



Freescale Semiconductor, Inc.

ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005



MOTOROLA

Freescale Semiconductor, Inc.

ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

Embedded SDK (Software Development Kit)

Targeting Motorola DSP56852 Platform

SDK131/D
Rev. 4, 11/08/2002





Freescale Semiconductor, Inc.

ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

Freescale Semiconductor, Inc.

ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

**For More Information On This Product,
Go to: www.freescale.com**

Because of an order from the United States International Trade Commission, BGA-packaged product lines and part numbers indicated here currently are not available from Freescale for import or sale in the United States prior to September 2010: DSP56852VFE

Contents

About This Book

Audience	xxix
Organization	xxix
Suggested Reading	xxix
Conventions	xxx
References.	xxxii

Chapter 1

Introduction

1.1 Before You Start	1-1
1.2 Quick Start	1-1
1.2.1 Install CodeWarrior Development Tools	1-1
1.2.2 Install DSP56852EVM Hardware	1-2
1.2.3 Install SDK	1-2
1.2.4 Build Released Software.	1-3

Chapter 2

Directory Structure

2.1 DSP56852 SDK Directory Structure	2-1

Chapter 3

Target Configuration for DSP56852

3.1 SDK Configuration	3-1
3.1.1 3DES	3-1
3.1.2 AEC	3-2
3.1.3 AGC	3-2
3.1.4 BSP	3-2
3.1.5 BUTTON	3-3
3.1.6 CALLER_ID.	3-3
3.1.7 CAS_DETECT	3-3
3.1.8 CODEC.	3-4
3.1.9 COP.	3-4
3.1.10 CPT.	3-4
3.1.11 CTG	3-5
3.1.12 CYCLE_COUNT	3-5
3.1.13 DES.	3-5
3.1.14 DSPFUNC	3-6
3.1.15 DTMF_DET	3-6



3.1.16	DTMF_GEN	3-7
3.1.17	FILEIO	3-7
3.1.18	G.165	3-7
3.1.19	G.168	3-8
3.1.20	G.711	3-8
3.1.21	G.726	3-8
3.1.22	GPIO	3-9
3.1.23	IO	3-9
3.1.24	ITCN	3-9
3.1.25	Keypad	3-10
3.1.26	LCD	3-10
3.1.27	LED	3-10
3.1.28	MEMORY	3-11
3.1.29	Noise Suppression	3-11
3.1.30	PC Master Software	3-11
3.1.31	PCS	3-12
3.1.32	PLL	3-12
3.1.33	QUAD_TIMER	3-12
3.1.34	RSA	3-13
3.1.35	SCI	3-13
3.1.36	SERIAL_DATA_FLASH	3-13
3.1.37	SIM	3-14
3.1.38	SPI	3-14
3.1.39	SSI	3-14
3.1.40	STACK_CHECK	3-15
3.1.41	TDC1	3-15
3.1.42	TIMER	3-15
3.1.43	UCOS	3-16
3.1.44	VAD	3-16
3.1.45	V.8bis	3-16
3.1.46	V.21	3-17
3.1.47	V.22bis	3-17
3.1.48	V.42bis	3-17
3.1.49	VR Lite-1	3-18
3.2	Memory Configuration	3-19
3.2.1	External Memory Operation	3-19
3.2.1.1	Linker Command File	3-19
3.2.1.2	Linking: Set Up Memory Partitions	3-22
3.2.1.3	Programming Target Memory	3-22
3.2.2	Internal Memory Operation	3-23
3.2.2.1	Linker Command File	3-24
3.2.2.2	Linking: Set Up Memory Partitions	3-26
3.2.2.3	Programming Target Memory	3-26
3.3	DSP56852 Boot Sequence Configuration	3-28
3.3.1	First Stage Bootloader	3-29
3.3.1.1	Boot from Byte-Wide External Memory (Mode 0)	3-29
3.3.1.2	Boot from SPI (Mode 1)	3-30

