729A project.

As in the G.729 project, the profiling performed on the ported G.729A code revealed that 32-bit operations required a great deal of processing time. Although the functions were quite small, the calling overhead became significant because of the frequency of the calls. **Table 11** shows the impact of three such 32-bit operations on the encoder time. Note that because the three functions take more than 1 MCPS, inlining 32-bit functions can provide a significant speed increase.

Table 11. Profile Data for Most Called DPF Operations

Function	Function Time (cycles)	Calls per Frame	Function Time × Calls per Frame (cycles)	Percentage of Encoder Time
Mpy_32_16()	8	816	6930	5.63%
L_extract()	8	538	4197	3.41%
Mpy_32	12	123	1511	1.23%

Unlike the G.729 project, the G.729A project was also concerned with reducing data memory consumption. The space occupied by the tables was reduced by 620 bytes by applying several techniques listed in **Table 12**. In some cases the elements were computed, but the size of the introduced code was smaller than the size of the elements eliminated. In addition, the overhead is not significant so the space gained justified the trade. In other cases the tables were regenerated on the function stack without increasing the maximum stack size of the vocoder.

Table 12. Tables Reduction

Reduction Method	Affected Table	Size Decrease (bytes)	Files/Functions Affected
Elimination of symmetrical parts	table2[]	62	lsf_lsp2 lsp_lsf2
	slope_cos[]	64	lsf_lsp2
	slope_acos[]	64	lsp_lsf2
	grid[]	50	az_lsp
Removal of tables that can be dynamically computed	tab_zone[]	306	test_err update_exc_err
Elimination of unused elements from a table	a100[]	2	post_process
	a140[]	2	pre_process()
Storage type redefinition	map1[]	8	qua_gain
	map2[]	16	
	imap1[]	8	dec_gain
	imap2[]	16	
Removal of the table needed by the vocoder caller.	bitsno[]	22	bits.c file created

One method employed to reduce stack size was to map data structures with different and disjointed lifetimes to the same storage area, allocated as soon as possible in the functions' calling chain. In addition, some redundant parameters from the function prototypes were removed. **Table 13** lists the speed and memory usage after project-level optimization.

Table 13. Performance Results After Project-Level Optimization

Speed	ROM		RAM	
	Program	Tables	Channel Data	Stack
13.82 MCPS	30.40 KB	4270 bytes	$3108 \times N^1$ bytes	2240 bytes
NOTES: 1.N = Number of channels				

2.3 Algorithmic Changes

As described in the introduction, if one were simply to C-optimize or hand-assemble only those portions of the code with the worst C performance, the estimated speed would be about 5.77 MCPS. This figure can be improved if the algorithms in certain functions or blocks of chained functions are changed as described in [3]. Even if these algorithmic changes do not improve speed directly, they allow the implementation of many subsequent C optimization techniques in the next phase, function level C optimizations, described in **Section 2.4**. A profiling session was run to identify the functions to be algorithmic changed. The 80-20 rule of thumb was applied in order to group the functions into two sets:

- G1 set – the most time-consuming functions, consuming 80 percent of the frame processing time
- G2 set – the functions that take the remaining 20 percent of the frame processing time

Most functions selected for the algorithmic changes belong to the top of the G1 set. The G1 set also includes closely-linked functions that provide input data to or use the results from the time consuming functions.

Table 14 lists the functions selected for algorithmic changes. All the functions are encoder functions because the encoder consumes about 80% of the vocoder time. These functions consume 56.9 percent of the encoder time.

Table 14. Functions Selected for Algorithmic Changes.

Functional Module	G1 Function	Encoder Time	Related Function	Encoder Time	Total Encoder Time
Fixed codebook search	D4i40_17_fast()	15.4%	Acelp_code_A()	0.23%	28.18%
	Cor_h_X()	8.93%			
	Cor_h()	3.62%			
LP analysis	Chebps_1x()	5.65%			16.71%
	Az_lsp()	4.38%			
	Autocorr()	3.88%	Lag_window()	0.13%	
	Levinson()	2.67%			
Open loop pitch analysis	Pitch_ol_fast()	6.33%			6.33%
Gain quantization	Qua_gain()	5.68%			5.68%

Two types of changes are applied in this stage:

1. Platform-independent changes – improvements in code flow:
   a. avoid repeated computations of the same value
   b. reorder computations to avoid repeated fetches of the same value
   c. reduce the number of tests
   d. replace time consuming operations like *div* or *log* with simpler operations
2. Platform dependent changes – improvements that take advantage of the parallel architecture of the processor:
   a. Reorder or restructure the vectors in order to perform sequential packed accesses
   b. Searches based on interval splitting
   c. Perform sequential non-dependent computations in parallel
   d. Adapt the computations to pipelines with four computation units.

