AN2280
Rev. 1, 11/2004

3GPP-AMR-NB With ETSI-EFR Implementation on the StarCore™ SC140/SC1400 Cores

By Razvan Ungureanu, Bogdan Costinescu, and Costel Ilas

Among the vocoders defined by the Third Generation Partnership Project (3GPP) for use in the 3G world is the narrowband Adaptive Multi-Rate Codec for speech (AMR-NB). This application note describes the steps in implementing AMR-NB on the Freescale StarCore™ SC140 DSP core. The implementation methodology used for this AMR-NB speech codec project is the same as that for two earlier speech codec projects, in which Freescale proved the viability of this methodology.[1] These earlier projects used the methodology to gain speed; however, the project described in this application note uses the methodology to reduce the memory footprint (code and data size) because the product is targeted for 3G subscriber products. In addition, the most time-consuming functions are optimized for speed, achieving an efficient size/speed trade-off.

In contrast to a speed optimized implementation in which the requirements can be met by optimizing only few of the most time-consuming functions, the size optimizations require modification of a large number of functions. The challenge is to identify the functions to optimize and to predict the size gain. Even small changes in functions can yield a significant size reduction. Although the size optimizations have a negative impact on the Million Cycles Per Second (MCPS)

1. The two projects are described in detail in two Freescale application notes: *ITU-T G.729 Implementation on StarCore SC140* (AN2094/D) [1] and *ITU-T G.729A Implementation on StarCore SC140* (AN2151/D) [2].

CONTENTS

measurements[2], this implementation demonstrates that an acceptable trade-off between speed versus size can be achieved if the most time-consuming functions are carefully optimized for speed. In this project, the AMR-NB with EFR code size was reduced by 25.45 KB while the speed was increased 3.27 times, compared to the ported to SC140 AMR with EFR code just compiled with the size optimization switch turned on.

1 AMR-NB With EFR Vocoder Basics

The 3GPP-AMR-NB vocoder is based on the algebraic code excited linear prediction (ACELP) model. The encoder input is a 13-bit PCM signal sampled at 8 KHz. This signal is processed on 20 ms frames divided into four subframes of 5 ms each, transmitting the adaptive and fixed codebook parameters after each subframe. Depending on the transmission channel bandwidth, the bitstream rate can be adapted to one of the rates listed in **Table 1**.

Table 1. AMR-NB Bit Rates

Mode	Bit rate (kbit/s)	Number of Bits Per Frame	Similarity To
MR122	12.20	244	GSM EFR
MR102	10.20	204	
MR795	7.95	159	
MR74	7.40	148	TDMA IS-641A
MR67	6.70	134	
MR59	5.90	118	
MR515	5.15	103	
MR475	4.75	95	

The encoder implements a mechanism called Discontinuous Transmission (DTX) for turning off the transmission during silence speech frames. The DTX mechanism uses a Voice Activity Detector (VAD) on the encoder side and a Comfort Noise Generator (CNG) on the decoder side. Reference C code provides a choice of two different VAD mechanisms, VAD1 and VAD2 (the Freescale VAD), along with the proper test vectors for both. Although the bit rate and the encoded parameters in the MR122 mode of the AMR-NB vocoder are the same as those for the ETSI EFR vocoder, the two vocoders implement different VAD/DTX mechanisms. Due to many additional implementation differences, the encoded parameters of MR122 AMR-NB and EFR are not bit exact. Also, the auxiliary information appended to the encoded bitstream is different. **Table 2** lists the AMR-NB EFR functions that differ from EFR functions. The EFR vocoder also uses three new tables:

- *SID_codeword_bit_idx[]* used by encoder
- *dhf_mask[]* used by decoder
- and *interp_factor[]* used by decoder

Table 2. Comparison of EFR Code and AMR-NB Code

Module	Function	Comparison	Comments
common	update_gcode0_CN update_lsf_history update_lsf_p_CN	EFR only	DTX module functions

2. The MCPS number indicates the processing power needed to encode and decode a frame of 20 ms of speech in real time.

Table 2. Comparison of EFR Code and AMR-NB Code (Continued)

