Embedded SDK (Software Development Kit)

Full Duplex Speakerphone Library

SDK141/D
Rev. 3, 1/13/2003

Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

Contents

Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

List of Tables

Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

List of Figures

Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

List of Examples

Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

About This Document

This manual describes the Full Duplex Speakerphone Library for use with Motorola's Embedded Software Development Kit (SDK).

Audience

This document targets software developers implementing communication features for analog telephone lines.

Organization

This manual is arranged in the following sections:

- **Chapter 1, Introduction**—provides a brief overview of this document
- **Chapter 2, Directory Structure**—provides a description of the required core directories
- **Chapter 3, Full Duplex Speakerphone Library Interfaces**—describes all of the Full Duplex Speakerphone Library functions
- **Chapter 4, Building the Full Duplex Speakerphone Library**—tells how to build a test application project with the pre-built Full Duplex Speakerphone Library
- **Chapter 5, Linking Applications with the Full Duplex Speakerphone Library**—describes linking projects with the Full Duplex Speakerphone Library
- **Chapter 6, Full Duplex Speakerphone Library Applications**—describes the use of the Full Duplex Speakerphone Library through test/demo applications
- **Chapter 7, License**—provides the license required to use this product

Suggested Reading

We recommend that you have a copy of the following references:

- *Motorola DSP56800E Reference Manual*, DSP56800ERM/D
- *Motorola DSP568xx User's Manual*, for the DSP device you're implementing
- *Inside CodeWarrior: Core Tools*, Metrowerks Corp.

Conventions

This document uses the following notational conventions:

Typeface, Symbol or Term	Meaning	Examples
`Courier Monospaced Type`	Code examples	`//Process command for line flash`
Italic	Directory names Project names Calls Functions Statements Procedures Routines Arguments File names Applications Variables Directives Code snippets in text	...and contains these core directories: *applications* contains applications software... ...CodeWarrior project, *3des.mcp* is... ...the *pConfig* argument.... ...defined in the C header file, *aec.h*....
Bold	Reference sources Paths Emphasis	...refer to the **Targeting DSP5685x Platform** manual.... ...see: **C:\Program Files\Motorola\Embedded SDK\help\tutorials**
ALL CAPITAL LETTERS	# defines/ Defined constants	# define INCLUDE_STACK_CHECK
Brackets [...]	Function keys	...by pressing function key [F7]
Quotation marks, "..."	Returned messages	...the message, "Test Passed" is displayed.... ...if unsuccessful for any reason, it will return "NULL"...
Blue Text	Linkable on-line	...refer to Chapter 7, License....
Number	Any number is considered a positive value, unless preceded by a minus symbol to signify a negative value	3V -10 DES^{-1}

Definitions, Acronyms, and Abbreviations

The following list defines the acronyms and abbreviations used in this document. As this template develops, this list will be generated from the document. As we develop more group resources, these acronyms will be easily defined from a common acronym dictionary. Please note that while the acronyms are in solid caps, terms in the definition should be initial capped ONLY IF they are trademarked names or proper nouns.

ADC Analog-to-Digital Converter

AGC	Automatic Gain Control
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BPF	Bandpass Filter
CID	Caller ID (Calling Party Name and/or Number Identification)
CPE	Customer Premises (telephony) Equipment
CQ	Call Qualifier
DAA	Data Access Arrangement
DSP	Digital Signal Processor or Digital Signal Processing
DTMF	Dual Tone Multiple Frequency
FSK	Frequency Shift Keying modulation
IDE	Integrated Development Environment
LPF	Low Pass Filter
MDMF	Multiple Data Message Format of GR-30-CORE
MIC	Microphone
MIPS	Million Instructions Per Second
OnCE™	On-Chip Emulation
PC	Personal Computer
PCM	Pulse Code Modulation
PSTN	Public Switched Telephone Network
SDK	Software Development Kit
SDMF	Single Data Message Format of GR-30-CORE
SRC	Source
VMWI	Visual Message Waiting Indicator

References

The following sources were referenced to produce this book:

1. *Motorola DSP56800E Reference Manual*, DSP56800ERM/D
2. *Motorola DSP568xx User's Manual*, for the DSP device you're implementing
3. *Targeting Motorola DSP568xx Platform*, for the DSP device you're implementing
4. *Motorola Embedded SDK Programmer's Guide,* SDK101/D
5. SR-3004, *Testing Guidelines for Analog Type 1, 2, and 3 CPE as Described in SR-INS-002726 (a module of ADSI, FR-12),* Telcordia Technologies, January 1995.

Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

Chapter 1
Introduction

Welcome to Motorola's family of Digital Signal Processors, or DSPs. This document describes the Full Duplex Speakerphone Library, which is a part of Motorola's comprehensive Software Development Kit, SDK, for its DSPs. In this document, you will find all the information required to use and maintain the Full Duplex Speakerphone Library interface and algorithms.

Motorola provides these algorithms to you under license for use with Motorola DSPs to expedite your application development and reduce the time it takes to bring your own products to market.

Motorola's Full Duplex Speakerphone Library is a licensed software library for use on Motorola DSP56800E series processors. Please refer to the Software License Agreement in **Chapter 7** for license terms and conditions.

1.1 Quick Start

Motorola's Embedded SDK is targeted to a large variety of hardware platforms. To take full advantage of a particular hardware platform, use **Quick Start** from the **Targeting Motorola DSP5685x Platform** documentation.

For example, the **Targeting Motorola DSP5685x Platform** manual provides more specific information and examples about this hardware architecture. If you are developing an application for the DSP56858EVM board, or any other DSP56858 development system, refer to the **Targeting Motorola DSP5685x Platform** manual for **Quick Start** or other DSP56858-specific information.

Note: "DSP568xx" refers to the specific device for which you're developing, as shown in the preceding example.

1.2 Telephony Features Libraries

The Full Duplex Speakerphone Library is one of a set of Motorola Embedded SDK modules and applications consisting of the following:

- Type 1 Telephony Features Library
- Type 1 and 2 Telephony Features Library
- Type 1 and 2 Telephony Parser Library
- Full Duplex Speakerphone Library

- Generic Echo Canceller Library
- Feature Phone Application Software

These modules are designed to interoperate to provide all of the software necessary to implement a feature phone with full duplex speakerphone and Type 1 and 2 Caller ID functionality. Using some or all of these modules, several other types of telephony applications are also possible. Each module may also be used independently.

1.3 Overview of the Full Duplex Speakerphone Library

This library implements the algorithm for a Full Duplex Speakerphone in Customer Premises Equipment (CPE). It uses two adaptive echo cancellers, with variable filter lengths, to cancel echoes on both the line side and on the acoustic side. It also has an elaborate suppression analysis algorithm to compliment the echo cancellers, making it completely full duplex in nature.

1.3.1 Background

The source of echoes in a speakerphone can be understood by considering a simplified connection between two telephone subscribers one of whom is using a speakerphone.