Note: Algorithmic changes may not be necessary if the algorithm is designed from the start to take advantage of the SC140 architecture.

Table 15 summarizes the vocoder results at the end of this stage.

Table 15. Performance Results After Algorithmic Changes

Speed	ROM		RAM	
	Program	Tables	Channel Data	Stack
15.47 MCPS	32.90 KB	4740 bytes	$3108 \times N^1$ bytes	2480 bytes
NOTES: 1.N = Number of channels				

2.4 Function-Level C Optimizations

The C code was modified so that the compiler can generate code that takes full advantage of the SC140 parallel architecture. First, a new profiling session was run to identify the G1 set of functions after the algorithmic changes phase. Function-level C optimizations were then performed on all the G1 functions. The general optimization techniques included:

- Multisample
- Split summation
- Loop unrolling
- Loop merging
- Loop splitting.

The selection methodology and the optimization techniques are fully described in [3]. Special care was taken when applying the multisample technique to two nested loops. Typically, multisample by 4 is applied to maximize execution speed, but this sometimes results in an unacceptable increase in code size. In these cases, multisample by 2 combined with split summation can generate smaller code with similar speed improvement, and this was the technique applied to the `Lsp_pre_select()` and `Autocorr()` functions.

A similar problem occurs when searching in two vectors for the index of the elements that gives the maximum ratio. Ideally, the fastest solution is to perform the search with three partial maxima with the vectors already computed. However, in the real world the terms of the ratio must be computed first, as with the fixed codebook search (the `d4i40_17_fast()` function). In this case, computing the ratio terms and searching the index with two local maxima is more efficient than pre-computing the vectors and then performing the search because it saves both code size and stack size with a very small speed penalty. In addition to the programming tips described in [3], it is worth mentioning that the modulo addressing mode is now supported by the C compiler. This technique was applied to the `Cor_h()` function to save code size without a speed penalty.

Another way to reduce code size is to eliminate the code redundancies if the time penalty is negligible. For example, the only differences between the two Chebychev polynomial evaluation functions—`Chebps_10()` and `Chebps_11()`—are three constant values. The two functions were replaced with a new generic function, `Chebps()`, which receives the constants as parameters. **Table 16** summarizes the results after function-level C optimization.

Table 16. Performance Results After Function-Level C Optimization

Speed	ROM		RAM	
	Program	Tables	Channel Data	Stack
6.81 MCPS	32.80 KB	4736 bytes	2240 × N[1] bytes	2312 bytes
NOTES: 1.N = Number of channels.				

2.5 Assembly Implementation

While results for compiled C in the G.729A project were impressive, performance still fell short of estimates of capability of the SC140 architecture. The next step was to manually intervene and hand-assemble critical portions of the code that consumed a lot of cycles. Typical areas where developer intervention can increase performance over compiled code include

- Complex aliasing analysis without the *–Og* compilation option
- Register pressure in long loops

- Inability to force parallel operations in C
- Complex alignment-preservation analysis

For functions in which the generated code is almost optimal, the following time-saving approach can be used:

1. Compile the C-optimized code and retain the assembly output
2. Manually modify the generated assembly code
3. Replace the C file with the new assembly file

The set of functions already implemented in assembly in G.729 was reused with minor adjustments in some cases. **Table 17** lists the functions that required assembly implementation.

Table 17. Assembly-Implemented Functions in G.729A

Module	Function	Cycle Count per Call		Reused from G.729	Comments
		C-Optimized Version	Assembly Version		
common	Get_lsp_pol()	213	121	yes	
	Pred_lt_3()	424	392	yes	
	Residu()	203	164	yes	One minor change
	Syn_filt()	347	184	yes	Few modifications
encoder	Autocorr()	1308	921	no	
	Az_lsp()+Chebps()	8827	3861	no	
	Cor_h()	1732	1154	no	Compiler generated assembly with small optimizations
	D4i40_17_fast()	5458	2438	no	
	Levinson()	2023	1252	yes	Minor adaptations
	Pitch_fr3_fast()	2584	2512	no	Compiler generated assembly with minor optimizations
	Pitch_ol_fast()	2838	2197	no	
	Pre_process()	937	575	yes	
	Qua_gain()	2450	1234	no	
decoder	Post_process()	926	575	yes	

Table 18 summarizes the results after assembly implementation.

Table 18. Performance Results After Function Implementation in Assembly

Speed	ROM		RAM	
	Program	Tables	Channel Data	Stack
4.7 MCPS	28.09 KB	4736 bytes	$2240 \times N^{1}$ bytes	2312 bytes
NOTES: 1.N = Number of channels.				