Module	Function	Comparison	Comments
Encoder	acf_averaging vad_computation predictor_values vad_hangover tone_detection spectral_comparison threshold_adaptation energy_computation schur_recursion step_up compute_rav1 periodicity_update	EFR only	VAD module functions
	cn_encoding aver_gain_code_history aver_lsf_history tx_dtx update_gain_code_history_tx compute_cn_excitation_gain sid_codeword_encoding		DTX module functions
	pitch_fr6		
	cbsearch cl_ltp cod_amr gainquant subframePostProc subframePreProc	AMR functions that also contain blocks of code used in EFR	
	vad_decision q_gain_code	Completely different	Function from VAD -
	encoder pitch_ol Q_plsf	Many differences	- - Q_plsf_5 in AMR-NB
	gain_quant enc_lag6 g_code g_pitch lag_max levinson q_gain_pitch	Few differences	MR795_gain_quant in AMR-NB - - - - -
Decoder	gmed5	EFR only	
	rx_dtx interpolate_cn_lsf interpolate_cn_param update_gain_code_history_rx		DTX module functions
	d_gain_code d_gain_pitch D_plsf_5 Decoder_amr decoder	Many differences	
	agc	Few differences	

2 Implementation Phases

Table 3 presents the main phases of the methodology used to implement AMR-NB on the SC140 core. A full description and examples for each step are provided in [1]. **Table 3** includes an additional phase necessary for our application: EFR integration. In the EFR integration phase, some functions or pieces of code, tables, channel data, and constants from the EFR reference code are included in the AMR-NB vocoder to achieve bit-exact EFR functionality.

Table 3. Methodology Phases

Development Phase	Description
Porting AMR-NB to the SC140 core	Data type definitions, introduction of intrinsic functions, StarCore adaptations.
EFR integration	Integration of ETSI-EFR code within the AMR-NB code
Project-level optimizations	Inlining fast 32-bit DPF operations, data alignment
Algorithm changes	Platform-independent and platform-dependent changes in algorithms
Function-level C optimizations	C optimization techniques (multisample, loop unrolling, split summation), and better use of intrinsic functions
Function implementation in assembly	Implementation of selected functions in assembly for best optimization

As discussed in **Section 1**, there are differences between the bitstream formats of the AMR-NB encoder and decoder and ETSI EFR encoder and decoder. Therefore, in addition to the reference code, a *wrapper interface* feeds each of the two vocoders with the correct data stream. **Code Listing 1** presents the API for the AMR-NB with EFR implementation.

Example 1. AMR-NB with EFR API

```
void MDCR_AMR750_amrefr_enc
(
       INT16                 *input_buf, /* input speech data, 160 words */
       INT16                 *output_buf, /* output data, 245 words       */
       AMREFR_ENC_CTRL_PTR   *ctrl_ptr,
       AMREFR_ENC_STATUS_PTR *status_ptr

);

typedef struct {
       UINT32   temp_2: 24;   /* place holders                    */
       UINT32   mode:    4;   /* AMR rate place holder            */
       UINT32   temp_1:  1;   /* place holder                     */
       UINT32   dtx:     1;   /* 1 - Enable                       */
       UINT32   coder:   1;   /* 0 - AMR, 1 - EFR                 */
       UINT32   init:    1;   /* 1 - initialize                   */
       UINT8    *static_ptr;  /* static channel data, 3072 bytes */
       UINT8    *scratch_ptr; /* scratch space, 4096 bytes       */
} AMREFR_ENC_CTRL_PTR;

typedef struct {
       UINT8    used_mode;    /* Unused for EFR                   */
       UINT8    sp_flag;
       UINT8    vad_flag;
} AMREFR_ENC_STATUS_PTR;
```

```
void MDCR_AMR750_amrefr_dec
(
       INT16                 * inbuf_ptr,  /* input data, 245 words         */
       INT16                 * outbuf_ptr, /* output speech data, 160 words */
       AMREFR_DEC_CTRL_IN_T  * ctrl_inptr,
       AMREFR_DEC_STATUS_PTR * status_ptr
);

typedef struct {
       UINT32   temp_2: 20;   /* place holders                    */
       UINT32   mode:    4;   /* AMR rate place holder            */
       UINT32   temp_1:  2;   /* place holders                    */
       UINT32   sid:     2;   /* SID frame type for EFR           */
       UINT32   bfi:     1;   /* BFI for EFR                      */
       UINT32   taf:     1;   /* Signifies TAF alignment for EFR */
       UINT32   coder:   1;   /* 0 - AMR, 1 - EFR                 */
       UINT32   init:    1;   /* 1 - initialize                   */
       UINT8    *static_ptr;  /* static channel data, 1776 bytes */
       UINT8    *scratch_ptr; /* scratch space of 4096 bytes     */
} AMREFR_DEC_CTRL_PTR;

typedef struct {
       UINT8    temp
} AMREFR_DEC_STATUS_PTR;
```