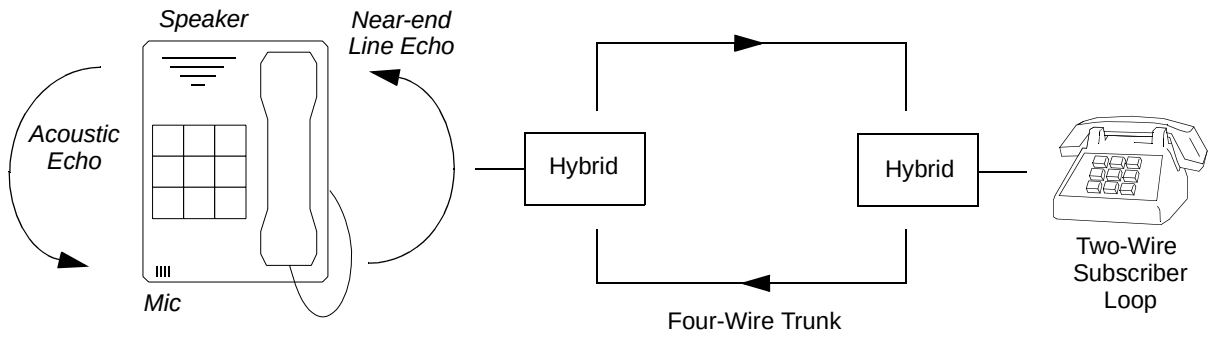

Figure 1-1. The Two Echoes

As shown in **Figure 1-1**, there are two sets of echoes, Acoustic Echo and Line Echo, that must be cancelled for improved quality in communication; digital filters are used for this task. However, as the exact nature of the echo is impossible to predetermine, the only way to effectively cancel the echoes is by using Adaptive Filters. In this way, even if the echo characteristics change, usually due to change in echo paths, the adaptive filters will readjust to the new echo characteristics. The LMS algorithm is used to train the adaptive filters. For the adaptive filters to be able to train to the present echo, it is important to be able to determine which side is speaking and which side is silent, or if both sides are talking (called *double talk*). This determination requires a voice activity detection algorithm. Since there will be situations when the echo cancellers, which are adaptive in nature, will require time to adapt to changed or new echoes, it also becomes necessary to suppress signals on the side that is not talking. Suppression analysis is necessary to be able to apply and remove suppression according to the training of the adaptive echo cancellers. Suppression analysis and voice activity detection go hand-in-hand, therefore simplifying the entire algorithm.

1.3.2 Features and Performance

This module contains a complete acoustic echo cancellation system and logarithmic suppression controller. The module has a simple interface with the Generic Echo Canceller software module, which is used to combat line echo. Together, the modules provide all the algorithms and state machines needed for a natural-sounding full duplex telephone conversation. The module also supports other standard features, including digital volume control and system diagnostic software, which allow a developer to tune the speakerphone settings for optimal results for a specific set of hardware by simply setting a few parameters in the application software. The echo cancellers can be made to a choice of lengths to accommodate the expected delay (in milliseconds) that the echo to be cancelled will have. Tail lengths from 8ms to 64ms (in increments of 8ms) may be selected by setting an input parameter to the module.

The module also provides a method to add another voice stream received from a separate source, such as the Internet or a soundcard, to the speaker and line, by providing digital switches that can be controlled by the application writer and user of the speakerphone.

Figure 1-2 demonstrates the three port functionality of the Full Duplex Speakerphone Library. Switches D1-D6 are set using the *teldefs_sControl* structure elements. This functionality makes it easy to implement various signal routing scenarios between the Full Duplex Speakerphone and an outside speech source such as a Voice over Internet Protocol - IP stream or samples from a soundcard, including the ability for a three-way conference call.

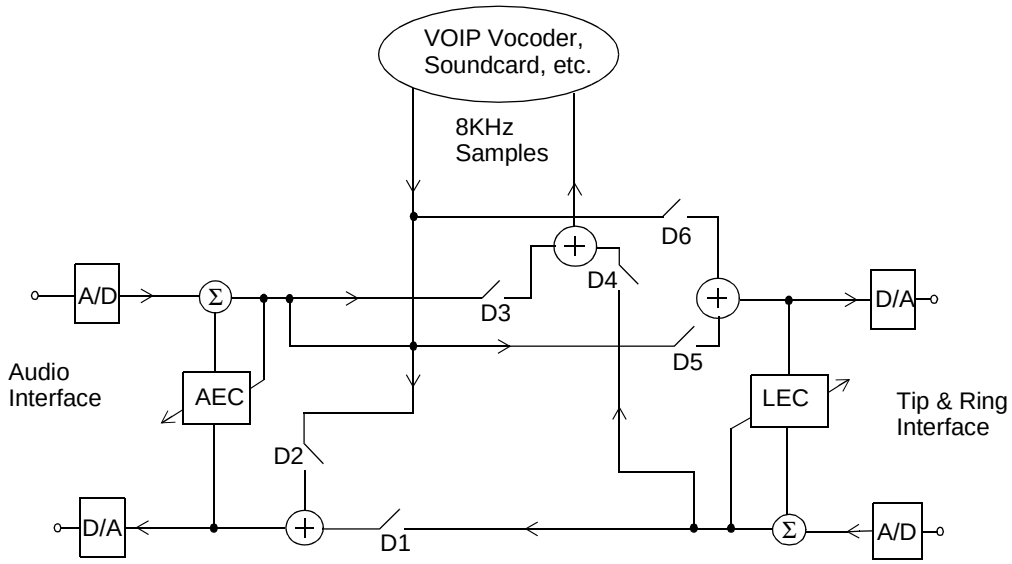

Figure 1-2. The Digital Switches

There are two versions of code; choosing which to use depends on whether the data structure used by the library is placed in internal memory or external memory.

Chapter 2
Directory Structure

Note: "DSP568xx" refers to the specific device for which you're developing, as shown in Chapter 1, "Introduction." .

2.1 Required Core Directories

Figure 2-1 details required platform directories:

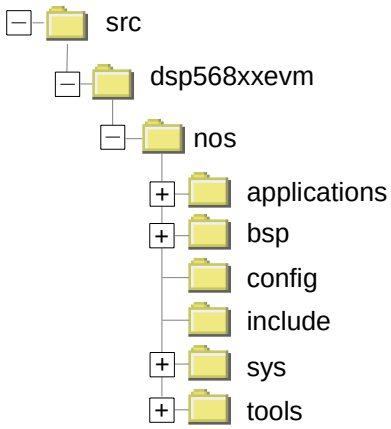

Figure 2-1. Core Directories

As shown in **Figure 2-1**, DSP56858EVM has a *no operating system support (nos)* directory, which includes the following core directories:

- ***applications*** contains applications software that can be exercised on this platform
- ***bsp*** contains board support package specific for this platform
- ***config*** contains default hardware and software configurations for this platform
- ***include*** contains SDK header files which define the Application Programming Interface
- ***sys*** contains required system components
- ***tools*** contains utilities used by system components

There are also optional directories that include domain-specific libraries.

2.2 Optional (Domain-Specific) Directories

Figure 2-2 demonstrates how the Full Duplex Speakerphone Library directory, *fdspk*, is encapsulated in the domain-specific *telephony* directory.

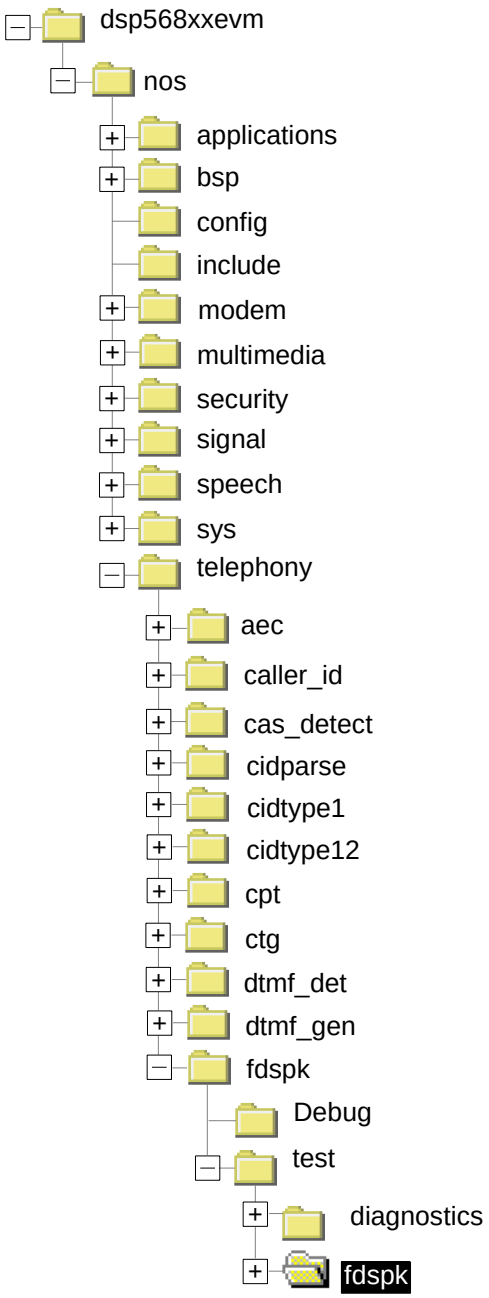

Figure 2-2. *fdspk* Directory

The *telephony* directory contains specific telephony modules, including vocoder algorithms such as G.728, or G.726, as well as the Full Duplex Speakerphone Library, the Caller ID Libraries, and others.