3 Results

Table 19 summarizes the results of the G.729A vocoder project in terms of the processing load (measured in Million Cycles per Second or MCPS) and memory consumption.

Table 19. Vocoder Performance Results

Development Stage	Speed (MCPS)	ROM Memory Consumption		RAM Memory Consumption	
		Program (KB)	Tables (bytes)	Channel Data (bytes)	Stack (bytes)
SRS	5.5	31.00	4812	3108 × N[1]	2560
Porting to SC140	15.07	33.42	5340	3144	2592
Project_Level Optimizations	13.82	30.42	4720	3108 × N	2240
Algorithmic Changes	15.47	32.90	4740	3108 × N	2480
Function_Level C Optimization	6.81	32.80	4736	2240 × N	2312
Function Implementation in Assembly - 4 Functions	5.47	30.51	4736	2240 × N	2312
Function Implementation in Assembly - 15 Functions	4.7	28.09	4736	2240 × N	2312
NOTES: 1.N = the number of channels.					

The number of MCPS required to encode/decode a frame is obtained by multiplying the measured number of cycles by the number of frames to be processed per second (in G.729A, 100 frames of 10 ms each must be processed per second), and dividing the result by 1,000,000. For example, if it takes 50,000cycles to encode or decode a frame, the processing power required is (50,000 × 100) / 1,000,000 = 5 MCPS. The cycle count can be measured with either the `simsc100.exe` program in the StarCore simulator or the MSC8101 Application Development System (ADS). The simulator has a special ‘cyc’ register that counts the clock cycles elapsed from the beginning of the simulation. This register also counts the memory stalls caused by concurrent accesses to the same memory block. In the ADS, the cycle count is measured by configuring the EOnCE event counter to count the core clock cycles (for details see [17]). Unfortunately this counter does not count the memory stalls. The overall vocoder cycle count is the sum of the encoder and decoder worst case results, which were obtained after performing the measurements on all the ITU test vectors. For every test, the cycle counts measured in hardware were smaller than those measured with the simulator, so the simulator was used to report the results presented here.

Stack consumption is measured by tracing the movement of the stack pointer. The Perl script listed in Appendix B parses the log file generated by simsc100.exe and computes the largest stack frame used by the tested module—encoder or decoder. The chosen result is the larger of the two results. The script also reports the calling chain that generated maximum stack growth. It generates simulator log files by calling the appropriate simulator commands. The primary script routines include

- stack_analyzer_encoder.sc
- stack_analyzer_decoder.sc
- stack_analyzer_frame_start.sc
- stack_analyzer_frame_end.sc

Table 19 also shows the MDCR Software Requirements Specification (SRS) for G.729A. The SRS specifies the target performance in terms of speed (MCPS) and memory consumption (bytes) for the entire vocoder (encoder + decoder). The SRS was established by correlating several factors, including

- A speed estimation of 5.77 MCPS if the functions that take 80% of the vocoder time are only C-optimized
- Experience with memory consumption on the G.729 project
- Other Freescale Semiconductor internal documents
- Third party implementations of the same vocoder on similar processors.

If the final implementation is run on StarCore at 300 Mhz the total number of channels processed in parallel is

$$N = 300/4.7 = 63 \qquad \textbf{Equation 1}$$

The memory required to process 63 channels is 172.72 KB. **Figure 1** and **Figure 2** show the evolution of the data size and code size respectively for each development stage.