Both the AMR-NB and ETSI-EFR standards require the implementation to be bit exact with the C reference code. To test the bit exactness of the code, the two standards bodies provide sets of test vectors for all operating modes of each vocoder. **Table 4** presents the tests performed to verify the bit exactness of the application.

Table 4. Bit-Exactness Tests

Vocoder Type	Operating Mode	Number of Frames
3GPP-AMR-NB	MR475	7938
	MR515	7986
	MR59	7986
	MR67	7986
	MR74	7986
	MR795	7988
	MR102	7988
	MR122	9913
	MRDTX	2927
ETSI-EFR	speech	7645
	DTX	4222

The tests are performed on the MSC8101 Application Development System (MSC8101ADS), but they can be performed on the simulator instead, yielding the same results. The software development tools are provided with Metrowerks® CodeWarrior® IDE, Release 1.1. The performance results will improve as the C compiler evolves. The algorithmic changes are tested on a PC using a PC-native development environment. Speed measurements, presented as MCPS, are collected using the MSC8101 on-chip timers. The stack peak is measured on the MSC8101ADS using the *watermark* technique described in [3]. The speed and stack measurements can also be performed by processing the `simsc100.exe` simulator log file. This technique is described in [1], [2], and [3]. All the performance measurements are *worst case* measurements; that is, they are the maximum values obtained after all tests are performed in all AMR-NB and EFR operating modes.

2.1 Porting AMR-NB Reference Code

After the data types, such as Word8, Word16, Word32, and Flag, are redefined to comply with the SC140 architecture, the "out-of-the-box" C reference code is compiled to obtain an SC140 binary file. However, significantly better performance in both speed and size is obtained by replacing the calls to the DSP emulation functions—prototyped in `basic_op.h` and defined in `basicop2.c`—with compiler intrinsics defined in `prototype.h`. Intrinsic usage eliminates the need for *Overflow* and *Carry* boolean flags because their functionality is assured by the corresponding processor flags. However the overflow flag must be tested in several functions, so three intrinsics are used to give access to processor Overflow flag. See `prototype.h`: Set_Overflow(), Get_Overflow(), and Test_Overflow().

In the methodology presented in [1] and [2], *file splitting* and *multichannel transformations* are part of the project-level optimizations phase. However, to reduce development time, they are performed in the first phase in parallel with the other actions. File splitting yields more than 250 files. Although file splitting is time-consuming, it is very useful in the optimization process. Although this is a subscriber implementation, multichannel transformations allow conference calls between multiple parties and also make the project "infrastructure ready" without any performance penalties. **Table 5** presents the performance results in two cases: all C sources compiled with size optimizations turned on (`–O3 –Os`) or speed optimizations turned on (`–O3`).

Table 5. Performance Results of the Porting Phase

Development Stage	Speed (MCPS)	Memory Consumption (ROM)		Memory Consumption (RAM)	
		Program (KB[1])	Tables (KB)	Channel Data (KB)	Stack (KB)
AMR-NB porting to SC140 size compilation	35.45	70.7	28.3	$4.6 \times N^2$	5.4
AMR-NB porting to SC140 speed compilation	25.50	91.1	28.3	$4.6 \times N$	5.2

NOTES: 1. 1 KB = 1024 bytes.
2. N = the number of channels.