3.3.1.3	Boot from Normal Expanded Mode (Mode 2)	3-32
3.3.1.4	Boot from Development Expanded Mode (Mode 3)	3-32
3.3.1.5	Boot from SCI (Mode 6).	3-32
3.3.1.6	Unused Boot Modes (Modes 4, 5, and 7)	3-33
3.3.2	Second Stage Bootloader	3-34
3.3.2.1	Second Stage SPI Bootloader	3-34
3.3.2.1.1	Second Stage SPI Bootloader Demo	3-36
3.3.3	User Application	3-38
3.3.4	DSP56852EVM Serial Data Flash Programmer.	3-38
3.3.4.1	Specification	3-38
3.4	Clock (PLL) Configuration.	3-40

Chapter 4 Interrupt Processing for the DSP56852

4.1	Interrupt Dispatcher	4-1
4.1.1	Super Fast Interrupt Dispatcher	4-1
4.1.2	Fast Interrupt Dispatcher	4-1
4.1.3	Normal Interrupt Dispatcher.	4-1
4.1.4	SDK ISR Overhead.	4-2
4.1.5	SDK ISR Stack Requirements	4-2
4.2	Nested Interrupts.	4-3
4.3	Interrupt Priorities.	4-3
4.4	Device Drivers	4-3
4.4.1	Button Driver	4-3
4.4.2	Codec Driver.	4-3
4.4.3	GPIO Driver	4-3
4.4.4	LED Driver	4-4
4.4.5	Quad Timer Driver	4-4
4.4.6	SCI Driver.	4-4
4.4.7	Serial Data Flash Driver	4-4
4.4.8	SPI Driver	4-4
4.4.9	SSI Driver	4-5
4.4.10	Timer Driver	4-5

Chapter 5 On-Chip Drivers

5.1	Computer Operating Property (COP) Driver	5-2
5.1.1	Initialization	5-2
5.1.2	API Definition.	5-4
5.1.2.1	copInitialize()	5-5
5.1.2.2	copReload().	5-6
5.1.2.3	copForceReset()	5-7
5.1.2.4	copIsReset()	5-8
5.1.3	COP Driver Tests	5-12

5.1.4	COP Driver Demos	5-12
5.1.4.1	COP Demo	5-12
5.1.4.1.1	Set-up for COP Demo	5-12
5.2	GPIO Driver	5-14
5.2.1	Introduction	5-14
5.2.2	Initialization	5-14
5.2.3	API Definition	5-14
5.2.4	GPIO Device-Independent I/O API Specifications	5-16
5.2.4.1	open	5-17
5.2.4.2	close	5-18
5.2.4.3	ioctl	5-19
5.2.5	GPIO Low-Level I/O API Specifications	5-21
5.2.5.1	gpioOpen	5-22
5.2.5.2	gpioClose	5-23
5.2.5.3	gpioIoctl	5-24
5.2.6	GPIO Driver Application	5-25
5.3	Programmable Chip Select (PCS) Driver	5-26
5.3.1	Initialization	5-26
5.3.2	API Definition	5-28
5.3.2.1	pcsInit()	5-29
5.3.2.2	pcsInitEMI()	5-33
5.4	Quad Timer Driver	5-35
5.4.1	Introduction	5-35
5.4.2	Initialization	5-35
5.4.3	API Definition	5-36
5.4.4	Quad Timer Device-Independent I/O API Specifications	5-42
5.4.4.1	open	5-43
5.4.4.2	close	5-44
5.4.4.3	ioctl	5-45
5.4.5	Quad Timer Low-Level I/O API Specifications	5-49
5.4.5.1	qtOpen	5-50
5.4.5.2	qtClose	5-51
5.4.5.3	qtIoctl	5-52
5.4.6	Quad Timer Driver Tests	5-53
5.4.7	Quad Timer Driver Demos	5-53
5.4.7.1	Quad Timer Flashing LEDs Demo	5-53
5.4.7.1.1	Set-up for Quad Timer Flashing LEDs Demo	5-53
5.4.7.1.2	Procedure for Quad Timer Flashing LEDs Demo	5-53
5.5	SCI Driver	5-53
5.5.1	Introduction	5-53
5.5.2	Initialization	5-53
5.5.3	API Definition	5-56
5.5.4	SCI Device Independent I/O API Specifications	5-57
5.5.4.1	open	5-58
5.5.4.2	write	5-59
5.5.4.3	read	5-60
5.5.4.4	close	5-61
5.5.4.5	ioctl	5-62