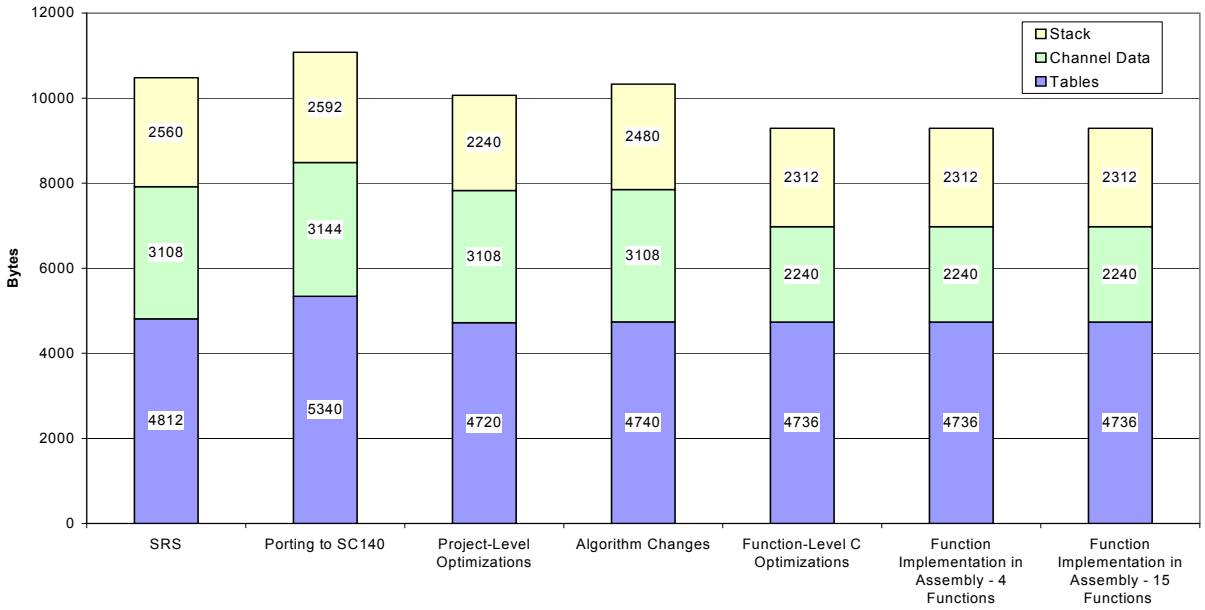

Figure 1. Data Size Versus Development Stage

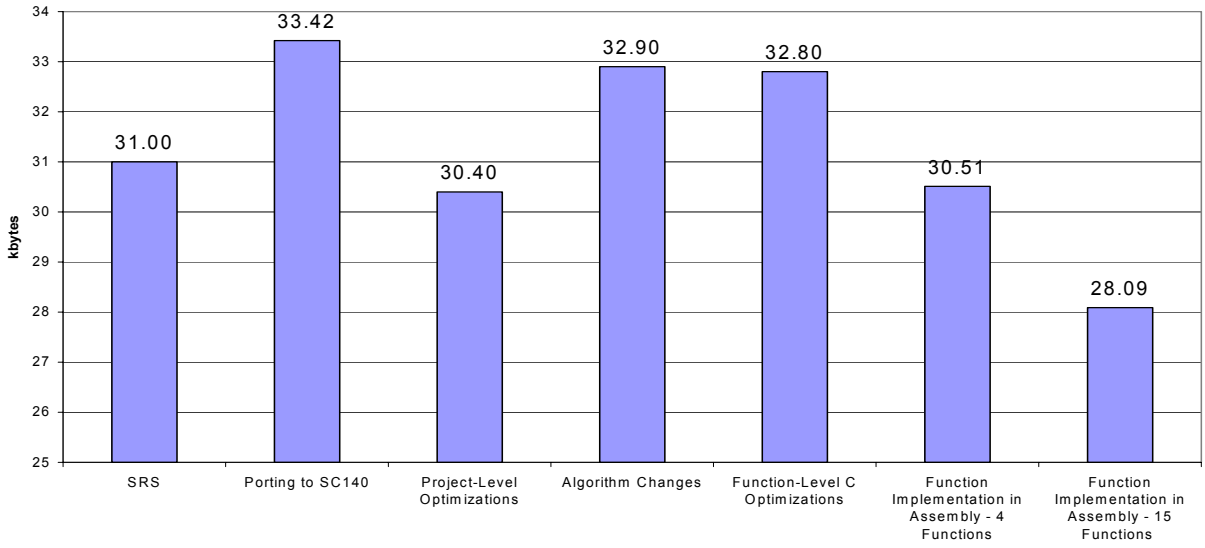

Figure 2. Program Size Versus Development Stage

Equation 2 links the performance of the group of functions selected for optimization (G1) to the overall application performance (see also [3]).

$$S = \frac{1}{1 - \frac{P\,(f-1)}{f}} = \frac{f}{P(1-f)+f} \qquad \textbf{Equation 2}$$

where S is the application performance improvement
P is the percentage of the application run-time taken by the G1 functions
f is the optimization factor, which represents the average speed-up factor when one portion of reference C code is C-optimized or written in assembly.

Given S, f can be computed as

$$f = \frac{SP}{1 + SP - S} \qquad \textbf{Equation 3}$$

In the G.729A project, $P = 92\%$ instead of the initially proposed 80% in order to exceed the performance described in the SRS. The already optimized functions reused from G.729 helped us to reach this goal. **Table 20** depicts the computed values for S and f for the function level C optimization phase and for function implementation in assembly. It can be seen that the values computed for the optimization factor f are consistent with the values obtained in G.729 project—2.44 for optimized C and 3.99 for assembly implementation.