2.2 EFR Integration

The overall steps performed to add EFR functionality to the ported AMR-NB are listed as follows:

1. Introduce the amr_efr_selector flag into the data channel to indicate the current mode of operation: AMR-NB or EFR functionality.
2. Add the EFR tables, code, and channel data to the AMR-NB project.
3. Adapt the EFR code to use the AMR-NB name space when accessing constants or functions with the same functionality but different naming.

In a few cases, parts of the EFR code must be ported to the SC140 architecture, as described in **Section 2.1**. For each new function that deals with channel data of its own, a new data structure is defined in the `amr_status.h` file. For functions that are similar in the two vocoders but have different channel data, the different channel data is added to the existing data structures from `amr_status.h`.

In some cases, the code from several different EFR functions resides in one AMR-NB function. These EFR functions are dropped in favor of using the AMR-NB function for EFR mode of operation. Many sections of code from EFR functions are added to AMR-NB functions with similar functionality. In this case, the `amr_efr_selector` flag is tested before entering the code block that is not common to the two vocoders (see **Example 2**):

Example 2. Testing of the `amr_efr_selector` Flag

```
#ifdef  MDCR_AMR_WITH_EFR
              if( amr_efr_selector ==  MDCR_EFR_VOCODER)
              {
               .. EFR code here ..
              }
              else
              {
#endif /* MDCR_AMR_WITH_EFR* /

               .. AMR-NB code ..
#ifdef  MDCR_AMR_WITH_EFR
              }
#endif /* MDCR_AMR_WITH_EFR* /
```

All non-AMR-NB code introduced is compiled and conditioned by the MDCR_AMR_WITH_EFR preprocessor *define* to allow easy removal of EFR code and obtain an AMR-NB-only implementation. **Table 6** lists the performance figures at the end of the EFR integration phase.

Table 6. Performance Results of the EFR Integration Phase

Development Stage	Speed (MCPS)	Memory Consumption (ROM)		Memory Consumption (RAM)	
		Program (KB)	Tables (KB)	Channel Data (KB)	Stack (KB)
EFR integration, size compilation	36.15	80.4	28.6	4.7 × N[1]	5.6
EFR integration, speed compilation	25.80	103.2	28.6	4.7 × N	5.3

NOTES: 1. N = the number of channels.

2.3 Project-Level Optimizations

The main code transformations involved in the project-level optimizations are as follows:

1. Change the memory representation of Double-Precision Format (DPF) numbers. Combine and store the two 16-bit portions of a DPF number as a single 32-bit number.
2. Inline the fast 32-bit operations on DPF numbers.

The fast versions of the DPF operations, defined in the `oper_32b.c` file, use the processor support for 32-bit operations. These functions are very short, and often the computations can be performed in parallel with other computations. Inlining these functions yields a significant speed gain because of the high call frequency. Also, eliminating the calling overhead reduces the code size.

3. Reduce table size, as follows:
   - Eliminate the symmetrical parts.
   - Remove the tables or parts of tables that can be dynamically computed.
   - Eliminate the unused elements from a table.

— Redefine storage types.

— Remove tables needed by the vocoder tester.

4. Reduce the stack by mapping the data structures with different and disjoint lifetime to the same storage area, allocated as soon as possible in the function calling chain. Reducing the number of function parameters also improves code size.

5. Allocate static memory for the channel data.

Almost 6 KB of code are saved by removing the malloc() calls and combining all channel data initialization and reset functions. Also, we gain size by replacing the individual data initialization with 0 with a global initialization to 0 of all the channel data structures.

The first four techniques are described in detail in [1] and [2]. The last task in this development phase is to perform a profiling session on the compiled for speed C sources in order to identify the two following function categories according to the 80–20 percent rule of thumb:

- *G1 set*. The top time-consuming functions whose processing time requires up to 80 percent of the frame processing time (see **Table 7**).
- *G2 set*. The functions that consume the remaining 20 percent of the frame processing time.

