Freescale Semiconductor
Application Note

AN2274
Rev. 2, 9/2005

Generic Tone Generation on the StarCore™ SC140/SC1400 Cores

By Marc Cougoule, Lúcio F. C. Pessoa, David Melles, Valentin Emiya

The use of DSPs in computationally intensive applications, such as media gateways, voice over packet networks (VoIP, VoFR, VoATM), and xDSL modems, has resulted in efficient designs and systems with increasing computational power. Most communication systems require precise tone generation and detection for functions such as dual tone multiple frequency (DTMF) signaling, call progress tones, and FAX requesting tones. The ITU-T Recommendation E.180/Q.35 [1] specifies the technical characteristics of tones used in most telephone services available today. Most off-the-shelf components do not meet the cost, size, and power consumption requirements of many systems. However, all the required signaling tones rely on the same basic principle, which can be efficiently implemented in today's DSPs. The use of DSPs enables many unrelated tone protocols to be synthesized with the same firmware and used across a wide range of tone frequencies.

This application note presents the design, implementation, and testing of a generic tone generator on the Freescale MSC8101 processor. The first member of the Freescale MSC8100 family of DSPs, the MSC8101 processor [2] is based on the StarCore™ SC140 four-ALU DSP core. This device has 512 KB of memory and a Communications Processor Module (CPM), which make it a versatile device for communication applications. The proposed module has two major components: initialization (`fsl_tone_gen_init`) and generation (`fsl_tone_gen`). The first component relies on polynomial approximation of a cosine for calculating initial states and coefficients to be used in the second component. The core of the

CONTENTS

1	Digital Sine Wave Generation	2
1.1	Digital Oscillator	2
1.2	Polynomial Approximation	3
2	Tone Generator Implementation	4
2.1	Elementary Components	4
2.2	Tone Cycles	6
2.3	Functional Interface	6
2.4	Implementation Constraints	8
2.5	Status Reporting	9
2.6	Tone Generator Code Structure	9
3	Using and Testing the Generic Tone Generator	11
3.1	Test Set-up	11
3.2	Initialization Examples of Multi-Frequency Tones	11
3.3	Tests	15
3.4	Examples of Generated Signaling Tones	19
3.5	Performance Measurements	20
4	References	20

© Freescale Semiconductor, Inc., 2002, 2005. All rights reserved.

freescale™ semiconductor

tone generator (second component) relies on a digital oscillator. We tested this approach with a large number of signaling tones (such as DTMF, MF-R1, MF-R2), and it showed a good compromise in terms of performance, memory requirements, and precision. The module requires less than 0.1 MCPS on average, which can be improved with further hand-optimization of the assembly code. We also verified compliance with related ITU-T standards in terms of frequency accuracy, using both FFT magnitude plots of actual data generated from the module and real-time measurements with a spectrum analyzer.

1 Digital Sine Wave Generation

The most common ways to synthesize sine waves are:

- *Table look-up*. Offers the lowest complexity but is not used in the proposed tone generator because it cannot guarantee high precision with reasonable table sizes.
- *Polynomial approximation*. Offers the same high precision for every frequency, but requires the highest computational complexity.
- *Digital oscillator*. Uses intrinsic properties of trigonometric functions, which can be implemented with low memory and computational complexity, and usually exceeds precision requirements.

The proposed tone generator employs both the digital oscillator and polynomial approximation methods.

1.1 Digital Oscillator

The core of the tone generation module is a digital oscillator [3], that is, an IIR filter whose poles are on the unit circle. Our implementation uses a second-order filter, from which single-frequency tones are generated. The phase (θ) of the pole is related to the frequency of the synthesized sine wave:

$$\theta = 2\pi f / f_s$$

The equation for the oscillator is given as follows:

Equation 1

$$x(n) = 2.\cos\left(\frac{2\pi f}{f_s}\right) \cdot x(n-1) - x(n-2)$$

where $x(n)$ is a sine wave signal, f is the required frequency, and fs is the sampling frequency.

The amplitude and phase of the sine wave are determined by the initial states x (–1) and x (–2). Depending on initial conditions, either a cosine or a sine signal is generated by a digital oscillator. This algorithm has two main advantages:

- Low data and program memory requirements
- Low computational complexity

A digital oscillator requires the following initializations:

- The coefficient

$$2.\cos(2\pi f / f_s)$$

- The two initial conditions: x(–1) and x(–2)

The following table shows the computation of initial conditions:

COSINE Initialization	SINE Initialization
$x(-2) = \pm amplitude$ $x(-1) = \pm amplitude \times \frac{coefficient}{2} = amplitude \times \cos\left(\frac{2\pi f}{f_s}\right)$	$x(-2) = \pm amplitude \times \sin\frac{2\pi f}{f_s} = amplitude \times \cos\left(\frac{2\pi f}{f_s} - \frac{\pi}{2}\right)$ $x(-1) = 0$

For a pre-defined number of target frequencies, the initialization values can be pre-computed. However, a generic tone generation requires the computation with high precision for any frequency. A polynomial approximation, described in the following section, is employed for this calculation.

1.2 Polynomial Approximation

A polynomial approximation is used to compute the cosine necessary to initialize the digital oscillator. This method uses the Taylor's expansion of a cosine:

Equation 2

$$\cos(x) = \sum_{k=0}^{\infty} a_k x^{2k}, \; a_k = \frac{(-1)^k}{(2k)!}$$

Replacing x with the following:

$$2\pi f/f_s$$

and defining the normalized frequency as follows yields **Equation 3**:

$$f_n = f/f_s$$

Equation 3

$$\cos(2\pi f_n) = \sum_{k=0}^{\infty} b_k f_n^{2k}, \; b_k = (2\pi)^{2k} a_k$$

The actual implementation uses a factorized version of this formula with a finite number of elements and normalized coefficients. A sixth-order polynomial provides sufficient precision for 16-bit coefficients; furthermore, the same polynomial can be used to compute either sine or cosine, as follows:

$$\sin(x) = \cos\left(x - \frac{\pi}{2}\right)$$

Thus, the final polynomial approximation is as shown in **Equation 4**:

Equation 4

$$\cos(2\pi f_n) = 2^m(((c_3 f_n^2 + c_2)f_n^2 + c_1)f_n^2 + c_0),\ c_k = 2^{-m}b_k$$

The corresponding normalized coefficients to be stored in memory are scaled by 2^{-7} (that is, m = 7) and listed in **Table 1**.