5.5.5	SCI Low-Level I/O Specifications	5-65
5.5.5.1	sciOpen	5-66
5.5.5.2	sciWrite	5-67
5.5.5.3	sciRead	5-68
5.5.5.4	sciClose	5-69
5.5.5.5	sciIoctl	5-70
5.5.6	SCI Driver Tests	5-73
5.5.6.1	SCI Loopback Test	5-73
5.5.6.1.1	Set-up for SCI Loopback Test	5-73
5.5.6.1.2	Procedure for SCI Loopback Test	5-73
5.5.7	SCI Driver Demos	5-73
5.5.7.1	SCI Serial Port Demo	5-73
5.5.7.1.1	Set-up for SCI Serial Port Demo	5-73
5.5.7.1.2	Procedure for SCI Serial Port Demo	5-73
5.5.7.2	SCI Echo Demo	5-74
5.5.7.2.1	Set-up for SCI Echo Demo	5-74
5.5.7.2.2	Procedure for SCI Echo Demo	5-74
5.6	SPI Driver	5-74
5.6.1	Introduction	5-74
5.6.2	Initialization	5-74
5.6.3	API Definition	5-75
5.6.4	SPI Device-Independent I/O API Specifications	5-77
5.6.4.1	open	5-78
5.6.4.2	write	5-79
5.6.4.3	read	5-80
5.6.4.4	close	5-81
5.6.4.5	ioctl	5-82
5.6.5	SPI Low-Level I/O API Specification	5-84
5.6.5.1	spiOpen	5-85
5.6.5.2	spiBlockedWrite, spiNonBlockedWrite	5-86
5.6.5.3	spiBlockedRead, spiNonBlockedRead	5-87
5.6.5.4	spiBlockedClose, spiNonBlockedClose	5-88
5.6.5.5	spiIoctl	5-89
5.6.6	SPI Driver Applications	5-91
5.6.7	SPI Driver Tests	5-91
5.6.7.1	SPI Test	5-91
5.6.7.1.1	Set-up for SPI Test	5-91
5.6.7.1.2	Procedure for SPI Test	5-91
5.7	SSI Driver	5-91
5.7.1	Introduction	5-91
5.7.2	Initialization	5-91
5.7.3	API Definition	5-99
5.7.4	SSI Device Independent I/O API Specifications	5-101
5.7.4.1	open	5-102
5.7.4.2	write	5-103
5.7.4.3	read	5-104

5.7.4.4	close	5-105
5.7.4.5	ioctl	5-106
5.7.5	SSI Low-Level I/O Specifications	5-108
5.7.5.1	ssiOpen	5-109
5.7.5.2	ssiWrite	5-110
5.7.5.3	ssiRead	5-111
5.7.5.4	ssiClose	5-112
5.7.5.5	ssiIoctl	5-113
5.7.6	SSI Driver Tests and Demos	5-114
5.8	Timer Driver	5-115
5.8.1	Introduction	5-115
5.8.2	Timer Driver Tests	5-115
5.8.3	Timer Driver Demos	5-115
5.8.3.1	Timer Driver Flashing LEDs Demo	5-115
5.8.3.1.1	Set-up for Timer Driver Flashing LEDs Demo	5-115
5.8.3.1.2	Procedure for Timer Driver Flashing LEDs Demo	5-115

Chapter 6 Off-Chip Drivers

6.1	Button Driver	6-1
6.1.1	Introduction	6-1
6.1.2	Initialization	6-1
6.1.3	API Definition	6-1
6.1.4	Button Device-Independent I/O API Specifications	6-3
6.1.4.1	open	6-4
6.1.4.2	close	6-5
6.1.5	Low-Level I/O Services API Specifications	6-6
6.1.5.1	buttonOpen	6-7
6.1.5.2	buttonClose	6-8
6.1.6	Button Driver Tests	6-9
6.1.6.1	Button Driver Test	6-9
6.1.6.1.1	Set-up for Button Driver Test	6-9
6.1.6.1.2	Procedure for Button Driver Test	6-9
6.1.7	Button Driver Applications	6-9
6.2	Codec Driver	6-10
6.2.1	Introduction	6-10
6.2.2	Initialization	6-11
6.2.3	API Definition	6-16
6.2.4	IO Layer API Specification	6-17
6.2.4.1	open	6-18
6.2.4.2	write	6-19
6.2.4.3	read	6-20
6.2.4.4	close	6-21
6.2.4.5	ioctl	6-22
6.2.5	Low-Level Codec Driver API	6-26
6.2.5.1	codecOpen	6-27
6.2.5.2	codecWrite	6-28