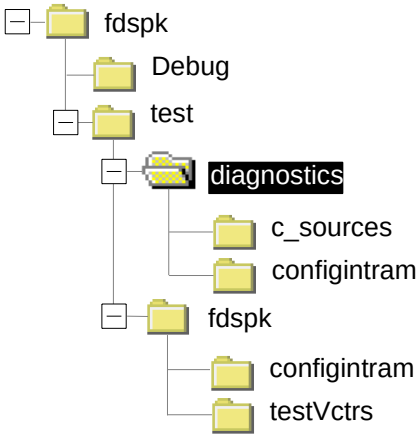

Figure 2-3. *fdspk* Directory Structure

The Full Duplex Speakerphone Library directory, *fdspk*, is shown in **Figure 2-3**. It includes:

- ***fdspk*** contains the Full Duplex Speakerphone Library, header files, and the pre-built library
  - ***Debug*** contains the *gec.lib* library which can be included in application projects that will use the procedures provided by this module
  - ***test*** contains a test project which uses the library. It includes input and output test vectors that can be used to verify that the library is functioning correctly. It also contains a diagonstic example application.
  - ***diagnostics*** contains an example application demonstrating how diagnostics for the speakerphone should be conducted. These diagnostics must be performed before the speakerphone is used in normal mode to obtain the worst-case performance characteristics of the acoustic system that is being used as the speakerphone. The plastics, packaging, and any analog microphone or speaker gains are among the characteristics diagnosed to determine how they affect the acoustic echo.
    - ***c_sources*** contains C source code for the diagnostics application
    - ***configintram*** contains the *linker.cmd* file that uses only internal memory for the diagnostics project. It also contains *appconfig.h* and *appconfig.c*, which specify the necessary library options for this project.
  - ***fdspk*** contains an example application demonstrating how diagnostics for the speakerphone should be used.
    - contains C source code for the test application
    - ***configintram*** contains the *linker.cmd* file that uses only internal memory for the entire project. It also contains *appconfig.h* and *appconfig.c*, which override the SDK's *config.h*.
    - ***testVctrs*** contains header files with test vectors for different tail lengths. They are used by the test application to validate the library code for different tail lengths.

Chapter 3
Full Duplex Speakerphone Library Interfaces

The Full Duplex Speakerphone Library is defined as:

fdspk.lib

The service, interface, and function calls included are described in this chapter.

3.1 Full Duplex Speakerphone Library Interface Services

The Full Duplex Speakerphone Library interface provides procedures to diagnose and use the speakerphone. The speakerphone requires the use of the Generic Echo Canceller Library procedures for the line-side echo cancellation. If the speakerphone only is being used, it processes each sample with a usual sample rate of 8000 samples/sec. The diagnostic procedures need be run only once for each acoustic system being used for the speakerphone. The diagnostic procedures obtain key parameter values used by the speakerphone during regular use, which characterize the acoustic qualities of the speakerphone. Once the parameters are determined and are inserted in the application that uses the speakerphone, the Full Duplex Speakerphone Library is ready to use, as long as the acoustic set-up does not change.

3.2 Interface

The Full Duplex Speakerphone Library is written entirely in the ASM assembly language. The library requires declarations of the following structure *typedefs* defined in *teldefs.h* for each instance, as well as a handle or pointer of type *fdspk_sData** defined in *fdspk.h*:

```
teldefs_sControl
teldefs_sSamples
```

It is also important to remember that this library works with the Generic Echo Canceller Library, which also requires structures *teldefs_sControl* and *teldefs_sSamples* in addition to a pointer of type *gec_sData** defined in *gec.h.*

The Generic Echo Canceller Library is discussed in detail in the SDK manual **Generic Echo Canceller Library**. The Generic Echo Canceller Library will be discussed in this document only to the extent necessary to understand how it works with the Full Duplex Speakerphone Library.

The *teldefs_sControl* structure *typedef* consists of all control variables used by the Full Duplex Speakerphone Library. This structure contains parameters that must be set by the user, which then tune the speakerphone. It contains state variables to be set by the application and when the state of the phone changes. It also contains variables that are used for communication between the Full Duplex Speakerphone Library and the Generic Echo Canceller Library modules

The *teldefs_sSamples* structure *typedef* consists of speech samples that are obtained from the codec drivers used by the application and those that are to be sent out to the codec drivers as speech. It also consists of two sample buffers, representing intermediate processing stages, to be used only by the modules themselves. There are six buffers: two are he inputs/outputs from/to line and audio; two are intermediate speech sample buffers; two are samples that are inputs/outputs from a third source of speech. Refer to **Code Example 3-1** for its definition.

The handle to the structure *fdspk_sData* is to be declared by the application and memory space is allocated by the *fdspkCreate()* function, which is used by the Full Duplex Speakerphone Library functions from that point on. It is essential that no other function, or the application itself, accesses the memory space allocated for this structure. The Full Duplex Speakerphone Library stores values that it needs to reuse, so tampering with the contents of the structure will cause the function to report erroneous results.

The acoustic echo canceller also requires a circular buffer of a size that is dependent on the length of the echo canceller (refer to **Table 3-7**). Before the module's initialization code is called, the pointer to this buffer must be given to the *teldefs_sContro*l instance in the *paecCircularBuffer* variable. All of this is done by the *fdspkCreate()* function before the *fdspkStateInit()* function is called.

To ensure the data is using the entire internal memory to reduce the MIPS of this library, be sure the application has allocated sufficient dynamic internal memory through its *linker.cmd* file.

See header file *fdspk.h*, detailed in **Code Example 3-2** for a description of the use of the function prototypes and structure instances by the Full Duplex Speakerphone Library.

Code Example 3-3 and **Code Example 3-4** contain sample applications exercising the Full Duplex Speakerphone Library. These examples show how to write the declarations mentioned, the assignments, the header files, and how to use the diagnostic procedures and speakerphone procedures under regular mode, as well as how to test the speakerphone procedures. Additional information is located in Chapter 8 of the **Targeting Motorola DSP5685x Platform** manual for the device you're implementing.

Function prototypes and structures for the Full Duplex Speakerphone Library are defined in header files *fdspk.h* and *teldefs.h*, detailed in **Code Example 3-1** and **Code Example 3-2**. Some of the variables in *teldefs.h* are also used in other libraries of the SDK, such as the **Type 1 & 2 Telephony Features Library**, the T**ype 1 & 2 Telephony Parser Library**, and the **Generic Echo Canceller Library**. For details about these libraries, refer to the SDK manuals.