Table 1. Normalized Coefficients

Coefficients	Fractional Value	16-bit Fixed Point
C_0	0.0078125	256 (0x0100)
C_1	–0.1542053	60483 (0xEC43)
C_2	0.5073242	16624 (0x40F0)
C_3	-0.6676025	43660 (0xAA8C)

2 Tone Generator Implementation

The proposed tone generator synthesizes tones composed of up to four simultaneous frequencies with arbitrary duration and frequency values. The method of defining tones is easy and flexible. The output signal is decomposed into elementary components. **Figure 1** shows an example of a possible desired tone.

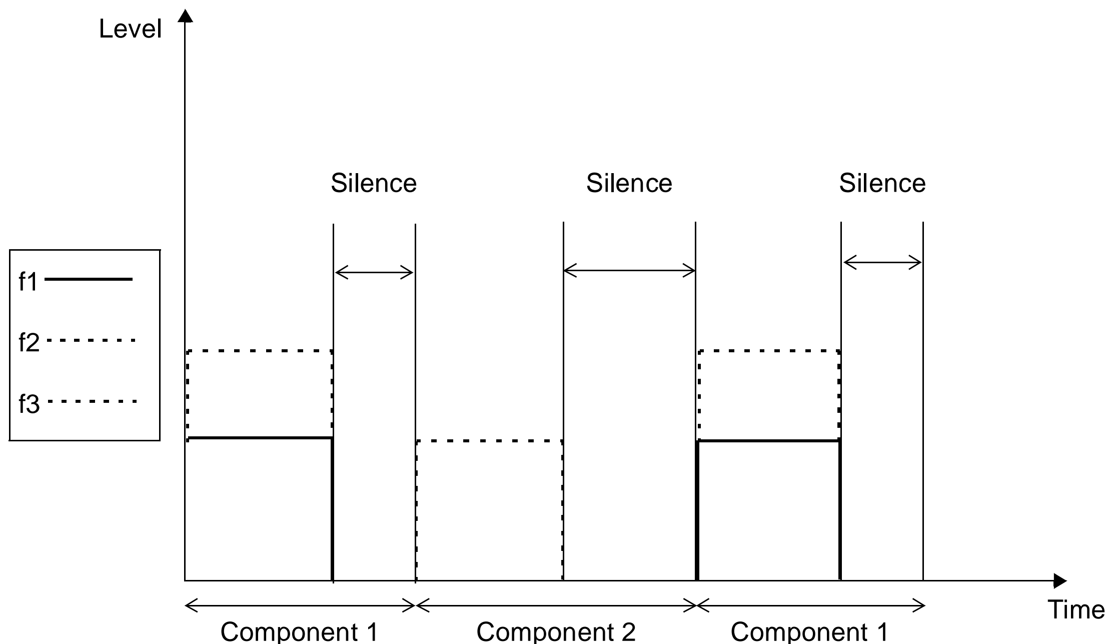

Figure 1. Example of a Desired Tone

2.1 Elementary Components

The elementary tone component is a pair composed of a sound period followed by a silence period (see **Figure 2**). This pair (sound + silence) can be repeated as often as needed to yield a component. In the example shown in **Figure 1**, each component has only one consecutive occurrence of its base pair.

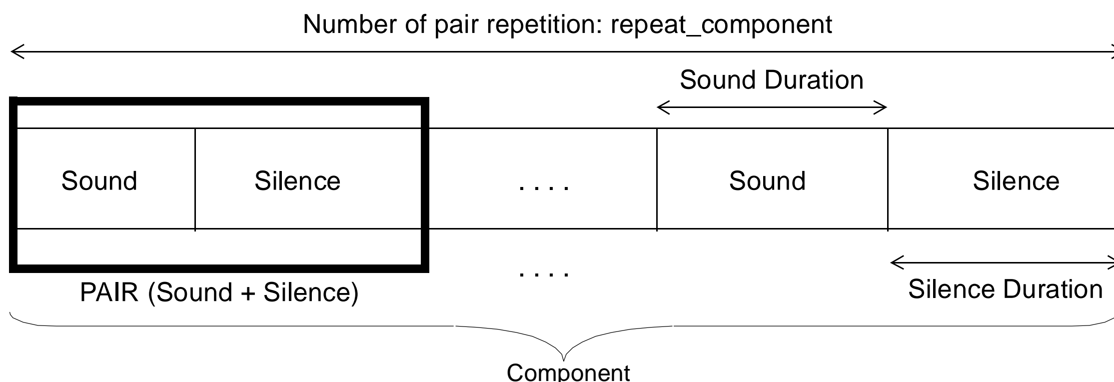

Figure 2. Elementary Tone Component

Each component is defined by the following parameters:

- *Frequencies* (in Hz).
- *Levels* (in dBm0). The signal amplitude per frequency is given as follows:

$$A = 10^{-Level(dBm0)/20} / \sqrt{2}$$

- *Sound duration*. The duration of the sound (in ms).
- *Silence duration*. The duration of the silence period (in ms) after the sound period. The software fills in the output buffer with zeroes to cover that period. A value of zero indicates no silence period.
- *repeat_component*. This value specifies the number of pairs (sound + silence) that must be generated to complete the component. A value of zero indicates that the pair should be repeated indefinitely.
- *Control word*. Defines whether the two sinusoids are to be added or modulated, the initial phase (0 or 180°), and the frequency unit (f or f/3). This flag is stored in 16 bits as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
—	—	—	—	—	—	P4	P3	P2	P1	U4	U3	U2	U1	C	M

- Bit 0: Add or modulate:
  — 0 The four frequencies are added.
  — 1 The four frequencies are modulated in pairs and then added (that is, f1 × f2 + f3 × f4).
- Bit 1: AM modulation type:
  — 0 AM modulation without carrier.
  — 1 AM modulation with carrier.
- Bit 2: Frequency unit 1:
  — 0 Frequency1.
  — 1 Frequency1 divided by 3.
- Bit 3: Frequency unit 2:
  — 0 Frequency2.
  — 1 Frequency2 divided by 3.

- Bit 4: Frequency unit 3:
  - — 0 Frequency3.
  - — 1 Frequency3 divided by 3.
- Bit 5: Frequency unit 4:
  - — 0 Frequency4.
  - — 1 Frequency 4 divided by 3.
- Bit 6: Phase 1:
  - — 0 Frequency1 initialized with 0° phase.
  - — 1 Frequency1 initialized with 180° phase.
- Bit 7: Phase 2:
  - — 0 Frequency2 initialized with 0° phase.
  - — 1 Frequency2 initialized with 180° phase.