Table 20. *S* and *f* Computation

Development Stage	Speed (MCPS)	*S*	*f*
Porting to SC140	15.07	—	—
Function-level C optimization	6.81	2.21	2.47
Function implementation in assembly	4.7	3.21	3.97

Figure 3 shows the evolution of speed versus the effort in man-months.

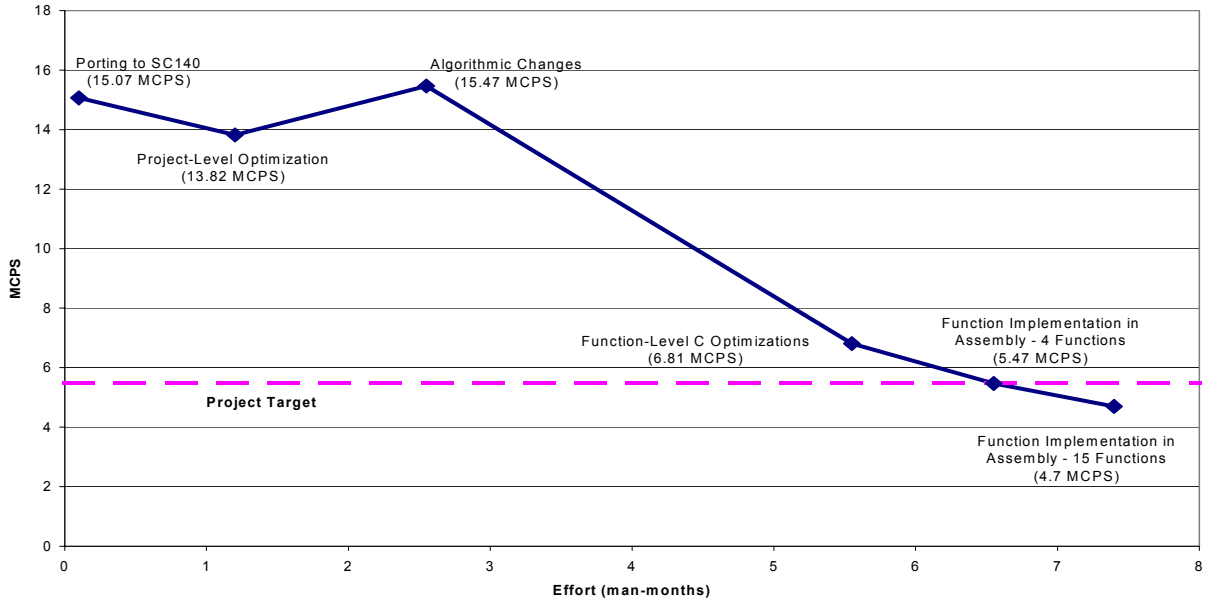

Figure 3. Processing Load Versus Effort

Note that the project target can be reached sooner by implementing fewer functions in assembly. **Table 21** shows the functions in descending order of the time gained per frame.

Table 21. Speed Gain for Assembly Implementations

Function	Cycle Count per Call		Gain per Call	Calls per Frame	Gain per Frame	Sum
	C-Optimized Version	Assembly Version				
D4i40_17_fast()	5458	2438	3020	2	6040	6040
Az_lsp() + Chebps()	8827	3861	4966	1	4966	11006
Qua_gain()	2450	1234	1216	2	2432	13438
Syn_filt()	347	184	163	14	2282	15720
Cor_h()	1732	1154	578	2	1156	16876
Levinson()	2023	1252	771	1	771	17647
Pitch_ol_fast()	2838	2197	641	1	641	18288
Autocorr()	1308	921	387	1	387	18675
Get_lsp_pol()	213	121	92	4	368	19043
Pre_process()	937	575	362	1	362	19405
Post_process()	926	575	351	1	351	19756
Pred_lt_3()	424	392	32	8	256	20012
Residu()	203	164	39	4	156	20168
Pitch_fr3_fast()	2584	2512	72	2	144	20312

Only the first four functions, D4i40_17_fast(), Az_lsp(), Chebps() and Qua_gain(), need be implemented in assembly to gain 1.34 MCPS, which is more than needed to make up the difference of 1.31 MCPS between the best C implementation and the project target. The four functions represent 17.32% of the optimized C code.

4 Conclusions

The G.729A implementation on the StarCore SC140/SC1400 cores demonstrates how the core architecture can substantially enhance performance using the optimization techniques and methodologies presented in this application note. A DSP-intensive C reference code can be sped up 2.4 times by optimizing the C code. If greater speed is required, assembly implementation can further improve performance by a factor of 4. Applying the methodology described here proved that speed improvement can gained in conjunction with reduced memory consumption. On the SC140, the G.729A vocoder speed improved from 15.07 MCPS (ported ITU reference), which allows only 19 simultaneous channels on a 300-Mhz DSP, to 4.7 MCPS, which allows 63 channels. Moreover, the memory consumption (code + data) for a single-channel implementation was reduced from 44.23 KB to 37.16 KB.