Table 7. Top Time-Consuming Functions

Common (bytes)	Encoder (bytes)	Decoder (bytes)
build_cn_code (180) filter_excitation (110) inv_sqrt (94) pred_lt_3or6 (164) pseudonoise (110) syn_filt (502) get_lsp_pol (304) residu (372) pre_post_process (340) restorevector (116) window4x (58) energy4x (68) correlation4x (90) scaleright4x (48)	az_lsp (654) block_norm (204) chebps (130) convolve (96) cor_h (488) lag_max_wgh (364) mr475_gain_quant (1120) mr795_gain_code_quant3 (914) mr795_gain_code_quant_mod (966) qua_gain (610) search_10and8i40 (3652) search_2i40_11bits (582) search_2i40_9bits (522) search_3i40_14bits (974) search_4i40_17bits (1300) vq_subvec (862) vq_subvec3 (306) vq_subvec_s (862) c_fft (492) comp_corr (210) cor_h_x2 (824) levinson (1256)	agc (330) preemphasis (88)

From now on, the G1 set of functions is compiled with speed optimizations turned on (`–O3`), and the G2 set of functions is compiled with the size optimizations options turned on (`–O3 –Os`). **Table 8** depicts the speed and the memory usage at the end of project-level optimizations.

Table 8. Performance Results of the Project-Level Optimizations Phase

Development Stage	Speed (MCPS)	Memory Consumption (ROM)		Memory Consumption (RAM)	
		Program (KB)	Tables (KB)	Channel Data (KB)	Stack (KB)
Project_Level Optimizations	18.14	65.7	24.3	5.6 × N[1]	6.4
NOTES: 1. N = the number of channels.					

2.4 Algorithmic Changes

Table 7 shows that several functions in the G1 set consume much memory when compiled for speed. Therefore, the purpose of the algorithmic changes phase is to reduce the size of some of these functions while keeping them in the list of speed-optimized functions. The only candidate for the algorithmic change is the fixed codebook search block of functions. Reordering some vectors provides efficient data access and also simplifies several computations. **Table 9** presents all the functions involved in the algorithmic change and the size reduction obtained.

Table 9. Functions in Algorithmic Changes Phase

Function	Code Size Reduction (Bytes)	Comments
build_code_10i40_35bits build_code_2i40_11bits build_code_2i40_9bits build_code_3i40_14bits build_code_4i40_17bits build_code_8i40_31bits	246 140 86 116 94 188	G1 set
code_2i40_11bits code_2i40_9bits code_3i40_14bits code_4i40_17bits	–104 216 190 190	G1 set
cor_h	–130	G1 set
set_sign	–94	G1 set
set_sign12k2.c	–48	G1 set
search_10and8i40 search_2i40_11bits search_2i40_9bits search_3i40_14bits search_4i40_17bits	86 116 70 48 146	
TOTAL	1556	

Table 10 depicts the speed and the memory usage at the end of the algorithmic change phase.

Table 10. Performance Results of the Algorithmic Changes Phase

Development Stage	Speed (MCPS)	Memory Consumption (ROM)		Memory Consumption (RAM)	
		Program (KB)	Tables (KB)	Channel Data (KB)	Stack (KB)
Algorithmic changes	18.48	64.2	24.3	5.6 × N[1]	6.4
NOTES: 1. N = the number of channels.					

2.5 Function-Level C Optimizations

Because our focus is to reduce code size, the most-used C optimization techniques are as follows:

- code factorization
- loop merging
- removal of data redundancies
- test number reduction
- avoiding repeated fetches
- avoiding repeated computation of the same value

These techniques are extensively applied to the G2 set of functions and sometimes applied to the G1 set of functions because they may reduce the register pressure, resulting in more efficient compiled code. Unfortunately, an accurate prediciton for size reduction in a C code optimization is impossible. In contrast to speed optimizations, size optimizations have no rule to specify the results of optimizing a few functions that require only 20 percent of the code size. Therefore, we must inspect many functions to see if some optimization techniques can be applied. We hand-optimized more than 50 functions to obtain the required code size. The first candidates for the code factorization technique are the fixed codebook search block of functions. Although there are five different search functions, they have a very similar structure based on two basic search blocks of code that are repeated with different variables and indexes. Therefore, we isolated the two search blocks into two small functions, search_calc1_index() and search_calc2_index(), fully optimized for speed to compensate for the calling overhead caused by the high calling frequency.