  The unit bits produce frequencies with 1/3 Hz. For example, 16 2/3 Hz is generated with 50 Hz when the unit bit is set (16 2/3 Hz = 50/3 Hz).
- Bit 8: Phase 3:
  - — Frequency3 initialized with 0° phase.
  - — Frequency3 initialized with 180° phase.
- Bit 9: Phase 4:
  - — Frequency4 initialized with 0° phase.
  - — Frequency4 initialized with 180° phase.

2.2 Tone Cycles

When all components are ready, they must be gathered into the desired tone. A component group is called a cycle, as shown in **Figure 3**. Up to six components can be grouped into one cycle. Finally, the created cycle can be repeated as often as needed to form the signaling tone.

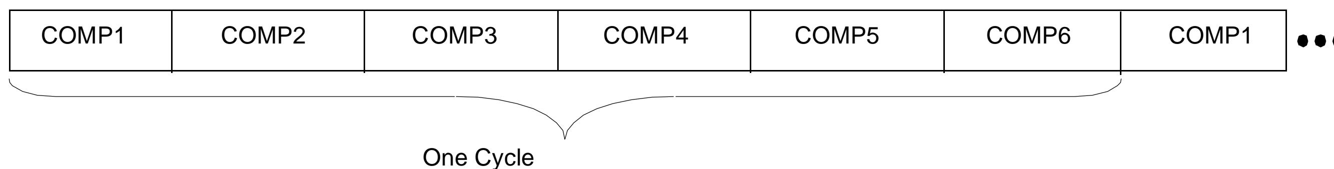

Figure 3. Cycle Components

2.3 Functional Interface

There are many different types of tones, such as single, dual, partial dual, and modulated. The starting times of sinusoids with different frequencies are arbitrary, but they are easily synchronized within the components. The external interface to the software consists of the following C functions (see **Figure 4**):

```
void fsl_tone_gen_init (GEN_CHANNEL *gen_channel, TONE_GENERIC *tone_control);
void fsl_tone_gen (GEN_CHANNEL *gen_channel, SIGNAL *linear_out, unsigned samples_out);
```

- *TONE_GENERIC* is a structure that describes the tone to be generated (tone descriptor).
- *SIGNAL* is a 16-bit signed fractional type.
- *linear_out* is an output buffer.
- *samples_out*, a multiple of eight, indicates how many samples the software must generate in this call.
- *GEN_CHANNEL* is a structure containing the description of the tone to be generated and the actual internal states of the generation process.
- *gen_channel* is a channel-dependent instance of `GEN_CHANNEL` containing all channel data for generation.

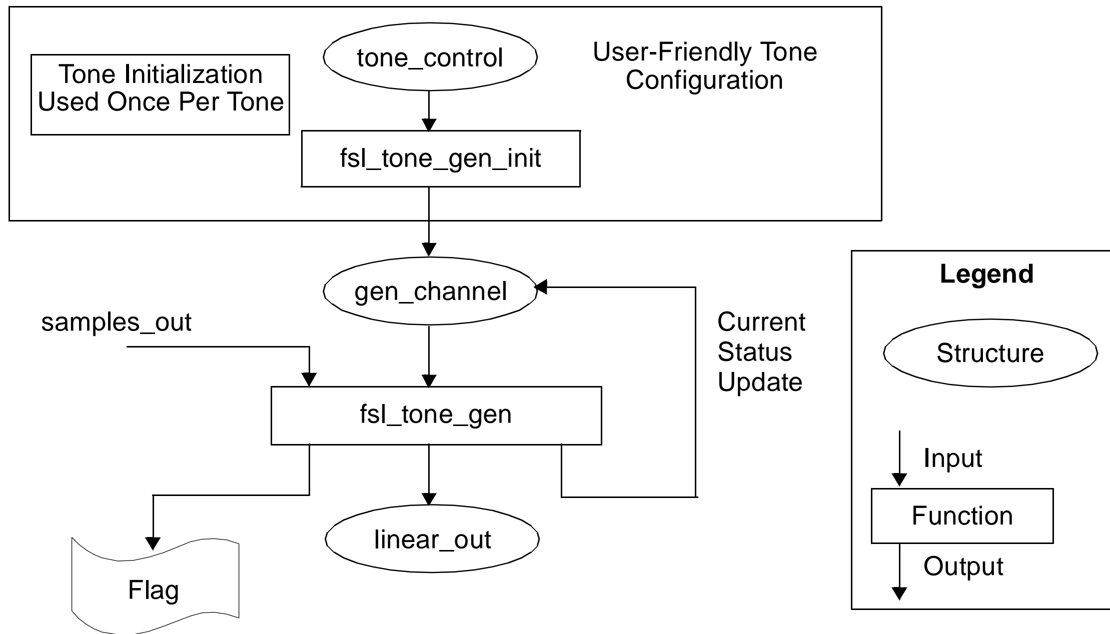

Figure 4. External Interface to Software

`fsl_tone_gen` returns the following integer flag:

- Bit 0 (LSB): Indicates whether tone generation is complete.
  — 0 Generation is not complete.
  — 1 Generation is complete.
- Bit 1: Indicates whether the content of the entire buffer is silence.
  — 0 Buffer or part of the buffer contains sound.
  — 1 Entire buffer is silence

The main structures are defined as follows:

```
typedef struct {
                UINT32 reserved;
                UINT16 tone_repeat_cycle;
                UINT16 tone_component_count;
                TONE_COMPONENT tone_component[6];
        } TONE_GENERIC;
```

```
typedef struct {
                    UINT32  sound_duration;
                    UINT32  silence duration;
                    UINT16  flags;
                    UINT16  repeat_component;
                    UINT16  frequency[4];
                    UINT8   level[4];
               } TONE_COMPONENT;
```

where:

- *tone_repeat_cycle*. Number of cycle repetitions needed to generate the full tone.
- *tone_component_count*. Number of components in the cycle.
- *tone_component*[6]. An array that contains the description of every component.

TONE_COMPONENT parameters are described at the beginning of this section. The TONE_GENERIC structure is not persistent, so the `fsl_tone_gen_init` function does not save pointers to any part of this structure. After `fsl_tone_gen_init` returns, the software copies any information that is needed in the channel-dependent structure gen_channel. If the M bit is set to 1, the level of the smallest frequency in a given pair ((f1, f2) or (f3, f4)) represents the AM modulation depth in percent; the largest frequency is the carrier.

2.4 Implementation Constraints

The implementation constraints apply to both tone initialization and tone generation.

2.4.1 Tone Initialization

The coefficients for all frequencies are calculated with a cosine polynomial approximation during the initialization routine, which restricts the coefficient precision to 14 bits. We tested the error introduced by this method with MATLAB simulations and found the error to be within the one percent tolerance required by ITU-T standards. We decreased the full-scale level to 0x7FC0 (instead of 0x7FFF) to avoid saturation. The impact of such an "under-level" is negligible after A- or µ-law encoding stages [4]. This 0 dB level can be considered as a full scale one. Half scale tones are often preferred.