6.2.5.3	codecRead	6-29
6.2.5.4	codecClose	6-30
6.2.5.5	codecIoctl	6-31
6.2.6	Codec Driver Tests	6-35
6.2.7	Codec Driver Applications	6-35
6.2.7.1	Codec Loopback Application	6-35
6.2.7.1.1	Set-up for Codec Loopback Application	6-35
6.2.7.1.2	Procedure for Codec Loopback Application	6-35
6.3	FILE I/O Driver	6-36
6.3.1	Introduction	6-36
6.3.2	Initialization	6-37
6.3.3	API Definition	6-37
6.3.4	API Specifications	6-39
6.3.4.1	open	6-40
6.3.4.2	write	6-41
6.3.4.3	read	6-42
6.3.4.4	close	6-43
6.3.4.5	ioctl	6-44
6.3.5	FILE I/O Driver Application	6-44
6.4	Keypad Driver	6-45
6.4.1	Introduction	6-45
6.4.2	Initialization	6-45
6.4.3	API Definition	6-46
6.4.4	Keypad Device-Independent API Specification	6-49
6.4.4.1	open	6-50
6.4.4.2	ioctl	6-51
6.4.4.3	close	6-52
6.4.5	Keypad Device-Dependent API Specification	6-53
6.4.5.1	keypadOpen	6-54
6.4.5.2	keypadIoctl	6-55
6.4.5.3	keypadClose	6-58
6.4.6	Demo Applications	6-60
6.4.7	Test Applications	6-60
6.4.7.1	Hardware Set-up	6-61
6.4.7.2	Procedure for TDC1 Keypad Test	6-61
6.5	LCD Driver	6-61
6.5.1	Introduction	6-61
6.5.2	Initialization	6-62
6.5.3	API Definition	6-63
6.5.4	LCD Device-Independent API Specification	6-64
6.5.4.1	open	6-65
6.5.4.2	write	6-66
6.5.4.3	ioctl	6-67
6.5.4.4	close	6-68
6.5.5	LCD Device-Dependent API Specification	6-69
6.5.5.1	lcdOpen	6-70
6.5.5.2	lcdWrite	6-71

6.5.5.3	lcdIoctl	6-73
6.5.5.4	lcdClose	6-75
6.5.6	Demo Applications	6-76
6.5.7	Test Applications	6-76
6.5.7.1	Hardware Set-up	6-76
6.5.7.2	Procedure for LCD Test	6-76
6.6	LED Driver	6-77
6.6.1	Introduction	6-77
6.6.2	Initialization	6-77
6.6.3	API Definition	6-78
6.6.4	LED Device-Independent I/O API Specifications	6-79
6.6.4.1	open	6-80
6.6.4.2	close	6-81
6.6.4.3	ioctl	6-82
6.6.5	Low-Level Device Drivers I/O API Specifications	6-84
6.6.5.1	ledOpen	6-85
6.6.5.2	ledClose	6-86
6.6.5.3	ledIoctl	6-87
6.6.6	LED Driver Tests	6-88
6.6.6.1	LED Driver Test	6-88
6.6.6.1.1	Set-up for LED Driver Test	6-88
6.6.6.1.2	Procedure for LED Driver Test	6-88
6.6.7	LED Driver Applications	6-88
6.7	PC Master Software SCI Communication Driver	6-89
6.7.1	Introduction	6-89
6.7.2	Initialization	6-89
6.7.3	API Definitions	6-89
6.7.4	PC Master Software Low-Level I/O API Specification	6-93
6.7.4.1	pcmasterdrvInit - Initialize PC Master Software Communication	6-94
6.7.4.2	pcmasterdrvGetAppCmdSts - Get Application Command Status	6-95
6.7.4.3	pcmasterWriteAppCmdSts - Write Application Command Status	6-96
6.7.4.4	pcmasterdrvIsr - Main Communication Function	6-97
6.7.4.5	pcmasterdrvRecorder - Sample Recorded Variables	6-98
6.7.5	PC Master Software Tests	6-99
6.7.6	PC Master Software Applications	6-99
6.7.6.1	PC Master Software Demo	6-99
6.7.6.1.1	Set-up for PC Master Software Demo	6-99
6.7.6.1.2	Procedure for PC Master Software Demo	6-99
6.8	Serial Data Flash Driver	6-100
6.8.1	Introduction	6-100
6.8.2	Initialization	6-101
6.8.3	API Definition	6-101
6.8.4	Serial Data Flash Device-Independent I/O API Specifications	6-102
6.8.4.1	open	6-103
6.8.4.2	write	6-104
6.8.4.3	read	6-105
6.8.4.4	close	6-106
6.8.4.5	ioctl	6-107