An important reduction in code size is obtained by performing the ABI-compliant register save/restore with function calls. This code can take up to 26 bytes if the d6, d7, r6, r7 registers must be saved. Calls to subroutines that perform this task require only 8 bytes, so the maximum size reduction can be $S = 18 \times n$ (bytes), where *n* is the number of functions that use this custom context saving. In this project, n=86 functions and the size reduction is 876 bytes. This technique is made possible by defining a custom calling convention.

Another important reduction in code size (812 bytes) is obtained by using the integer operations in several functions instead of the similar fractional intrinsics (that is, “+“ instead of add()). This is possible because the reference code developers used the *basic_op* calls on integer values for profiling purposes. However, care must be taken because some integer operations must be saturating operations. This can be achieved only with *intrinsic/basic_op* calls.

One concern is to reduce the size reserved for the stack segment. The memory space reserved on the stack for large local vectors is moved into a “scratch memory” space. To obtain the size of this scratch memory, we compare the sizes reserved by all disjoint call chains and choose the maximum one. The effort spent on this action is inversely proportional with the minimum size of the vectors that are moved. In this project, the minimum size for the eligible vectors is 100 bytes.

The following techniques are used to increase the speed of the functions placed in the G1 set:

- multisampling
- split summation
- loop unrolling
- loop splitting
- software pipelining

These techniques, which take advantage of the SC140 parallel architecture, are detailed in [1], [2], [8] and [9]. However, because of size constraints, care must be taken when implementing them. In some cases, software pipelining is not implemented, and also the split or unroll factor when applying these techniques is 2 instead of 4. One advantage is that some functions are similar or even identical to the ones used in ITU G.729 series (see **Table 11**), so, we can reuse or adapt the versions optimized for speed from G.729A projects to the AMR-NB project.

Table 11. Functions Adapted from ITU G.729 Series

Function	Comparison	Comments
syn_filt	same G.729AB	
get_lsp_pol	same G.729AB	
residu	same G.729AB	
convolve	same G.729	
az_lsp	similar G.729	
chebps	similar G.729	
levinson	similar G.729AB	EFR DPF operations different from AMR-NB or G729AB

Table 12 depicts the vocoder results of the end of this stage.

Table 12. Performance Results of the Function-Level C Optimizations Phase

Development Stage	Speed (MCPS)	Memory consumption ROM		Memory Consumption RAM	
		Program (KB)	Tables (KB)	Channel Data (KB)	Stack and Scratch (KB)
Function-Level C Optimizations	16.55	57.4	24.3	$4.8 \times N^{1}$	2.2 + 4
NOTES: 1. N = the number of channels.					

2.6 Assembly Implementation

The next step is to hand assemble the critical portions of the code that are high cycle consumers, as when the increased register pressure makes the compiler generate too much spill code. **Table 13** presents the assembly functions included in the project in this phase. The functions are listed in decreasing order of importance: most time consuming first in the speed row, and largest first in the size row.

Table 13. Hand Assembly Functions

Assembly Implementation Focus	Function
Speed	search_calc1_index search_calc2_index cor_h comp_corr restorevector az_lsp syn_filt pred_lt_3or6 prm2bits chebps convolve log2_norm c_fft vq_subvec_s residu mdcr_correlation4x set_sign12k2 set_sign mdcr_energy4x levinson inv_sqrt cor_h_x2 pre_post_process get_lsp_pol mdcr_scaleright4x mdcr_window4x
Size	decompress_code compress_code decompress10 sqrt_l_exp log2_norm compress10 copy

The functions in the “size assembly implementation focus” row are functions in the G2 set of small to medium complexity. The purpose is to see the size gain of hand-assembling such functions. The “size assembly implementation” strategy is to eliminate the spill code by performing a better register allocation and take advantage of the entire SC140 instruction set. We use the instructions that can perform the computations for which the compiler generates a larger number of simpler instructions. The conclusion is that too much effort is spent for a 10 –20 percent size gain. Again, for the functions listed in **Table 11** the assembly implementations are adapted from the G.729 project series. The fixed codebook search kernel from search_calc2_index() is also adapted from a G729AB codebook search. **Table 14** lists the performance figures at the end of this phase.