The initialization is divided into two parts:

- Creating a GEN_CHANNEL structure in conformance with the user's requirements, computing coefficients and duration for each component and copying any information needed after `fsl_tone_gen_init` returns.
- Updating the channel-dependent structure for every new component using an internal function, which updates the initial samples (x(–2) and x(–1)).

Each time new samples are produced, the number of sound and silence samples is updated in `fsl_tone_gen`. When four frequencies are defined, they can either be added or modulated. Modulation is performed in pairs (that is, f1 × f2, f3 × f4) and the resulting modulated signals are added to produce the output signal.

2.4.2 Tone Generation

The SC140 multiple ALU architecture allows simultaneous generation of samples for both frequencies. The performance is optimal with two frequencies because required data can be loaded and updated in a single clock cycle: four fractional moves for the four previous samples and two fractional moves for the two coefficients.

To reduce MCPS, blocks of eight samples are generated. Furthermore, duration times are given in ms (number of samples is 8*(duration time)) so that eight samples are generated as grouped bursts. Each component includes a silence duration after a sound period. If a property of the tone must be changed without prior silence, the silence duration must be set to zero at the end of the previous component.

If a frequency level is 63 (–63 dBm0), the initial filter states are initialized with zeroes so that no samples are generated for this frequency. If a single frequency is generated with the modulation flag set, the result is silence. When a continuous tone must be generated, it is better to repeat a component with a long sound_duration because the code runs faster than with a short sound_duration repeated very often. Note that the filter state is initialized between two cycles, but not between two occurrences of the same component without silence in between. If only one component is used, it is better to repeat the component, not the tone cycle, to prevent harmonic distortions.

2.5 Status Reporting

The top-level modules are informed of the tone generation status with the return values given by `fsl_tone_gen`. The return flag bits are as follows:

- Bit 0 (LSB): indicates whether tone generation is complete.
  - 1 Complete.
  - 0 Not complete.
- Bit 1: indicates whether the content of the buffer is silence.
  - 1 Entire buffer is silence.
  - 0 Buffer or part of the buffer contains sound.

We recommend that silence be generated with the silence part of a component, not with the sound part initialized with null level (–63 dBm0) frequencies. Otherwise, the return flag does not reflect reality.

The “entire buffer is silence” flag detects whether the program has run the generation routine. If the generated sound is composed of null-level frequencies, the buffer contains only silence, but the returned flag indicates that sound has been generated. This approach saves MCPS in return flag computation (for testing every sample that would not be used).

2.6 Tone Generator Code Structure

Figure 5 shows an overview of the tone generation code structure. The main goal of the tone generation function is to generate the tone when required and update the associated state description structure.

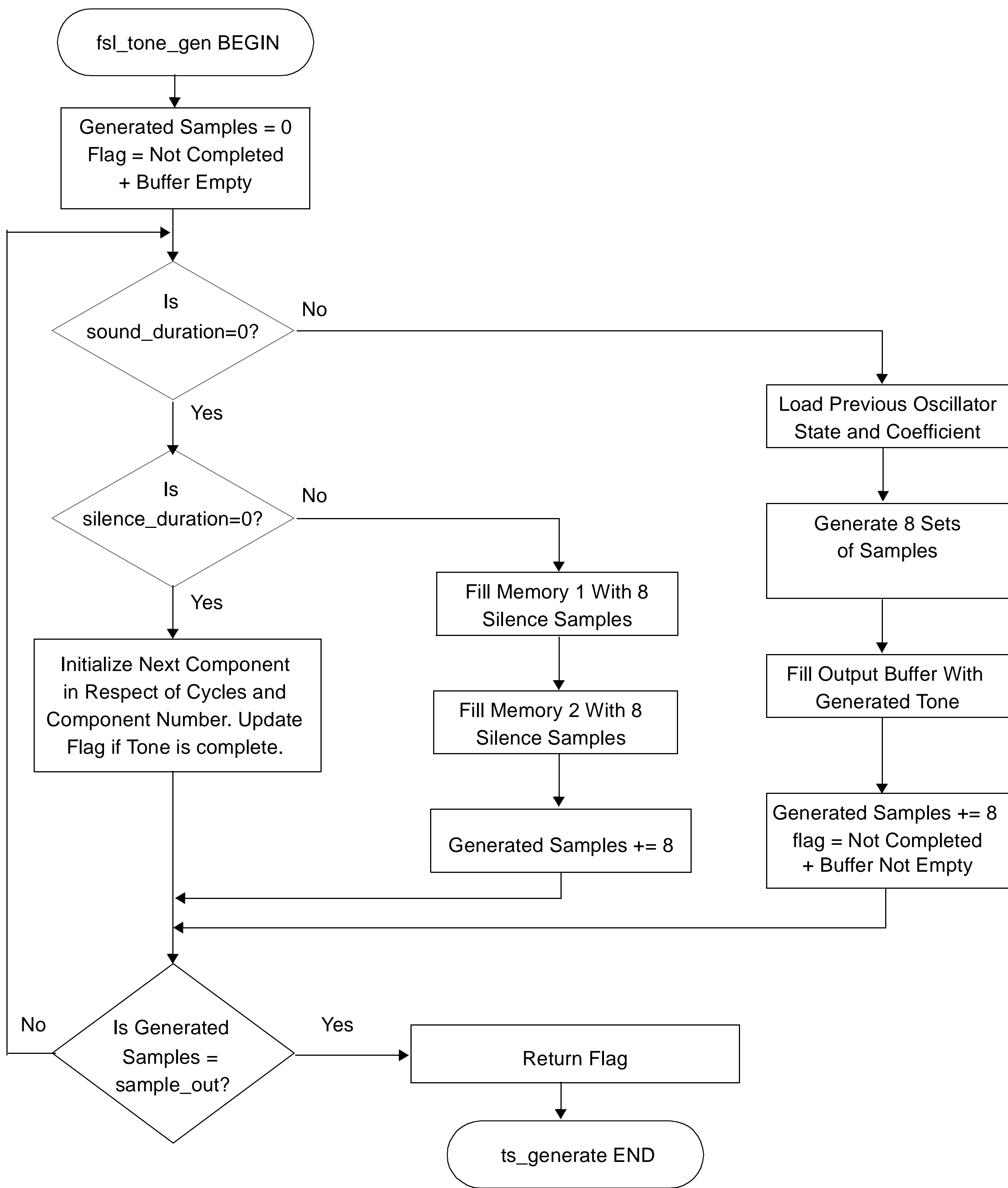

Figure 5. Structure of the Tone Generator Code

Example 1. Tone Generator Pseudo Code

```
Function name:              fsl_tone_gen
Input:                      Pointer to output
                            Pointer to buffer containing static variables
                            Number of samples to generate
Output:                     Return Flag
BEGIN
       FOR (Number of sample to generate/8)
              IF (sound duration is expired)
                     IF (silence duration is not expired)
                            Fill memory with 8 silence samples
                     ELSE
                            Initialize next component with respect to cycles
                            and component number.
                            Update flag if tone is complete.
                     END IF
              ELSE
                     FOR (8 couples of samples)
                     Load previous oscillator states and coefficients
                     Generate individual tone frequencies
                     Fill output buffer with generated tone
                     Update flag (sound generated)
                     END FOR
              END IF
       END FOR
       RETURN (Flag)
END BEGIN
```