6.8.5	Low-Level Device Drivers I/O API Specifications	6-110
6.8.6	Serial Data Flash Driver Tests	6-110
6.8.6.1	Serial Data Flash Driver Test	6-110
6.8.6.1.1	Set-up for Serial Data Flash Driver Test	6-110
6.8.6.1.2	Procedure for Serial Data Flash Driver Test	6-110
6.8.7	Serial Data Flash Driver Demo	6-110
6.9	TDC1 Driver	6-111
6.9.1	Introduction	6-111
6.9.2	Initialization	6-111
6.9.3	API Definition	6-120
6.9.4	Device-Independent API Specification	6-121
6.9.4.1	open	6-122
6.9.4.2	write	6-123
6.9.4.3	read	6-124
6.9.4.4	close	6-125
6.9.4.5	ioctl	6-126
6.9.5	Device-Dependent TDC1 Driver API	6-132
6.9.5.1	tdc1Open	6-133
6.9.5.2	tdc1Write	6-134
6.9.5.3	tdc1Read	6-135
6.9.5.4	tdc1Close	6-136
6.9.5.5	tdc1ioctl	6-137
6.9.6	TDC1 Driver Tests	6-147
6.9.7	TDC1 Driver Applications	6-147
6.9.7.1	TDC1 Loopback Application	6-147
6.9.7.1.1	Set-up for TDC1 Loopback Application	6-147
6.9.7.1.2	Procedure for TDC1 Loopback Application	6-147

Chapter 7
Libraries

7.1	Signal Processing Libraries	7-2
7.1.1	DSP Function Library	7-2
7.1.2	Performance	7-2
7.1.2.1	Trigonometric Math Function Performance	7-2
7.1.2.2	Signal Processing Function Performance	7-3
7.1.2.2.1	autoCorr	7-3
7.1.2.2.2	cbitrev	7-4
7.1.2.2.3	cfft	7-4
7.1.2.2.4	cifft	7-5
7.1.2.2.5	corr	7-5
7.1.2.2.6	fir	7-6
7.1.2.2.7	iir	7-6
7.1.2.2.8	rfft	7-7
7.1.2.2.9	rifft	7-7
7.2	Modem Libraries	7-8
7.2.1	V.8bis Library	7-8
7.2.1.1	Performance	7-9