Code Example 3-1. *teldefs.h* - Reference Definition for the Full Duplex Speakerphone Library

```
/*************************************************
Header File: teldefs.h
*************************************************/

#ifndef __TELDEFS_H
#define __TELDEFS_H
```

```
typedef struct teldefs_tsControl
{
      // phone state.
      int    hookSwitch;                    // on hook or off hook
      int    handsFreeLayer1;               // indicates speakerphone ON/OFF

      // ---------- Caller ID ---------------------------------
      // caller ID related variables are declared here.

      // ---------- Generic Echo Canceller ----------------
      // Generic Echo Canceller
      // related variables are
      // declared here.

// ---------- Full Duplex Speaker phone -------------
      int*   paecCircularBuffer;       // pointer to modulo buffer
      int    aecLengthIndex;           // choice of tail length for user to tune.
      int    erlFactor;                // required for Voice activity detection.
      int    silenceOffset;            // power of Silence.
      int    maxVolumeGaindB;          // maximum volume in dB allowed.
      int    volumeGaindB;             // current volume reqd in dB.
      int    totalSupression;          // linear amount of supression.
      int    totalSupressiondB;        // supression in dB.

      int    digitalSwitch1;           // digital switch 1.
      int    digitalSwitch2;           // digital switch 2.
      int    digitalSwitch3;           // digital switch 3.
      int    digitalSwitch4;           // digital switch 4.
      int    digitalSwitch5;           // digital switch 5.
      int    digitalSwitch6;           // digital switch 6.

} teldefs_sControl;

typedef struct teldefs_tsSamples
{
      int    line[5];                  // line input or audio output.
      int    audio[5];                 // audio input or line output.
      int    gec[5];                   // generic echo canceller's output.
      int    aec[5];                   // fdspk's output.
      int    voipinput[5];             // third port's input.
      int    voipoutput[5];            // third port's output.
      int    leccid[5];                // used by type2 to receive line
                                       // echo cancelled samples.

} teldefs_sSamples;

#endif
```

Code Example 3-2 contains *fdspk.h*. The structure *fdspk_sData typedef* is not shown in this example because it contains only memory space to be used by the module.

Code Example 3-2. *fdspk.h* - Reference Definition

```
/************************************************
Header File: fdspk.h
************************************************/

#ifndef __FDSPK_H
#define __FDSPK_H

/**************************
 Foundational Include Files
**************************/
#include "fdspkf.h"
#include "diag.h"

#ifndef __TELDEFS_H
#include "teldefs.h"
#endif

/**************************
 Function Prototypes
**************************/
extern void fdspkStateInit(fdspk_sData*,teldefs_sControl*);
extern void fdspkState(fdspk_sData*,teldefs_sControl*,teldefs_sSamples*);
extern fdspk_sData* fdspkCreate(teldefs_sControl* pCtrl);
extern int fdspkDestroy(fdspk_sData*,pfdspkData,teldefs_sControl* pCtrl);
extern void diagnosticsInit(diag_sData*,diag_sControl*);
extern void diagnostics(diag_sData*,diag_sControl*,diag_sSamples*);

/**************************
 Structures that must be
 defined before calling the
 functions. Examples shown.
**************************/
/*
teldefs_sControl    line1Control;       // their addresses sent
teldefs_sSamples    line1Samples;       // as arguments for the
fdspk_sData*        pfdspk1Data;        // functions listed above.
*/

/*Definition of fdspk_sData is listed in fdspkf.h which contains
various space allocators and housekeeping variables. However,
since these variables are not to be manipulated by the application
or other modules, the contents are not listed here. */

#endif
```

3.2.1 Variable Definition

This section contains a more detailed explanation of the variables in the structure *teldefs_sControl* that an application using the Full Duplex Speakerphone Library must set.

hookSwitch — Set by the application to indicate to the Generic Echo Canceller (*gec*) module whether the phone is ON- or OFF-hook. For ON-hook, it is set to 0; for OFF-hook, it is set to 1. This variable must be set each time the application detects or applies a physical change in or to the hookswitch state. The initial state also should be specified, implying that there should never be an unknown state.

handsFreeLayer1 — Set by the application to indicate to the *gec* module whether the speakerphone is currently ON or OFF. It is required by the *gec* module to switch between the speakerphone mode and regular mode. For speakerphone ON, this is set to 1; for speakerphone OFF, it is set to 0. This variable must be set each time the application detects or applies a physical change in or to the speakerphone state. The initial state should also be specified, implying that there should never be an unknown state.

paecCircularBuffer — Set by the *fdspkCreate()* function to provide the address of the modulo buffer for the echo canceller's data buffer. Via *malloc*, the function allocates a buffer aligned on an n-bit boundary which depends on the tail length that has been chosen for the echo canceller. This variable must be set before accessing any of the module's API functions.

aecLengthIndex — Set by the application to indicate the desired tail length for the acoustic echo canceller. The indices, the corresponding size of the buffer that must be allocated and the boundary on which it must be aligned are all shown in **Table 3-7**. This variable must be set before accessing any of the module's API functions.

erlFactor — A key parameter to be determined with the diagnostics API used by the Voice Activity Detection algorithm. Details of computing this parameter are discussed in **Section 3.3**.

silenceOffset — An optional parameter to be filled by the application user. It is an absolute power estimate of silence. If the noise level in an environment is exceptionally high, it could prove useful to fill this value with a desired number. Default = 200.

maxVolumeGaindB — The maximum digital volume applied to the speaker in decibels, used by the application for a specific acoustic set-up. The module allows a range of -24dB to 24dB. The application is expected to choose a range smaller than or equal to this range and to state the maximum digital volume it intends to use.

volumeGaindB — The current digital volume in decibels, to be applied to the samples that are to be sent out to the audio side (speaker); can be changed at any time.

totalSupression — A parameter to be computed by running the diagnostics API indicating the required maximum total suppression needed by a specific speaker-microphone set-up.

totalSupressiondB — A parameter to be computed by running the diagnostics API indicating the required maximum total suppression, in decibels, needed by a specific speaker-microphone set-up.

digitalSwitch1 — Switch to set the path between the line-in samples through the Full Duplex Speakerphone to audio-out samples. Refer to **Figure 1-2** for a diagram of all switches. ON = 1; OFF = 2.

digitalSwitch2 Switch to set the path between the audio-in samples through the Full Duplex Speakerphone to the third port's out samples; ON = 1; OFF = 2

digitalSwitch3 Switch to set the path between the third-port's in samples through the Full Duplex Speakerphone to audio-out samples; ON = 1; OFF = 2

digitalSwitch4 Switch to set the path between the line-in samples through the Full Duplex Speakerphone to third port's out samples; ON = 1; OFF = 2

digitalSwitch5 Switch to set the path between the audio-in samples through the Full Duplex Speakerphone to line-out samples; ON = 1; OFF = 2

digitalSwitch6 Switch to set the path between the third-port's in samples through the Full Duplex Speakerphone to line-out samples; ON = 1; OFF = 2

3.3 Specifications

The following sections describe the Full Duplex Speakerphone Library functions.

Function arguments for each routine are described *as in, out, or inout.* An *in* argument means that parameter value is an input only to the function. An *out* argument means that the parameter value is an output only from the functions. An *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Typically, *inout* parameters are input pointer variables in which the caller passes the address of a preallocated data structure to a function. The function stores its results within that data structure. The actual value of the *inout* pointer parameter is not changed.

3.3.1 *fdspkCreate*

Call(s):

```
fdspk_sData* fdspkCreate(teldefs_sControl* pControl);
```

Required Headers: *teldefs.h, fdspk.h*

Arguments:

Table 3-1. *fdspkCreate* Arguments

pControl	*inout*	Points to the structure where input and output control information is stored

Description: This function allocates data memory for the data structure and modulo buffer for the Full Duplex Speakerphone Library, then calls the *fdspkStateInit()* function. This function must be called once before using the *fdspkState()* function. The prototype of this function is defined in *fdspk.h*. The return value of the function is a pointer to the data structure.

Before calling this function, the following variables in the control structure are to be set by the application **according to requirements** for the length of filter for the echo canceller.

```
line1Control.aecLengthIndex
```

The acoustic echo canceller requires a circular buffer of a size that depends on the length of the filter chosen. The size, circular boundary, index, lengths and tail lengths of the echo canceller are tabulated in **Table 3-7**. Once the application specifies the index, the create function dynamically allocates a circular buffer placed on a boundary specified in Table 3-7. The pointer of the circular buffer must be placed in *paecCircularBuffer*.