Use of the StarCore compiler shortened project development time. For example, the C optimization phase of the G.729A project took 0.35 man-months per MCPS on average, and the assembler implementation phase took 0.87 man-months per MCPS on average. Bit-exactness tests were passed on both the StarCore simulator and the MSC8101 ADS for all ITU test vectors, as well as an extended set of Freescale internal test vectors.

5 References

[1] *ITU-T Recommendation G.729 (03/96): Coding of Speech at 8kbit/s Using Conjugate-Structure Algebraic-Code-Excited Linear-Prediction (CS-ACELP)*.

[2] *ITU-T Recommendation G.729 - Annex A (11/96): Reduced Complexity 8 kbit/s CS-ACELP Speech Codec*.

[3] *ITU-T G.729 Implementation on StarCore™ SC140/SC1400 Cores*, Freescale Semiconductor application note, AN2094.

[4] *CCITT Blue Book*. The International Telegraph and Telephone Consultative Committee. CCITT, Geneva, 1989.

[5] *A Practical Handbook of Speech Coders*, R. G. Goldberg and L. Riek, CRC Press, 2000. ISBN 0-8493-8525-3.

[6] *Vocoder Intelligibility and Quality Test Methods*, J. Tardelli and E. Kreamer. IEEE Int. Conf. Acoust. Sp. Sig. Proc., 1996, pp.1145-1148.

[7] "The Implementation of G.729 Speech Coder on a 16bit DSP Chip for the CDMA IMT_2000 System," J. Kim et al. *IEEE Trans. on Consumer Electronics*, vol. 45, no. 2, May 1999, pp. 443-448.

[8] *A New Low Bit Rate Low Delay Algebraic CELP (ACELP) Coder*, R. El-Kouatly and S. H. El-Ramly. Seventeenth National Radio Science Conference, Feb. 22-24, 2000, Minufiya University, Egypt.

[9] *ITU-T Recommendation P.810* (02/96). Modulated Noise Reference Unit (MNRU).

[10] *SC140 DSP Core Reference Manual*, MNSC140CORE, Rev.1, 6/2000.

[11] *SC100 C Compiler User's Manual*, MNSC100CC, Rev.1.7, 8/2000.

[12] *SC100 Assembly Language Tools User's Manual*, MNSC100ALT/D, Rev.1.5, 8/2000.

[13] *SC100 Application Binary Interface Reference Manual*, MNSC100ABI/D, Rev.1.8, 04/2000.

[14] *StarCore Multisample Programming Technique*, STCR140MLAN document, Rev1, 09/1999.

[15] *Efficient Programming Techniques for SC140* (Freescale Semiconductor internal document).

[16] *GSM EFR Vocoder on StarCore 140*, Dror Halahmi, Sharon Ronen, Yariv Mishlovsky, Assaf Naor, Shlomo Malka, Amit Gur, Haim Rizi, ICSPAT 1999.

[17] *Using the SC140/SC1400 Enhanced On-Chip Emulator Stopwatch Timer*, Freescale Semiconductor application note, AN2090.

Appendix A
Optimized 32-bit Operations

Code Listing 2. New Form of the 32-bit operations Mpy_32() and Mpy_32_16()

```
/*===========================================================================
FUNCTION: Mpy_32()

DESCRIPTION:
             Multiply two 32 bit integers (DPF). The result is divided by 2**31
             L_32 = (hi1*hi2)<<1 + ( (hi1*lo2)>>15 + (lo1*hi2)>>15 )<<1

             This operation can also be viewed as the multiplication of two Q31 number and
the result is also in Q31.

ARGUMENTS PASSED:

             a      first  number - hi1:lo1
             b      second number - hi2:lo2

RETURN VALUE:
             32 bit long signed integer (Word32) whose value falls in the
             range : 0x8000 0000 <= L_32 <= 0x7fff fff0.

PRE-CONDITIONS:
             None

POST-CONDITIONS:
             None

IMPORTANT NOTES:
             None

===========================================================================*/

static Word32 Mpy_32(Word32 a, Word32 b)
{
#pragma inline
   Word32 L_a, L_b;

   L_a = L_mult_ls(a, extract_h(b));
   L_b = mpysu_shr16(extract_h(a), b);
   L_a &= -2;
   L_b &= -2;
   return L_add(L_a, L_b);
}
/*===========================================================================*/
/*===========================================================================

FUNCTION: Mpy_32_16()

DESCRIPTION:
             Multiply a 16 bit integer by a 32 bit (DPF). The result is divided
             by 2**15

             This operation can also be viewed as the multiplication of a Q31
             number by a Q15 number, the result is in Q31.

             L_32 = (hi1*lo2)<<1 + ((lo1*lo2)>>15)<<1

ARGUMENTS PASSED:

             a      first  number - h1:lo1
             b      second number - lo2
```

```
RETURN VALUE:
              32 bit long signed integer (Word32) whose value falls in the
              range : 0x8000 0000 <= L_32 <= 0x7fff fff0.

PRE-CONDITIONS:
             None

POST-CONDITIONS:
             None

IMPORTANT NOTES:
             None

=================================================================================*/

static Word32 Mpy_32_16(Word32 a, Word16 b)
{
#pragma inline
   Word32 L_32;

   L_32 = L_mult_ls(a, b);
   return L_32 & -2;
}

/*=================================================================================*/
```