7.2.2	V.21 Library	7-10
7.2.2.1	Performance	7-10
7.2.3	V.22bis Library	7-11
7.2.3.1	Performance	7-11
7.2.4	V.42bis Library	7-12
7.2.4.1	Performance	7-13
7.3	Security Libraries	7-13
7.3.1	DES Library	7-14
7.3.1.1	Performance	7-14
7.3.2	Triple DES Library	7-15
7.3.2.1	Performance	7-16
7.3.3	RSA Library	7-17
7.3.3.1	Performance	7-18
7.4	Telephony Libraries	7-19
7.4.1	Acoustic Echo Canceller Library	7-19
7.4.1.1	Performance	7-19
7.4.2	Automatic Gain Control (AGC) Library	7-20
7.4.3	CAS Detect Library	7-22
7.4.3.1	Performance	7-22
7.4.4	Call Progress Tones Detect Library	7-23
7.4.4.1	Performance	7-24
7.4.5	Caller ID Library	7-24
7.4.5.1	Performance	7-25
7.4.6	Common Tone Generation (CTG) Library	7-25
7.4.6.1	Performance	7-25
7.4.7	DTMF Detection Library	7-27
7.4.7.1	Performance	7-27
7.4.8	DTMF Generation Library	7-28
7.4.8.1	Performance	7-28
7.4.9	G.165 Library	7-29
7.4.9.1	Performance	7-29
7.4.10	G.168 Library	7-30
7.4.10.1	Performance	7-30
7.4.11	G.711 Library	7-31
7.4.11.1	Performance	7-31
7.4.12	G.726 Library	7-32
7.4.12.1	Performance	7-32
7.4.13	MFCR2 Detection Library	7-33
7.4.13.1	Performance	7-34
7.4.14	Noise Suppression Library	7-34
7.4.15	Voice Activity Detector Library	7-35
7.4.15.1	Performance	7-35
7.5	Speech Libraries	7-36
7.5.1	VR Lite-1 Library	7-36
7.5.1.1	Performance	7-37
7.6	Feature Phone Libraries	7-38
7.6.1	Generic Echo Canceller Library	7-38
7.6.1.1	Performance	7-38



7.6.2	Type 1 and 2 Telephony Parser Library	7-39
7.6.2.1	Performance	7-39
7.6.3	Full Duplex Speakerphone Library	7-39
7.6.3.1	Performance	7-39
7.6.4	Type 1 Telephony Features Library	7-40
7.6.4.1	Performance	7-40
7.6.5	Type 1 and 2 Telephony Features Library	7-41
7.6.5.1	Performance	7-41

Chapter 8
Library Tests

8.1	System Library	8-1
8.1.1	System Test	8-1
8.1.1.1	Set-up for System Test	8-1
8.1.1.2	Procedure for System Test	8-1
8.2	Tools Library	8-2
8.2.1	FIFO Test	8-2
8.2.1.1	Set-up for FIFO Test	8-2
8.2.1.2	Procedure for FIFO Test	8-2
8.3	DSP Functions Library	8-2
8.3.1	Math Tests	8-2
8.3.1.1	Set-up for Math Test	8-2
8.3.1.2	Procedures for Math Tests	8-3
8.3.2	Filter Test	8-3
8.3.2.1	Set-up for Filter Test	8-3
8.3.2.2	Procedure for Filter Test	8-3
8.3.3	RFFT Test	8-3
8.3.3.1	Set-up for RFFT Test	8-3
8.3.3.2	Procedure for RFFT Test	8-4
8.3.4	CFFT Test	8-4
8.3.4.1	Set-up for CFFT Test	8-4
8.3.4.2	Procedure for CFFT Test	8-4
8.4	Security Libraries	8-4
8.4.1	DES Loopback Test	8-4
8.4.1.1	Set-up for DES Loopback Test	8-5
8.4.1.2	Procedure for DES Loopback Test	8-5
8.4.1.3	Triple DES Loopback Test	8-5
8.4.1.4	Set-up for Triple DES Loopback Test	8-5
8.4.1.5	Procedure for Triple DES Loopback Test	8-5
8.4.1.6	RSA Loopback Test	8-5
8.4.1.7	Set-up for RSA Loopback Test	8-5
8.4.1.8	Procedure for RSA Loopback Test	8-6
8.5	Telephony Libraries	8-6
8.5.1	AEC Test	8-6
8.5.1.1	Set-up for AEC Test	8-6
8.5.1.2	Procedure for AEC Test	8-6