3 Using and Testing the Generic Tone Generator

We integrated and tested the generic tone generator on the MSC8101 Application Development System (MSC8101ADS) board [2] to confirm MATLAB simulation results. Many different tones were measured.

3.1 Test Set-up

On the MSC8101ADS, an audio codec is connected to an MSC8101 device through one of its CPM multi-channel controllers (MCCs), which configures the codec, and one of its serial peripheral interfaces (SPIs), which sends audio samples out of the ADS. **Figure 6** shows a diagram of this testing set-up. Alternatively, the ADSFS set-up [5] can be used, allowing generated samples to be written into files using the Ethernet port with a high transfer rate. In this case, the codec is not used.

3.2 Initialization Examples of Multi-Frequency Tones

The examples shown here provide specific configurations of the initialization function to generate several sets of tones composed by one or two frequencies. Because frequency values, power levels, and durations may vary from one case to another, the generic character of the generator is evident. Modifying the code to other types of signaling tones should be a fairly simple task.

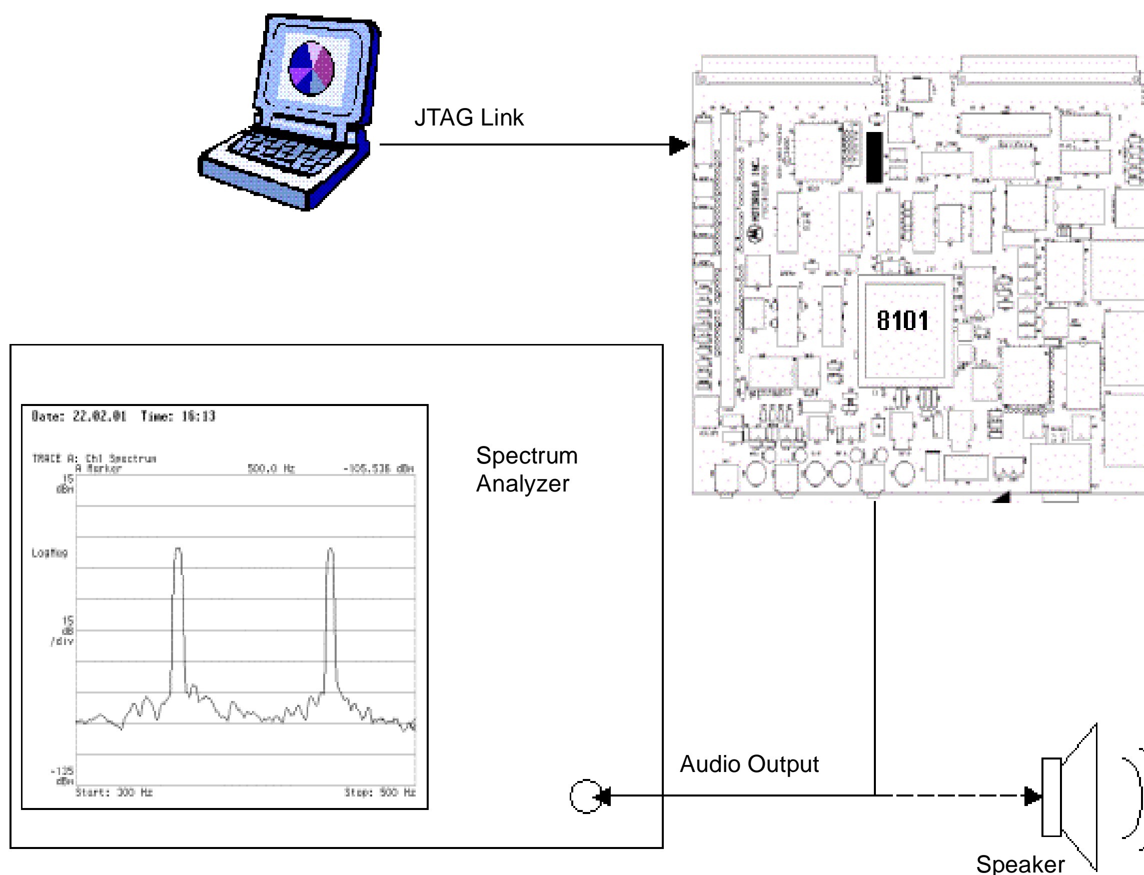

Figure 6. Testing Set-Up

Example 2. DTMF

```
UINT16 DTMFl[] = {697, 770, 852, 941};
UINT16 DTMFh[] = {1209, 1336, 1477, 1633};
void
test_initialize(TONE_GENERIC * testcases)
// testcases must point to a 16-item TONE_GENERIC structure
{
            int i, j;
            for (i = 0; i < 16; i++) {
                  // Inialize the input structures for the i+1 key
                  testcases[i].tone_repeat_cycle = 1;
                  testcases[i].tone_component_count = 1;
                  // 1 tone_component description
                  testcases[i].tone_component[0].flags = 0x0;
                  testcases[i].tone_component[0].repeat_component = 1;
                  testcases[i].tone_component[0].sound_duration = 50;
                  testcases[i].tone_component[0].silence_duration = 50;
                  testcases[i].tone_component[0].level[0] = 10;
                  testcases[i].tone_component[0].level[1] = 10;
                  testcases[i].tone_component[0].level[2] = 63;
                  testcases[i].tone_component[0].level[3] = 63;
            }
```

```
                // DTMF digits
                // Key:   1| 2| 3| 4| 5| 6| 7| 8| 9| 0| *| #| A| B| C| D
                // Code:  1| 2| 3| 5| 6| 7| 9|10|11|14|13|15| 4| 8|12|16
                for (i = 0; i < 4; i++) {
                for (j = 0; j < 4; j++) {
                        testcases[4 * i + j].tone_component[0].frequency[0] = DTMFl[i];
                        testcases[4 * i + j].tone_component[0].frequency[1] = DTMFh[j];
                        testcases[4 * i + j].tone_component[0].frequency[2] = 0;
                        testcases[4 * i + j].tone_component[0].frequency[3] = 0;
                        }
                }
}
```