Appendix B
Stack Consumption Measurement

Code Listing 3. stack_analyzer.pl - Perl Script that Measures the Stack Consumption

```
# Freescale Semiconductor DSP Center Romania

use warnings;
use strict;

# Receives as parameter a scalar that represents the name of the module being tested
# Returns a reference to a hash table containing the functions form the module
# received as parameter
# and with their addresses as keys
sub get_map_table
{
    my $module = shift; # Get the first parameter
    my %map_table;      # This is the map table that will be filled in this function
    my $fin;            # Map file descriptor
# Open map file
    open( $fin , "../bin/" . $module . ".map" ) ||
    die("open " . "../bin/" . $module . ".map" . ": $!\n");
# Process the whole map file
    while( <$fin> )
    {
# Match lines that contain a function name and its address
# Such lines appear as follows:
# 0x00013760                      _Syn_filt
# 0x00013850                      _Weight_Az
# 0x00013b60                      _ACELP_Code_A
        if( /^(0x[0-9a-fA-F]{8})\s+_(\w+)\s*$/ )
        {
# Hash key is the address of the function and data is the function name
            $map_table{ $1 } = $2;
        }
    }
# Close map file
    close( $fin );
# Return a reference to the map table built
    return \%map_table;
}

=head1
The script parses the log file generated by encoder_measure_stack.sc and
decoder_measure_stack.sc simulator scripts.

The output of this script are the maximum values for coder and decoder stack
size.

The script assumes a fixed directory structure. It receives only one parameter
which establishes whether or not the script also runs the simulation.

The log files are similar in structure, thus the same basic block is repeated
twice, once for the coder module and once for the decoder module.
=cut

# Get the name of analyzed module
my $module=<$ARGV[0]>;

# Run the module tester to extract build number and date
system( "runsc100 ../bin/" . $module . ".eld > tmp.txt" );

my $fin;
```

```
open( $fin , "tmp.txt") || die( "Cannot create temporary files!!!" );

open( my $report_file,">../../../reports/stack_analysis_" . $module . ".txt" ) || die(
"Cannot create report file!!!" );

print($report_file "\nStatistics made for $module :\n\n");

while( <$fin> )
{
    if(/.*build *(\d{4}).*/)
    {
       print $report_file "Build number : $1\n";
    }
    if(/.*time: *(.*)/)
    {
       print $report_file "$1\n\n";
    }
}

close( $fin );

unlink( "tmp.txt" );

# Run the module tester to log the stack evolution
system( "cd ../bin; simsc100 ../scripts/stack_analyzer_".$module.".sc" );

# Open the stack log file of the module analyzed
open( $fin , "../logs/stack_analyzer_".$module.".log" ) ||
                    die( "Can't open log file : $!\n" );

my $stack_top = 0x0;
my $stack_base = 0xffffffff;

# This array will store the function call stack (pc values and not function names)
my @functions;

# Parse the log file
# The stack pointer for the C code is esp. Typical display line for esp looks like:
#                       esp={00000164264}

# First, keep the first value of the stack pointer in the $stack_base
# The first occurence of the stack pointer should be at the entrance of
# g729a_encode()/g729a_decode()
while( <$fin> )
{
    if ( /\s*esp=\{0*([0-9a-fA-F]+)\}\s*/ )
    {
        $stack_base = $1;
        $_ = <$fin>;
        /^p:\$([0-9a-fA-F]{8})/;
        $functions[0] = "0x" . $1;
        last;
    }
}

my $max_stack_line;

# Find the maximum of the other values => $stack_top
while( <$fin> )
{
    if( /\s*esp=\{0*([0-9a-fA-F]+)\}\s*$/ )
    {
        if( $1 > $stack_top )
        {
            $stack_top = $1;
            $_ = <$fin>;
```

```
            /^p:\$([0-9a-fA-F]{8})/;
            $functions[0] = "0x" . $1;
# Store the next line in file where the maximum stack pointer appears
            $max_stack_line = tell( $fin );
        }
    }
}

# Output stack dimension
print( $report_file "Maximum stack size for ".$module." is ".( $stack_top - $stack_base )."
with the following call stack:\n" );

# Jump to the line that follows the line that contains the maximum stack pointer
seek( $fin , $max_stack_line , 0);

my $function= 1;

# Parse the log file to detect where the stack pointer decreases and store the value
# of program counter
# A function can appear more than one time in the array
while( <$fin> )
{
    if ( /\s*esp=\{0*([0-9a-fA-F]+)\}\s*/ )
    {
        if( $1 < $stack_top )
        {
            $stack_top = $1;
            $_ = <$fin>;
            /^p:\$([0-9a-fA-F]{8})/;
            $functions[ $function++ ] = "0x" . $1;
        }
    }
}

close( $fin );

my $function_name0="";
my $function_name1;
my $map_table = get_map_table( $module );

chop( $module );

# Iterate the function array to search the function name corresponding to each address
foreach $function ( @functions )
{
# For each address iterate the map table
    foreach my $function_address ( sort keys %$map_table )
    {
# If the current function has a lower address store its name.
# Otherwise quit the loop because the function name was identified and is stored in
# $function_name1
        if( hex( $function ) >= hex( $function_address ) )
        {
            $function_name1 = $$map_table{ $function_address };
        }
        else
        {
            last;
        }
    }
# If the name of the previous function differs from the current one display the
# current function name
    unless( $function_name0 eq $function_name1 )
    {
        print( $report_file $function_name1 . "\n" );
        $function_name0 = $function_name1;
```

```
    }
# If g729a_encode()/g729a_decode() is reached, quit
    if( $function_name1 eq "g729a_" . $module )
    {
        last;
    }
}

# Close the report file
close( $report_file );
```