The following must be set after performing the diagnostics on the speakerphone system, the packaging, and the placement of the microphone and speaker in relation to one another. A detailed explanation is given with **Code Example 3-2**.

```
spkControl.totalSupression
spkControl.totalSupressiondB
spkControl.erlFactor
spkControl.maxVolumeGaindB
```

3.3.2 *fdspkDestroy*

Call(s):

```
int fdspkDestroy(fdspk_sData* pData, teldefs_sControl* pControl);
```

Required Headers: *teldefs.h, fdspk.h*

Arguments:

Table 3-2. *fdspkDestroy* Arguments

pData	*inout*	Points to the structure where *fdspk* static data is stored
pControl	*inout*	Points to the structure where input and output control information is stored

Description: This function deallocates memory used by the current *fdspk* instance, characterized by the *fdspk_sData* pData* pointer. The prototype of this function is defined in *fdspk.h*.

3.3.3 *fdspkStateInit*

Call(s):

```
void fdspkStateInit(fdspk_sData* pSpkData, teldefs_sControl* pSpkControl);
```

Required Headers: *teldefs.h, fdspk.h*

Arguments:

Table 3-3. *fdspkStateInit* Arguments

pSpkData	*inout*	Points to a structure of type *fdspk_sData,* defined in *fdspk.h*. This pointer to the structure is sent as a parameter for the function to initialize member variables.
pSpkControl	*inout*	Points to a structure of type *teldefs_sControl*, defined in *teldefs.h*. A pointer to the structure is sent as a parameter for the function to determine the state of the phone and to initialize some member variables.

Description: The *fdspkStateInit* function performs an initialization of all necessary variables in both structures. It should be called once at the beginning of the application.

3.3.4 *fdspkState*

Call(s):

```
void fdspkState(fdspk_sData* pSpkData, teldefs_sControl* pSpkControl,
teldefs_sSamples* pSamples);
```

Required Headers: *teldefs.h*, *fdspk.h*

Arguments:

Table 3-4. *fdspkState* Arguments

pSpkData	*inout*	Points to a structure of type *fdspk_sData,* defined in *fdspk.h*. This is where the procedure stores all static variables it uses. After using the variables appropriately, it updates some of them based on the new samples that are received.
pSpkControl	*inout*	Points to a structure of type *teldefs_sControl,* defined in *teldefs.h*. A pointer to the structure is sent as a parameter for the function to determine the state of the phone and to use some of the parameters for computations. It fills in variables that will be referred to by the *gecLineEchoCanceller* function.
pSamples	*inout*	Points to a structure of type *teldefs_sSamples,* defined in *teldefs.h*. A pointer to the structure is sent as a parameter for the function to operate on the samples received from the line side and the audio side. It fills the *aec* buffer in the structure to be used by the *gecLineEchoCanceller* function that must be called next. It does not change the values in the line or audio buffers.

Description: This procedure performs the entire operation required for a Full Duplex Speakerphone. It uses the *gecEchoCanceller* function's output for line-side echo cancellation. The function carries out the acoustic echo cancellation and suppression analysis for the entire system. It also performs divergence detection for the acoustic echo canceller. It controls the *gecEchoCanceller* module's state machine to operate correctly under speakerphone conditions. All control communication is done through the control structure. It also uses the *gecEchoCanceller* function's output for Voice Activity Detection.

For the suppression analysis and divergence detection for the acoustic echo canceller, the Full Duplex Speakerphone Library uses default threshold values for parameters that are checked to control the state machine. These thresholds are initialized in the control structure. As there will be variations from speakerphone to speakerphone, it is possible to alter these values to acquire better performance on a particular system. However, this alteration should be done after *fdspkStateInit()* is called and before *fdspkState()* is called. This procedure should be called only when the speakerphone is being used; when the speakerphone is being used, *spkControl.handsFreeLayer1* must be set to 1.

Volume can be controlled in the speakerphone procedure by using *spkControl.volumeGaindB*. This number (*volumeGaindB*) should never exceed *spkControl.maxVolumeGaindB* (chosen from a range of -24dB to 24dB) which was initialized in the application before *fdspkInit()* and is identical to the *spkControl.maxVolumeGaindB* setting used in the diagnostic procedures.

Also, the digital switches must be set to ON or OFF, depending on which paths are to be connected and which paths are to be disconnected; refer to **Figure 1-2**. The switches D1 through D6 correspond to *spkControl.digitalSwitch1* through *spkControl.digitalSwitch6;* "1" indicates the switch is closed and "0"

indicates the switch is open. For a regular speakerphone, switches 1 and 5 should be closed by the application. The application writer should ensure that once the switches are closed (ON), appropriate samples (input) in that path are filled in the samples structure.

Samples for the third port are input to the module on the sample structure using the element *samples.voipinput[5]* and the output stream is given as *samples.voipoutput[5]*.

An example of how to use this function is shown in **Code Example 3-4**.

3.3.5 *fdspkDiagnosticsInit*

Call(s):

```
void fdspkDiagnosticsInit(diag_sData* pDiagData, diag_sControl* pDiagControl);
```

Required Headers: *fdspk.h*

Arguments:

Table 3-5. *fdspkDiagnosticsInit* Arguments

pDiagData	*inout*	Points to a structure of type *diag_sData,* defined in *diag.h*. The structure performs necessary computations for diagnosing the speakerphone. It contains data variables to be used only by the procedures.
pDiagControl	*inout*	Points to a structure of type *diag_sControl,* defined in *diag.h*. The structure is used by the application to set variables and pass on variables, as well as to receive computed results from the diagnostics.

Description: This procedure initializes all variables in the data structure and in the control structure, which are used by the diagnostics procedure. Both of the diagnostics procedures must be used once for each speakerphone set-up. This is necessary to obtain the parameters that must be set in the Full Duplex Speakerphone Library for best possible performance.

3.3.6 *fdspkDiagnostics*

Call(s):

```
void fdspkDiagnostics(diag_sData* pDiagData, diag_sControl* pDiagControl,
                      diag_sSamples* pDiagSamples);
```

Required Headers: *fdspk.h*

Arguments:

Table 3-6. *fdspkDiagnostics* Arguments

pDiagData	*inout*	Points to a structure of type *diag_sData,* defined in *diag.h*. The structure performs necessary computations for diagnosing the speakerphone. It contains data variables to be used by the procedures only.
pDiagControl	*inout*	Points to a structure of type *diag_sControl,* defined in *diag.h*. The structure is used by the application to set variables and pass on variables and to receive computed results from the diagnostics.
PDiagSamples	*inout*	Points to a structure of type *diag_sControl* defined in *diag.h*. The structure is used for the passing of samples. The samples are created by the procedure and the application is required to put them out to the line and audio of the speakerphone.

Description: This procedure diagnoses the speakerphone system. It creates samples (frequency sweep) and receives the samples that return from the external system (echoes), then computes the parameters that are used by the speakerphone procedure. It must be used once for each speakerphone set-up. Some parameters must be set before *fdspkDiagnostics* is called. The *diagControl.volumeGain* variable must be set before the call is made and should be set to the highest volume that will be used by the speakerphone system, within the speakerphone's volume range of -24dB to 24dB. The speakerphone application might choose to set the volume range differently. If so, the maximum desired volume should be set in this diagnostic application.

3.4 Feature Phone Tuning

This section illustrates through examples how the Full Duplex Speakerphone Library is used in products.

The parameters must be tuned to set up a speakerphone. In other words, the Full Duplex Speakerphone Library must have the correct parameters describing the speakerphone system in which it is to be used. These parameters are to be computed for the worst-case conditions on the particular speakerphone system; i.e., for maximum volume applied to speakers, maximum gain on the microphone, worst-case line conditions and any other such conditions that can be expected to affect the speakerphone's echoes on the line side and the audio side. This parameter computation is performed just once for a hardware set-up. If the worst-case placement of microphone and speaker relative to each other, analog speaker volume, analog microphone gain, codec ranges and other such conditions do not change, the parameters need not be recalculated. If the previously mentioned conditions change or are altered, these parameters must be computed for the new speakerphone system.