Table 14. Performance Results of the Function Implementation in Assembly Phase

Development Stage	Speed (MCPS)	Memory Consumption (ROM)		Memory Consumption (RAM)	
		Program (KB)	Tables (KB)	Channel Data (KB)	Stack and Scratch (KB)
Function Implementation in Assembly	11.05	54.95	24.3	4.8 × N[1]	2.2 + 4
NOTES: 1. N = the number of channels.					

3 Results

Table 15 summarizes the performance figures in terms of processing load (MCPS) and memory consumption. The final performance figures were achieved after an effort of eleven man months.

Table 15. Performance Results

Development Stage	Speed (MCPS)	Memory Consumption (ROM)		Memory Consumption (RAM)	
		Program (KB)	Tables (KB)	Channel Data (KB)	Stack and Scratch (KB)
AMR-NB Porting to SC140: Size compilation	35.45	70.70	28.3	4.6 × N[1]	5.4 + 0
AMR-NB Porting to SC140: Speed compilation	25.50	91.10	28.3	4.6 × N	5.2 + 0
EFR integration - size compilation	36.15	80.40	28.6	4.7 × N	5.6 + 0
EFR integration - speed compilation	25.80	103.20	28.6	4.7 × N	5.3 + 0
Project_Level Optimizations	18.14	65.70	24.3	5.6 × N	6.4 + 0
Algorithmic Changes	18.48	64.20	24.3	5.6 × N	2.4 + 4
Function_Level C Optimizations	16.55	57.40	24.3	4.8 × N	2.2 + 4
Function Implementation in Assembly	11.05	54.95	24.3	4.8 × N	2.2 + 4
AMR-NB only: Function Implementation in Assembly	10.90	46.40	24.1	4.5 × N	2.0 + 4
NOTES: 1. N = the number of channels.					

Table 16 presents the total number of channels processed in parallel with this subscriber implementation, on the SC140 architecture at 300 MHz.

Table 16. Number of Channels, Subscriber Implementation

Development Stage	Number of Channels	Memory Usage (KB)
AMR-NB Porting to SC140: Size compilation	8	141.2
AMR-NB Porting to SC140: Speed compilation	11	175.2
EFR integration - Size compilation	8	152.2
EFR integration - Speed compilation	11	188.80
Project_Level Optimizations	16	186.00
Algorithmic Changes	16	184.50
Function_Level C Optimizations	18	174.30

Table 16. Number of Channels, Subscriber Implementation (Continued)

Development Stage	Number of Channels	Memory Usage (KB)
Function Implementation in Assembly	27	215.05
AMR-NB only: Function Implementation in Assembly	27	198.00

The number of MCPSs 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 3GPP-AMR-NB and ETSI-EFR, 50 frames of 20 ms each must be processed in one second), and dividing the result by 1,000,000. For example, if 100,000 cycles are required to encode or decode a frame, the processing power required is (100,000 × 50) / 1,000,000 = 5 MCPS. The performance measurement techniques are detailed in [1], [2], [3] and [10]. The overall vocoder cycle count is the sum of the encoder and decoder worst case results, obtained after performing the measurements on all the test vectors with all modes of operation, EFR and AMR-NB, at every rate. The maximum stack frame is in fact the maximum between the encoder and the decoder maximum stack frame.

4 References

All of the following documents, except [9] are Freescale documents that are available at the web site listed on the back cover of this application note.

[1] *ITU-T G.729 Implementation on StarCore SC140 (AN2094)*.

[2] ITU-T G.729A Implementation on StarCore SC140 (AN2151).

[3] *Stack Measurement for StarCore SC140 Core* (AN2267).

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

[5] *SC100 C Compiler User's Manual* (MNSC100CC)..

[6] *SC100 Assembly Language Tools User's Manual (*MNSC100ALT/D).

[7] *SC100 Application Binary Interface Reference Manual* (MNSC100ABI/D).

[8] *StarCore Multisample Programming Technique* (STCR140MLAN/D).

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

[10] *Using the SC140 Enhanced OnCE Stopwatch Timer* (AN2090).

NOTES:

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

AN2280
Rev. 1
11/2004

freescale™
semiconductor