Code Listing 4. Simulator Command File for Encoder Stack Measurement

```
; Clear environment
display off
break off
output off
input off

; Display esp in unsigned format
radix u esp
display on esp

load encoder.eld

; Increment cnt1 and execute stack_analyzer_frame_start.sc macro
; when entering g729a_encode()function
break #1 _g729a_encode i1
break #2 _g729a_encode x ../scripts/stack_analyzer_frame_start.sc

; Execute stack_analyzer_frame_end.sc macro after exiting g729a_encode()function
break #3 _frame_end x ../scripts/stack_analyzer_frame_end.sc

; Stop the simulation after 10 frames
break #4 cnt1==11

; Write the esp to log file when it is modified
break #5 w esp s

; Temporarily disable breakpoint 5. This breakpoint should be enabled only
; when executing g729a_encode()function. This is done by the two macros.
break #5 d

log s ../logs/stack_analyzer_encoder.log -o

go
quit
```

Code Listing 1. Simulator Command File for Decoder Stack Measurement

```
; Clear environment
display off
break off
output off
input off

; Display esp in unsigned format
radix u esp
display on esp

load decoder.eld

; Increment cnt1 and execute stack_analyzer_frame_start.sc macro
; when entering g729a_decode()function
break #1 _g729a_decode i1
break #2 _g729a_decode x ../scripts/stack_analyzer_frame_start.sc

; Execute stack_analyzer_frame_end.sc macro after exiting g729a_decode()function
break #3 _frame_end x ../scripts/stack_analyzer_frame_end.sc

; Stop the simulation after 10 frames
break #4 cnt1==11

; Write the esp to log file when it is modified
break #5 w esp s

; Temporarily disable breakpoint 5. This breakpoint should be enabled only
; when executing g729a_decode()function. This is done by the two macros.
break #5 d

log s ../logs/stack_analyzer_decoder.log -o

go
quit
```

Code Listing 5. stack_analyzer_frame_start.sc

```
; Enable breakpoint 5 when entering g729a_encode() or g729a_decode() function
break #5 e
go
```

Code Listing 6. stack_analyzer_frame_end.sc

```
; Disable breakpoint 5 when leaving g729a_encode() or g729a_decode() function
break #5 d
go
```

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations not listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GMBH
Technical Information Center
Schatzbogen 7
81829 München, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T. Hong Kong
+800 2666 8080

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Order No.: AN2151
Rev. 1
1/2005

freescale™
semiconductor