To compute the parameters, a diagnostic application can be written for a one-time use. It is fairly simple, requiring for this application only the codec drivers for the particular hardware and the user's capability to send commands to the application.

Code Example 3-3 is an example of a diagnostic application. The full version is available in the *diagnostics* directory under the *fdspk* directory. The application has its own codec driver and communication between user and application through the serial port. This application will run on Motorola's DSP56858EVM Board. After running the application and obtaining the results, computations must be completed; formulas for these required calculations follow the example.

Code Example 3-3. Use of *diagnose.c*

```
#include <stdio.h>
#include <stdlib.h>
#include "serial.h"       // serial communications.
#include "codecdrv.h"     // codec driver.
#include "fdspk.h"
#define DLE '$'

int codec_buffer_leftin[5];
int codec_buffer_leftout[5];
int codec_buffer_rightin[5];    // audio in
int codec_buffer_rightout[5];   // audio out

volatile int flag = 0;

int main(void)
{

      struct diag_tsData        diagData;
      struct diag_tsControl     diagControl;
      struct diag_tsSamples     diagSamples;
      int i;
      // initialize the codec and daa
```

```
        codec_init();

        // initialize serial port
        serial_init();

        codec_enable();

        // initialize all process variables for diagnostics
        diagControl.maxVolumeGaindB = 18;

        diagnosticsInit(&diagData,&diagControl);

        while(1) {
                for( i = 0; i < 5 ; i++){
                        codec_buffer_leftout[i]  = diagSamples.audio[i];
                        codec_buffer_rightout[i]  = diagSamples.line[i];
                        diagSamples.line[i] = codec_buffer_leftin[i];
                        diagSamples.audio[i] = codec_buffer_rightin[i];
                }
                diagnostics(&diagData,&diagControl,&diagSamples);

                if(diagControl.flagTestComplete == 1) {       // waiting for completion
                                                              // of test from
                                                              // diagnostics procedure.
                        go_onhook();
                        print_string("For ");
                        print_char(diagControl.descriptor);   // char describing what
                                                              // is being output.
                        print_char(0x0d);
                        print_string("NES ");
                        print_integer(diagControl.nesResult);
                        print_string("FES ");
                        print_integer(diagControl.fesResult);

                        diagControl.flagTestComplete = 0;
                }
                process_host_event();           // process user's commands
                display_update();               // print out application's output
                while(flag == 0) ;
                flag = 0;
        }
        return 0;
}


/*************************************************************************
Name:        process_host_event
Purpose:
Parameters:
Returns:

*************************************************************************/


void process_host_event(void){
```

```
/* Process receive events from host */
/* For now we are simulating events that the HOST would send using */
/* the keyboard. Received HOST events are echoed to the screen. */

      int i;
      char ReadData;
      char ch;
      ssize_t size;

      /* Read one byte via SCIO */
      size = sciRead(SciDevice,&ReadData,1);

      if(size > 0){
      //echo back
            sendSerial(ReadData);
      ch = ReadData;
      if(ch == 0x0d) ch = ox0a;
      rx_byte = ch;
      rx_byte = rx_byte & 0x00ff;

      if(rx_byte == '?'){
            first_byte_found = 0;
      }
      else if(first_byte_found == 0){
            first_byte_found = 1;
            first_byte = rx_byte;
            host_byte_cntr = 1;
            command_string_indx = 0;
      }
      else{
            if(first_byte == DLE){
                  if(rx_byte == 'S'){
                        go_offhook();
                        diagControl.hookSwitch = 1;
                        diagControl.startTest = 1; // start the test
                        print_char(0x0d);
                  }
                  if(rx_byte == 'A'){
                        diagControl.testSide = 1; // chosen near side
                        print_char('O');
                        print_char('K');
                        print_char(0x0d);
                  }
                  if(rx_byte == 'B'){
                        diagControl.testSide = 2;
                        print_char('O');
                        print_char('K');
                        print_char(0x0d);
                  }
                  if(rx_byte == 'N'){
                        diagControl.testSide = 3;
                        print_char('O');
                        print_char('K');
                        print_char(0x0d);
```

```
                    }
                    if(rx_byte == 'F'){
                          diagControl.testSide = 4;
                          print_char('O');
                          print_char('K');
                          print_char(0x0d);
                    }
                    first_byte_found = 0;
                    } /* if(first_byte == DLE) */
                    else first_byte_found = 0;
              } /* if(first_byte_found == 0) */

      } /* if(size > 0) */

}
```


Once running, the application waits for the user to start the test; in **Code Example 3-3**, triggered by `$S`. Before opting for `$S`, it is important for the user to choose a side to be tested. In this application, `$A` sends out samples from the near end, analyzing the line echo. In the application, `$B` sends out samples from the far end, analyzing the acoustic echo. The command `$N` causes the diagnostic procedure to compute the power of silence on the near-end side. This is the optional default value and is used in the application, but could prove useful if the noise floor of the input sample is considerably high. The command `$F` causes the diagnostic procedure to compute the power of silence on the far-end side. This process is optional. After choosing a side, typing `$S` starts the test. The diagnostics procedure runs for approximately four seconds before it outputs results. The results may be found in the structure *diagControl* under the variables *descriptor*, *fesResult*, and *nesResult*. The *descriptor* is one of the following:

"A" = Result of analyzing Line Echo

"B" = Result of analyzing Acoustic Echo

"N" = Silence power on the near-end side

"F" = Silence power on the far-end side

It should be noted that during the computation of "B" and "N", there should be as much silence as possible, for any external source of sound may affect the analysis. Since it is the echo that is of interest, rather than direct speech, it is necessary that there be no external source of sound that could be assumed as echo by the procedure.

Values of "A" and "B" must be computed outside by the application writer. The formulas for all computations, in the order required, follow. The formulas are straight-forward mathematical formulas that many languages provide; if desired, a simple application could be written in Matlab or C to output the results.

$$A = 20\log\left(\frac{NESA}{FESA}\right) \quad \textit{Eqn. 3-1}$$

For the `$A` test:

NESA = *nesResult*

FESA = *fesResult*

$$B = 20\log\left(\frac{NESB}{FESB}\right) \quad \textit{Eqn. 3-2}$$

For the`$B` test:

NESB = *nesResult*

FESB = *fesResult*

$$\text{If } (B \geq A)$$
$$TOTAL\_SUPPRESSION\_DB = (B - A) + 3 \quad \textit{Eqn. 3-3}$$

$$\text{If } (B < A)$$
$$TOTAL\_SUPPRESSION\_DB = 0 \quad \textit{Eqn. 3-4}$$

$$B' = B - TOTAL\_SUPPRESSION\_DB \quad \textit{Eqn. 3-5}$$

$$MIDPOINT\_DB = \frac{A + B'}{2} \quad \textit{Eqn. 3-6}$$

$$MIDPOINT = 32768 \times 10^{\left[\frac{MIDPOINT\_DB}{20}\right]} \quad \textit{Eqn. 3-7}$$

$$TOTAL\_SUPPRESSION\_LINEAR = 32768 \times 10^{\left[\frac{-TOTAL\_SUPPRESSION\_DB}{20}\right]} \quad \textit{Eqn. 3-8}$$

The MIDPOINT, TOTAL_SUPPRESSION_LINEAR, and TOTAL_SUPPRESSION_DB are used in the Full Duplex Speakerphone Library application. The appropriate control structure variables, *erlFactor*, *totalSupression*, and *totalSupressiondB*, are to be initialized to the values obtained from the previous equiations before calling the *fdspkStateInit* routine. The numbers obtained from "N" and "F" can be used if there is a lot of noise; if the noise occurs at the microphone input, use "N" and if at the line side, use "F".