8.5.2	Automatic Gain Control (AGC) Test	8-7
8.5.2.1	Set-up for AGC Test	8-7
8.5.2.2	Procedure for AGC Test	8-7
8.5.3	Caller ID Detection Test	8-8
8.5.3.1	Set-up for Caller ID Detection Test	8-10
8.5.3.2	Procedure for Caller ID Detection Test	8-10
8.5.4	CAS Detection Test	8-10
8.5.4.1	Set-up for CAS Detection Test	8-10
8.5.4.2	Procedure for CAS Detection Test	8-10
8.5.5	CPT Detection Test	8-11
8.5.5.1	Set-up for CPT Detection Test	8-11
8.5.5.2	Procedure for CPT Detection Test	8-11
8.5.6	Common Tone Generation (CTG) Test	8-11
8.5.6.1	Set-up for Common Tone Generation Test	8-11
8.5.6.2	Procedure for Common Tone Generation Test	8-12
8.5.7	DTMF Detection Test	8-12
8.5.7.1	Set-up for DTMF Detection Test	8-12
8.5.7.2	Procedure for DTMF Detection Test	8-13
8.5.8	DTMF Generation Test	8-13
8.5.8.1	Set-up for DTMF Generation Test	8-13
8.5.8.2	Procedure for DTMF Generation Test	8-13
8.5.9	G.165 Test	8-14
8.5.9.1	Set-up for G.165 Test	8-14
8.5.9.2	Procedure for G.165 Test	8-14
8.5.10	G.168 Test	8-15
8.5.10.1	Set-up for G.168 Test	8-15
8.5.10.2	Procedure for G.168 Test	8-15
8.5.11	G.711 Test	8-15
8.5.11.1	Set-up for G.711 Test	8-16
8.5.11.2	Procedure for G.711 Test	8-16
8.5.12	G.726 Test	8-16
8.5.12.1	Set-up for G.726 Test	8-17
8.5.12.2	Procedure for G.726 Test	8-17
8.5.13	MFCR2 Detection Test	8-17
8.5.13.1	Set-up for MFCR2 Detection Test	8-17
8.5.13.2	Procedure for MFCR2 Detection Test	8-17
8.5.14	Noise Suppression (NS) Test	8-18
8.5.14.1	Set-up for NS Test	8-18
8.5.14.2	Procedure for NS Test	8-18
8.5.15	VAD Test	8-18
8.5.15.1	Set-up for VAD Test	8-19
8.5.15.2	Procedure for VAD Test	8-19
8.6	Modem Libraries	8-19
8.6.1	V.21 Loopback Test	8-19
8.6.1.1	Set-up for V.21 Loopback Test	8-20
8.6.1.2	Procedure for V.21 Loopback Test	8-20

8.6.2	V.22bis Loopback Test	8-20
8.6.2.1	Set-up for V.22bis Loopback Test	8-21
8.6.2.2	Procedure for V.22bis Loopback Test	8-21
8.6.3	V.8bis Test	8-22
8.6.3.1	Set-up for V.8bis Test	8-22
8.6.3.2	Procedure for V.8bis Test	8-23
8.7	Speech Libraries	8-24
8.7.1	VR Lite-1 Test	8-24
8.7.1.1	Set-up for VR Lite-1 Test	8-25
8.7.1.2	Procedure for VR Lite-1 Test	8-25
8.8	Feature Phone Libraries	8-25
8.8.1	Generic Echo Canceller Library Application	8-25
8.8.1.1	Speakerphone Application	8-25
8.8.1.2	Test Application	8-25
8.8.2	Type 1 and 2 Telephony Parser Library	8-27
8.8.2.1	Type 1 and 2 Telephony Parser Verification Test	8-28
8.8.2.1.1	Test Set-up and Procedure	8-28
8.8.2.2	Type 1 and 2 Telephony Parser Library Example Application	8-28
8.8.3	Full Duplex Speakerphone Library	8-32
8.8.3.1	Full Duplex Speakerphone Application	8-32
8.8.3.2	Test Application	8-32
8.8.3.3	Diagnostics Application	8-34
8.8.4	Type 1 Telephony Features Library	8-35
8.8.4.1	Type 1 Verification Test	8-35
8.8.4.2	Test Set-up and Procedure	8-35
8.8.4.2.1	Example Application Using cid1.lib	8-35
8.8.5	TYPE 1 and 2 Telephony Features Library	8-39
8.8.5.1	Type 2 Verification Test	8-39
8.8.5.1.1	Test Set-up and Procedure	8-39
8.8.5.2	Example Application Using cid12.lib	8-39