Example 3. MF-R1

```
UINT16 R1[] = {700, 900, 1100, 1300, 1500, 1700};
void
test_initialize(TONE_GENERIC * testcases)
// testcases must point to a 15-item TONE_GENERIC structure
{
                int i, j, k = 0;
                for (i = 0; i < 15; i++) {
                        // Inialize the input structures
                        testcases[i].tone_repeat_cycle = 1;
                        testcases[i].tone_component_count = 1;
                        // 1 tone_component description
                        testcases[i].tone_component[0].flags = 0x0;
                        testcases[i].tone_component[0].repeat_component = 1;
                        testcases[i].tone_component[0].sound_duration = 50;
                        testcases[i].tone_component[0].silence_duration = 50;
                        testcases[i].tone_component[0].level[0] = 10;
                        testcases[i].tone_component[0].level[1] = 10;
                        testcases[i].tone_component[0].level[2] = 63;
                        testcases[i].tone_component[0].level[3] = 63;
                }
                // MF-R1 digits
                // Key:   1| 2| 3| 4| 5| 6| 7| 8| 9| 0| ST3P| STP| KP| STP2| ST
                // Code:  1| 2| 3| 4| 5| 6| 7| 8| 9|10|   11|  12| 13|   14| 15
                for (j = 1; j < 6; j++) {
                        for (i = 0; i < j; i++) {
                                testcases[k].tone_component[0].frequency[0] = R1[i];
                                testcases[k].tone_component[0].frequency[1] = R1[j];
                                testcases[k].tone_component[0].frequency[2] = 0;
                                testcases[k++].tone_component[0].frequency[3] = 0;
                        }
                }
}
```

Example 4. MF-R2

```
UINT16 R2f[] = {1380, 1500, 1620, 1740, 1860, 1980};
UINT16 R2b[] = {1140, 1020, 900, 780, 660, 540};
void
test_initialize(TONE_GENERIC * testcases)
// testcases must point to a 30-item TONE_GENERIC structure
// First 15 for MF-R2 foward direction
// Last 15 for MR-R2 backward direction
{
                int i, j, k = 0;
```

```
                for (i = 0; i < 30; i++) {
                        // Inialize the input structures for the i+1 key
                        testcases[i].tone_repeat_cycle = 1;
                        testcases[i].tone_component_count = 1;
                        // 1 tone_component description
                        testcases[i].tone_component[0].flags = 0x0;
                        testcases[i].tone_component[0].repeat_component = 1;
                        testcases[i].tone_component[0].sound_duration = 50;
                        testcases[i].tone_component[0].silence_duration = 50;
                        testcases[i].tone_component[0].level[0] = 10;
                        testcases[i].tone_component[0].level[1] = 10;
                        testcases[i].tone_component[0].level[2] = 63;
                        testcases[i].tone_component[0].level[3] = 63;
                }
                // 15 MF-R2 forward direction digits
                for (j = 1; j < 6; j++) {
                        for (i = 0; i < j; i++) {
                                testcases[k].tone_component[0].frequency[0] = R2f[i];
                                testcases[k].tone_component[0].frequency[1] = R2f[j];
                                testcases[k].tone_component[0].frequency[2] = 0;
                                testcases[k++].tone_component[0].frequency[3] = 0;
                        }
                }
                // 15 MF-R2 backward direction digits
                for (j = 1; j < 6; j++) {
                        for (i = 0; i < j; i++) {
                                testcases[k].tone_component[0].frequency[0] = R2b[i];
                                testcases[k].tone_component[0].frequency[1] = R2b[j];
                                testcases[k].tone_component[0].frequency[2] = 0;
                                testcases[k++].tone_component[0].frequency[3] = 0;
                        }
                }
}
```

Example 5. Germany Dial Tone

```
void test_initialize(TONE_GENERIC * testcases)
{
                // Inialize the input structures
                testcases[0].tone_repeat_cycle = 0;
                testcases[0].tone_component_count = 1;
                // tone_component description
                testcases[0].tone_component[0].flags = 0x0;
                testcases[0].tone_component[0].repeat_component = 0;
                testcases[0].tone_component[0].sound_duration = 1000;
                testcases[0].tone_component[0].silence_duration = 0;
                testcases[0].tone_component[0].frequency[0] = 425;  // 425 or 450
                testcases[0].tone_component[0].level[0] = 0;
                // Only one frequency is used - level is initialized to -63dB
                testcases[0].tone_component[0].frequency[1] = 0;
                testcases[0].tone_component[0] .level[1] = 63;
                testcases[0].tone_component[0].frequency[2] = 0;
                testcases[0].tone_component[0] .level[2] = 63;
                testcases[0].tone_component[0].frequency[3] = 0;
                testcases[0].tone_component[0] .level[3] = 63;

}
```

Example 6. Japan Waiting Tone

```
void test_initialize(TONE_GENERIC * testcases)
{
        // Inialize the input structures
        testcases[0].tone_repeat_cycle = 0;
        testcases[0].tone_component_count = 2;
        // 2 tone_component description
        // COMPONENT 1
        testcases[0].tone_component[0].flags = 0x0;
        testcases[0].tone_component[0].repeat_component = 1;
        testcases[0].tone_component[0].sound_duration = 100;
        testcases[0].tone_component[0].silence_duration = 100;
        testcases[0].tone_component[0].frequency[0] = 440;
        testcases[0].tone_component[0].level[0] = 0;
        // Only one frequency is used - level is initialized to -63db
        testcases[0].tone_component[0].frequency[1] = 0;
        testcases[0].tone_component[0].level[1] = 63;
        testcases[0].tone_component[0].frequency[2] = 0;
        testcases[0].tone_component[0].level[2] = 63;
        testcases[0].tone_component[0].frequency[3] = 0;
        testcases[0].tone_component[0].level[3] = 63;
        // COMPONENT 2
        testcases[0].tone_component[1].flags = 0x0;
        testcases[0].tone_component[1].repeat_component = 1;
        testcases[0].tone_component[1].sound_duration = 100;
        testcases[0].tone_component[1].silence_duration = 1000;
        testcases[0].tone_component[1].frequency[0] = 440;
        testcases[0].tone_component[1].level[0] = 0;
        // Only one frequency is used - level is initialized to -63db
        testcases[0].tone_component[1].frequency[1] = 0;
        testcases[0].tone_component[1].level[1] = 63;
        testcases[0].tone_component[1].frequency[2] = 0;
        testcases[0].tone_component[1].level[2] = 63;
        testcases[0].tone_component[1].frequency[3] = 0;
        testcases[0].tone_component[1].level[3] = 63;
}
```

3.3 Tests

The following MATLAB script displays the FFT of generated tones. A Hanning windowed FFT is used to measure the output frequency. A typical frequency accuracy requirement is about ±1% of the nominal frequency. At an 8 kHz sampling rate, we must use at least 4096 samples (512 ms) to ensure that the error generated by the FFT is not prejudicial.