Running *diagnostic.mcp*:

1. Configure a HyperTerminal with the following settings: 38400 8N1, Set - "Append line feeds to incoming line ends"
2. Connect HyperTerminal
3. Run the diagnostic.mcp application
4. At the HyperTerminal, type: NS <cr>.
   - Look for a response on the HyperTerminal, such as NES00388; this is the noise level
5. Run the diagnostic.mcp application again
6. At the HyperTerminal, type: BS <cr>
   - A range of increasing frequencies will be heard coming out of the speaker
   - The result will be two numbers; the second number will always be 03405
7. For optimum performance, the first number should be close to 5256 and is added to the NES level obtained from Step 4; the result should be close to 5644
   - To change these numbers, adjust the volume level on the speaker and repeat Steps 3 through 6

Code Example 3-4 contains an example of an application *main* function that uses the Full Duplex Speakerphone Library functions, which required that the Generic Echo Canceller functions be used. **Code Example 3-4** shows how they are used and what initialization is required.

Code Example 3-4. *spkphone.c*

```
#include "teldefs.h"      /* telephony structures */
#include "gec.h"          /* generic echo canceller */
#include "fdspk.h"        /* speakerphone */

/* declarations of the structures */
struct fdspk_tsData* pfdspk1Data;
struct gec_tsData* pgec1Data;
struct teldefs_tsControl line1Control;
struct teldefs_tsSamples line1Samples;

void main(void)
{
      int i;

      // initialize values required before calling fdspkCreate()
      line1Control.aecLengthIndex = 2;       // choice of tail length 24ms.

      // the following are obtained from diagnostics
```

```
line1Control.totalSupression = 4125;    /* total suppression linear */
line1Control.totalSupressiondB = 18;    /* total suppression dB */
line1Control.erlFactor = 3784;          /* midpoint linear */
line1Control.maxVolumeGaindB = 18;      /* chosen max volume - should be
                                        the same as in diagnostics */
pfdspk1Data = fdspkCreate(&line1Control);

// initialize values required before calling gecEchoCancellerCreate().
line1Control.gecEchoCancellerCreate(&line1Control);

// optional parameter setting - can be done only after create() finishes.
line1Control.thresh6 = 200;             /* from N */
line1Control.silenceOffset = 363;       /* from F */

// specify intial state of phone device.
line1Control.hookSwitch = 0;            // ON hook.
line1Control.handsFreeLayer1 = 0;       // speakerphone OFF

line1Control.volumeGaindB = 12;

/* Regular speakerphone mode functionality - no third port */
line1Control.digitalSwitch1 = 1;
line1Control.digitalSwitch2 = 0;
line1Control.digitalSwitch3 = 0;
line1Control.digitalSwitch4 = 0;
line1Control.digitalSwitch5 = 1;
line1Control.digitalSwitch6 = 0;

while(1) {
      for( i = 0; i < 5 ; i++){
            Line1Samples.audio[i] = stream1[i];     // reference signal
            Line1Samples.line[i] = stream2[i];      // echo+independent input
      }

      for(i=0;i<5;i++){
            // call gecEchoCanceller only if off hook
            // call fdspkState only if offhook and speakerphone on.
            if(line1Control.hookSwitch = 1){
                  if(line1Control.handsFreeLayer1 = 1)
                        fdspkState(pfdspk1Data,&line1Control,&line1Samples);
                  gecEchoCanceller(&gec1Data,&line1Control,&line1Samples);
            }

            // if speakerphone is on, the line samples will modify to
            // echo cancelled samples. the audio samples will be the
            // same and the echo cancelled samples will be in line1Samples.
            // aec[] - to be used by application as required.
      }

      // monitor hook switch and speakerphone state change
      // and change line1Control.hookSwitch and
      // line1Control.handsFreeLayer1 accordingly.
```

```
            // monitor third port requests and change
            // line1Control.digitalSwitch's accordingly.
            // monitor volume requests and change
            // line1Control.volumeGaindB

            // get 5 new samples in stream1[] and stream2[]
        }

}
```

NOTE: When the samples are output they must be swapped, meaning that line samples must go out to the audio and the audio samples must go out to the line. This is true of the diagnostics application as well.

See **Table 3-7** for details on the possible lengths and corresponding tail lengths for the acoustic echo canceller filter plus the requirements for circular buffer size and on what boundary they should be placed.

Table 3-7. Acoustic Echo Canceller Filter Lengths

Index	Filter Length	Tail Length	Circular Buffer Size	Boundary
0	64	8ms	72	0x80
1	128	16ms	144	0x100
2	192	24ms	216	0x100
3	256	32ms	288	0x200
4	320	40ms	340	0x200
5	384	48ms	408	0x200
6	448	56ms	476	0x200
7	512	64ms	544	0x400

Providing for a circular buffer for the Full Duplex Speakerphone Library is discussed in **Chapter 5**.

Chapter 4
Building the Full Duplex Speakerphone Library

4.1 Building the Full Duplex Speakerphone Library

The Full Duplex Speakerphone Library combines all of the components described in the previous section into one library: *fdspk.lib*. The library is pre-built for the user, so a project for building the library is not provided. The library is located in the ...*telephony**fdspk**lib* directory of the SDK directory structure.

Figure 4-1 shows an example of how to build a project using the library. This project (shown below) is located in the */nos/applications/telephony/fdspk* directory where other applications are also built in their respective folders under */applications*.

.

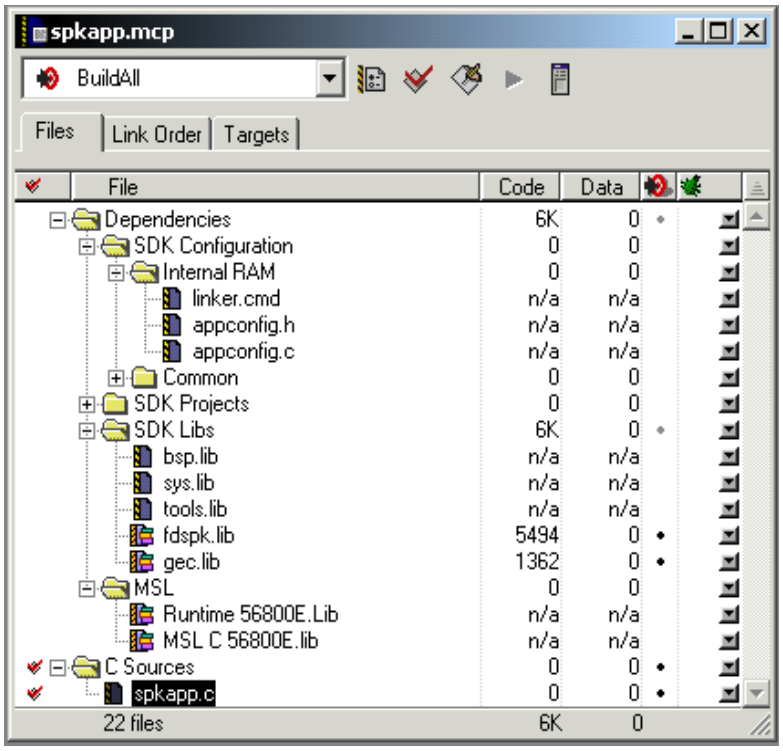

Figure 4-1. Example of an *fdspk* Library Link to a Project

Chapter 5
Linking Applications with the Full Duplex Speakerphone Library

5.1 Full Duplex Speakerphone Library

The Full Duplex Speakerphone Library consists of a complete implementation of a hands-free, full duplex telephone with suppression analysis and an acoustic echo canceller. It provides an API for both using the speakerphone algorithm and for conducting diagnostics on specific hardware. The implementation is re-entrant, and for every instance of run, structures containing static variables for the speakerphone, *teldefs_sControl*, *fdspk_sData*, and *teldefs_sSamples*, must be instantiated. The following APIs provide the interface between the user application and the Full Duplex Speakerphone Library:

- *fdspkCreate(....)*
- *fdspkDestroy(....)*
- *fdspkStateInit(....)*
- *fdspkState(....)*
- *fdspkDiagnosticsInit(....)*
- *fdspkDiagnostics(....)*

5.1.1 Library Sections

For a single instance, the data memory requirements for the Full Duplex Speakerphone Library are:

- 603 words for the *fdspk_sData* structure
- A circular buffer of a size that depends on the choice of filter length (see **Table 3-7**)

The data could be placed dynamically in internal or external memory, affecting only the MIPS performance of the module and no other functionality.