Example 7. Frequency Response of Generated Tones

```
disp('*** Generic Tone Generation on StarCore SC140 ***')
disp(' '), disp(' ')
REF = '.wav'; TONE_NAME = 'DTMF'; BITS    = 16;
SAMPLES = 8 * 50; SILENCE = 8 * 50; SPACE   = SAMPLES + SILENCE;
if TONE_NAME == 'DTMF'
              number_of_digits = 16; fid = fopen(['gen_dtmf' REF], 'r');
elseif TONE_NAME == 'MFR1'
              number_of_digits = 15; fid = fopen(['gen_mfr1' REF], 'r');
elseif TONE_NAME == 'MFR2'
              number_of_digits = 30; fid = fopen(['gen_mfr2' REF], 'r');
else
              number_of_digits = 0; disp('Unknown tone.')
end
if number_of_digits > 0
              x = fread(fid, inf, 'short');
              disp(['Playing ' TONE_NAME ' generated tones...'])
              wavplay(x, 8000);
              for n = 1 : number_of_digits
                     disp(['Plotting ' TONE_NAME ' tone ' int2str(n) '...'])
                     out = x(SPACE + (n - 1) * (SAMPLES + SILENCE + SPACE) + ...
                     (1 : SAMPLES)) .* 2 ^ -(BITS - 1);
                     abscisse = linspace(0, 4000, SAMPLES / 2);
                     fft_out = 20*log10( abs( fft( hanning( SAMPLES) .* out)));
                     fft_out = fft_out - max(fft_out);
                     [a, b] = sort(-fft_out ( 1 : SAMPLES / 2));
                     [freq1, c] = sort(round(abscisse(1, b(1:2))));
                     figure, subplot(211)
                     plot(out), grid on
                     xlabel('Sample #')
                     title([TONE_NAME ' Tone ' int2str(n)])
                     subplot(212), plot(abscisse, fft_out(1 : SAMPLES / 2))
                     grid on, hold on
                     plot(freq(1), -a(c(1)), 'bo'), plot(freq1(2), -a(c(2)), 'ro')
                     legend(['FFT of Tone ' int2str(n)], ...
                            ['F_1 = ' int2str(freq1(1)) ' Hz'], ...
                            ['F_2 = ' int2str(freq1(2)) ' Hz'])
                     xlabel('Frequency (Hz)'), ylabel('Magnitude (dB)')
              end
end
disp('Test complete.')
fclose(fid);
```

Example 8. Driver for Generating Time Domain Samples Using ADSFS [5]

```
#include "fsl_tone_gen.h"
#define TBUFSZ 4096
#define DTMF
//#define MFR1
//#define MFR2
#ifdef DTMF
#define NUMBER_OF_TONES 16
#endif
#ifdef MFR1
#define NUMBER_OF_TONES 15
#endif
#ifdef MFR2
#define NUMBER_OF_TONES 30
#endif
void test_pgm ()
{
fract16 samples[TBUFSZ];
#pragma align samples 8
              char samples2[10];
              Word32 txid; // to write samples in text files
              char file_name[20];
              int channel, len;
              TONE_GENERIC testcases[NUMBER_OF_TONES];
              GEN_CHANNEL gen_channel[1];
              int flag;
              test_initialize(testcases);
              for (channel = 0; channel < NUMBER_OF_TONES; channel++) {
                     // Opens a file
                     sprintf(file_name,"c:\\gen_out%d.dat",channel);
                     txid = open_file (file_name, WRITEONLY);
                     if (txid <= 0)
                            error (1);
                     // Tone generation
                     fsl_tone_gen_init (gen_channel, &testcases[channel]);
                     flag = fsl_tone_gen (gen_channel, samples, TBUFSZ);
                     // Saves samples
                     for (len=0; len<TBUFSZ; len++) {
                            write_file(txid,(unsigned char *)samples2,
                            sprintf(samples2,"%d\n",samples[len]));
                     }
                     close_file (txid);
               }
}
```

3.3.1 DTMF Frequency Accuracy Test

The digital oscillator algorithm has reasonable frequency accuracy. The tests listed in **Table 2** validate the use of the digital oscillator algorithm for DTMF generation, showing that it complies with the requirements of the ITU-T Recommendation Q.24 [6] for frequency accuracy. This test uses 4096 samples, generated with a –20 dBm0 level (11 percent of maximum scale). The measured DTMF tone precision is smaller than 0.2 percent.

Table 2. Digital Oscillator Algorithm for DTMF Generation

Frequency (Hz)	Precision (Percent)
697	0.12
770	0.08
852	0.16
941	0.14
1209	0.03
1336	0.06
1477	0.05
1633	0.07

3.3.2 MF-R2 Frequency Accuracy Test

The module capabilities were tested on the most constraining standard for frequency accuracy: MF-R2 signaling defined in the ITU-T Recommendation Q.454 [7]. A deviation of up to 4 Hz is tolerated on generated frequencies. Each frequency is measured with an 8192 point Hanning windowed FFT. At an 8 kHz sampling rate, the resulting accuracy is smaller than 1 Hz. In **Table 3** and **Table 4**, an error of 0 Hz means that the test is unable to detect any frequency deviation.

Table 3. Forward Frequencies Test

	Frequency (Hz)	Measured Frequency (Hz)	Error (Hz)	R2 Test
F0	1380	1379.6	0.4	PASS
F1	1500	1499.7	0.3	PASS
F2	1620	1619.8	0.2	PASS
F3	1740	1740.0	0.0	PASS
F4	1860	1859.1	0.9	PASS
F5	1980	1979.2	0.8	PASS

Table 4. Backward Frequencies Test

	Frequency (Hz)	Measured Frequency (Hz)	Error (Hz)	R2 Test
F0	1140	1139.3	0.7	PASS
F1	1020	1019.2	0.8	PASS
F2	900	900.0	0.0	PASS
F3	780	779.9	0.1	PASS
F4	660	659.7	0.3	PASS
F5	540	539.6	0.4	PASS

All tests of the tone generator passed. Our results indicate that the proposed tone generation module complies with the requirements for frequency accuracy of the ITU-T Recommendation Q.454 for MF-R2.

3.4 Examples of Generated Signaling Tones

Figure 7 and **Figure 8** illustrate the spectrum of two typical signaling tones. The figures on the left correspond to the FFT magnitude plots of actual data generated from the proposed module; the figures on the right correspond to measured spectra (using a spectrum analyzer). Observe that the noise floor of the measurements is around 80 dB below the peak values.

MATLAB	Measured Spectrum

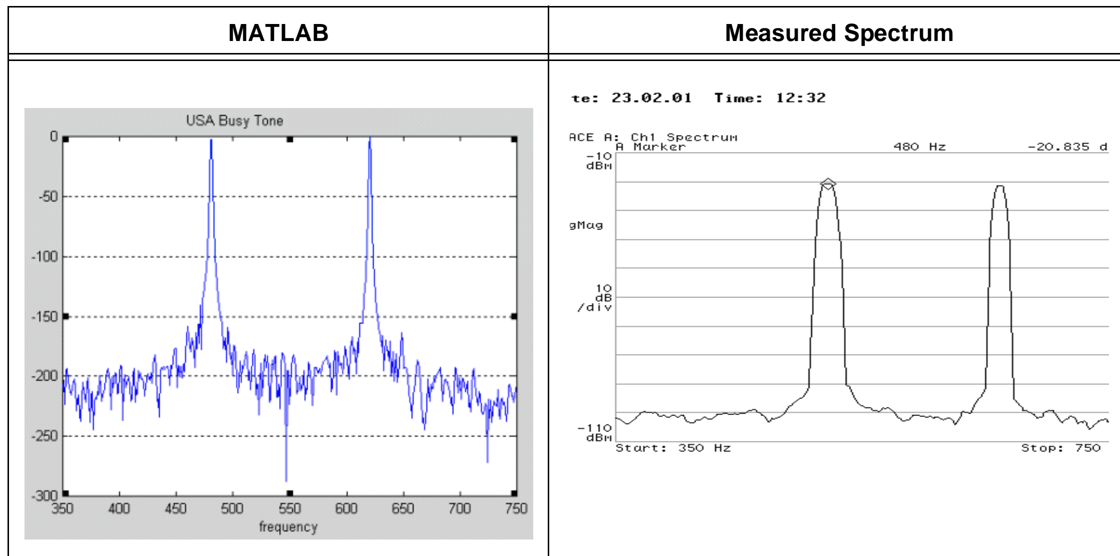

Figure 7. USA Busy Tone: 480 + 620 Hz

MATLAB	Measured Spectrum

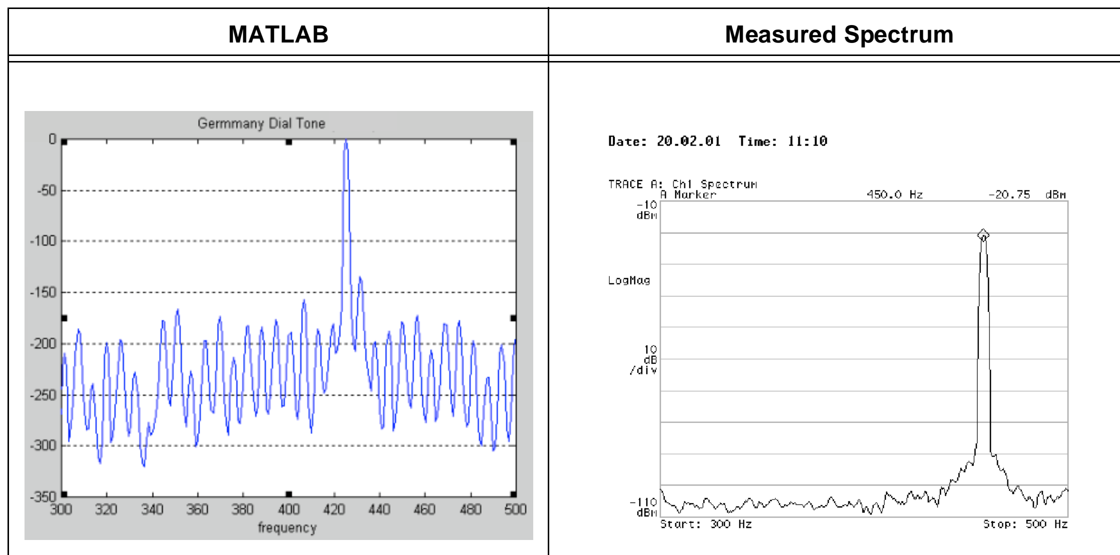

Figure 8. Germany Dial Tone: 425 or 450 Hz

3.5 Performance Measurements

The proposed tone generation module typically requires less than 0.15 MCPS. The worst case would be to change the component every millisecond, but this mode of operation is not used in practice (typical component change intervals are greater than 40 ms).

Table 5. Typical Average MCPS Values

Tone	Initialization (Cycles)	Generation (MCPS)
DTMF	945	0.073
Germany Dial Tone	560	0.106
Japan Waiting Tone	560	0.47

Table 6. Memory Usage

Data / Channel (Bytes)	Table Size (Bytes)	Stack Size (Bytes)	Program Size (Bytes)
272	156	24	2828

4 References

[1] ITU-T Recommendation E.180/Q.35, Technical characteristics of tones for the telephone service.

[2] *MSC8101 Application Development System User's Manual,* Freescale Semiconductor, Inc.

[3] A. V. Oppenheim, R. W. Schafer, J. R. Buck. *Discrete Time Signal Processing*. Prentice Hall, Second edition, 1999.

[4] *ITU-T Recommendation G.711, Pulse Code Modulation (PCM) of Voice Signals*, 1993.

[5] *Fast I/O Library for the MSC8101ADS Using Ethernet Communication* (AN2260), Freescale Semiconductor, Inc., 2004.

[6] ITU-T Recommendation Q.24, Multi-frequency push bottom signal reception, 11/98.

[7] ITU-T Recommendation Q.454, The Sending Part of the Multi-Frequency Signalling Equipment.

NOTES:

NOTES:

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

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters which may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals” must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. StarCore is a trademark of StarCore LLC. The StarCore DSP SC1400 core is based on StarCore technology under license from StarCore LLC. All other product or service names are the property of their respective owners.

© Freescale Semiconductor, Inc. 2002, 2005.

AN2274
Rev. 2
9/2005