For all instances, the program memory requirements for the Full Duplex Speakerphone Library are:

- 2203 words for the entire module

The Full Duplex Speakerphone Library must be linked into the application by placing *fdspk.text* in the appropriate memory section; (internal memory offers better performance). **Code Example 5-1** is an example of a linker file which demonstrates internal memory usage and linking of *fdspk.text*.

Code Example 5-1. Example of a *linker.cmd* File for the Full Duplex Speakerphone Library

```
#*********************************************************************
#
#            Linker.cmd file for DSP56858 External RAM
#             using only external program and data memory.
#
#*********************************************************************

MEMORY{

       .pInterruptVector   (RWX):ORIGIN

-------------------------------------------------------
MEMORY {

       .pInterruptVector   (RWX):ORIGIN = 0x000000, LENGTH = 0x00008C
       .pIntRAM            (RWX):ORIGIN = 0x00008C, LENGTH = 0x009F74
       .pExtRAM            (RWX):ORIGIN = 0x00A000, LENGTH = 0x1E6000

       .pIntROM            (RX):ORIGIN = 0x1F0000, LENGTH = 0x000400
       .xIntRAM            (RW):ORIGIN = 0x000000, LENGTH = 0x005000
       .xIntRAM_DynamicMem (RW):ORIGIN = 0x005000, LENGTH = 0x001000

       .xStack             (RW):ORIGIN = 0x006000, LENGTH = 0x000800
       .xExtRAM_DynamicMem (RW):ORIGIN = 0x006800, LENGTH = 0x001000
       .xExtRAM            (RW):ORIGIN = 0x000000, LENGTH = 0x005000

       .xPeripherals       (RW):ORIGIN = 0x1FFC00, LENGTH = 0x000400
       .xExtRAM2           (RW):ORIGIN = 0x200000, LENGTH = 0xDFFF00

       .xCoreRegisters     (RW):ORIGIN = 0xFFFF00, LENGTH = 0x000100
}

#*********************************************************************

FORCE_ACTIVE {FconfigInterruptVector}

#*********************************************************************

SECTIONS {

       #*****************************************************************

       .ApplicationInterruptVector:
              {
                     vector.c (.text)

              }>.pInterruptVector

       #*****************************************************************

              .ApplicationCode:
              {
       # Place all code into Program RAM

                     *(.text)
                     *(rtlib.text)
                     *(fp_engine.text)
                     *(user.text)
```

```
                # Place all data into Program RAM

                F_Pdata_start_addr_in_ROM = 0;
                F_Pdata_start_addr_in_RAM = .;
                 pramdata.c (.data)
                F_Pdata_ROMtoRAM_length = 0;

        F_Pbss_start_addr = .;

                 _P_BSS_ADDR = .;
                 pramdata.c (.bss)

                F_Pbss_length = .-_P_BSS_ADDR;

        }>.pExtRAM

#*********************************************************************

        .FDSPKLibraryCode:
        {
# Place fdspk and gec code into Program Internal RAM

                *(fdspk.text)
                *(gec.text)

        }>.pIntRAM

#*********************************************************************

        .ApplicationData:
        {
                # Define variables for C initialization code

                F_Xdata_start_addr_in_ROM =.;
                F_StackAddr         = ADDR(.xStack);
                F_StackEndAddr      = ADDR(.xStack) + SIZEOF(.xStack) - 1;
                F_Xdata_start_addr_in_RAM =.;

                # Define variables for SDK mem library

                # Data (X) memory Layout

                      _EX_BIT = 0;

                 # Internal Memory Partitions (for mem.h partitions)

                      _NUM_IM_PARTITIONS = 0; # IM_ADDR_1 (no IM_ADDR_2)

                 # External Memory Partition (for mem.h partitions)

                      _NUM_EM_PARTITIONS = 1; # EM_ADDR_1

                 FmemEXbit =.;
                      WRITEH(_EX_BIT);
                 FmemNumIMpartitions =.;
                      WRITEH(_NUM_IM_PARTITIONS);
                 FmemNumEMpartitions =.;
                      WRITEH(_NUM_EM_PARTITIONS);
                 FmemIMpartitionList =.;
                      WRITEH(ADDR(.xIntRAM_DynamicMem)*1);
                      WRITEH(SIZEOF(.xIntRAM_DynamicMem)*1);
                 FmemEMpartitionList =.;
                      WRITEH(ADDR(.xExtRAM_DynamicMem)*1);
                      WRITEH(SIZEOF(.xExtRAM_DynamicMem)*1);

                # Add rest of the data into External RAM
```

```
                        *(.const.data)
                        *(.data)
                        *(fp_state.data)
                        *(rtlib.data)

                       F_Xdata_ROMtoRAM_length = 0;

                       F_Xbss_start_addr =.;
                       _X_BSS_ADDR =.;

                        *(rtlib.bss.lo)
                        *(rtlib.bss)
                        *(.bss)

                       F_Xbss_length =.-_X_BSS_ADDR; # Copy DATA

       }>.xExtRAM

#*****************************************************************

       FArchIO             =0x0000;
       FArchCore           =ADDR(.xCoreRegisters);
       FArchInterrupts     =ADDR(.pInterruptVector);

}
```

As mentioned previously, the Full Duplex Speakerphone Library requires a circular buffer (also called a modulo buffer) of a size that depends on the length of the filter chosen by the application (see **Table 3-7**). The buffer and the data structure (*fdspk_sData*) are dynamically allocated by the *fdspkCreate()* function. The application should ensure sufficient data memory for dynamic allocation in either internal memory (preferred for lesser MIPS) or external memory through the linker command file's *xIntRAM_DynamicMem* or *xExtRAM_DynamicMem*.

Chapter 6
Full Duplex Speakerphone Library Applications

6.1 Test and Demo Applications

To verify the Full Duplex Speakerphone algorithm, test and demo applications have been developed. Refer to the **Targeting Motorola DSP568xx Platform Manual** for the DSP you are using to see if the test and demo applications are available for your target.

Chapter 7
License

7.1 Limited Use License Agreement

This software is available under a separate license agreement from Motorola Incorporated. Licensing information can be obtained from your Motorola sales representative or authorized distributor.

For additional product information, see **http://www.motorola.com/semiconductors/**.

Freescale Semiconductor, Inc.

ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

Index

Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

O

P

S

T

V

Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

How to reach us:
USA/EUROPE/Locations Not Listed: Motorola Literature Distribution; P.O. Box 5405, Denver, Colorado 80217. 1–303–675–2140 or 1–800–441–2447

JAPAN: Motorola Japan Ltd.; SPS, Technical Information Center, 3–20–1, Minami–Azabu. Minato–ku, Tokyo 106–8573 Japan. 81–3–3440–3569

ASIA/PACIFIC: Motorola Semiconductors H.K. Ltd.; Silicon Harbour Centre, 2 Dai King Street, Tai Po Industrial Estate, Tai Po, N.T., Hong Kong. 852–26668334

Technical Information Center: 1–800–521–6274

HOME PAGE: http://www.motorola.com/semiconductors/

SDK141/D