Chapter 9 DSP56852 Applications

9.1	BSP Demos	9-1
9.1.1	Codec Demo	9-1
9.1.2	COP Demo	9-1
9.1.3	PC Master Software Demo	9-1
9.1.4	Quad Timer Demo	9-1
9.1.5	SCI Demo	9-2
9.1.6	SCI Echo Demo	9-2
9.1.7	Timer Demo	9-2
9.2	Security Demos	9-2
9.2.1	DES Demo	9-2
9.2.1.1	DES File I/O demo	9-2
9.2.1.1.1	Set-up for DES File I/O Demo	9-3
9.2.1.1.2	Procedure for DES File I/O Demo	9-3



9.2.1.2	DES Real-time demo	9-3
9.2.1.2.1	Set-up for DES Real-time Demo	9-4
9.2.1.2.2	Procedure for DES Real-time Demo	9-4
9.2.2	Triple DES Demo	9-5
9.2.2.1	Triple DES File I/O demo.	9-5
9.2.2.1.1	Set-up for Triple DES Demo	9-5
9.2.2.1.2	Procedure for Triple DES Demo	9-5
9.2.2.2	Triple DES Real-time demo	9-6
9.2.2.2.1	Set-up for Triple DES Real-time Demo	9-6
9.2.2.2.2	Procedure for Triple DES Real-time Demo	9-6
9.2.3	RSA Demo	9-7
9.2.3.1	Set-up for RSA Demo.	9-7
9.2.3.2	Procedure for RSA Demo	9-8
9.3	Telephony Demos.	9-8
9.3.1	Automatic Gain Control (AGC) Demo.	9-8
9.3.1.1	Set-up for AGC Demo	9-8
9.3.1.2	Procedure for AGC Demo	9-9
9.3.2	CPT Demo	9-9
9.3.2.1	Set-up for CPT Demo	9-9
9.3.2.2	Procedure for CPT Demo	9-10
9.3.3	Common Tone Generation Demo.	9-10
9.3.3.1	Set-Up for CTG Demo	9-11
9.3.4	DTMF Detection Demo	9-11
9.3.4.1	Set-up for DTMF Detection Demo.	9-11
9.3.4.2	Procedure for DTMF Detection Demo	9-12
9.3.5	DTMF Generation Demo	9-12
9.3.5.1	Set-up for DTMF Generation Demo.	9-13
9.3.5.2	Procedure for DTMF Generation Demo.	9-14
9.3.6	Feature Phone Application	9-14
9.3.7	G.165 Demo	9-14
9.3.7.1	Set-up for G.165 Demo.	9-14
9.3.7.2	Procedure for G.165 Demo.	9-15
9.3.8	G.168 Demo	9-15
9.3.8.1	Set-up for G.168 Demo.	9-16
9.3.8.2	Procedure for G.168 Demo.	9-16
9.3.9	G.711 Demo	9-17
9.3.9.1	Set-up for G.711 Demo.	9-17
9.3.9.2	Procedure for G.711 Demo.	9-17
9.3.10	G.726 Demo	9-17
9.3.10.1	Set-up for the G.726 Demo.	9-18
9.3.10.2	Procedure for G.726 Demo.	9-18
9.3.11	Noise Suppression (NS) Demo.	9-18
9.3.11.1	Set-up for NS Demo	9-19
9.3.11.2	Procedure for NS Demo	9-19
9.3.12	VAD Demo.	9-20
9.3.12.1	Set-up for VAD Demo	9-20
9.3.12.2	Procedure for VAD Demo	9-20



- 9.4 Modem Demos 9-21
 - 9.4.1 V.22bis and Demo 9-21
 - 9.4.1.1 V.22bis Demo 9-21
 - 9.4.1.2 Hardware set-up 9-21
 - 9.4.1.3 Answering Modem Demo for 2400bps 9-22
 - 9.4.1.4 Answering Modem Demo for 1200bps 9-23
 - 9.4.1.5 Calling Modem Demo for 2400bps 9-24
 - 9.4.1.6 Calling Modem Demo for 1200bps 9-25
 - 9.4.1.7 Special Considerations During Demo. 9-27
 - 9.4.2 V.42bis Demo 9-27
 - 9.4.2.1 Set-up for V.42bis Demo 9-27
 - 9.4.2.2 Procedure for V.42bis Demo 9-28
- 9.5 Speech Demos. 9-28
 - 9.5.1 VRLite-1 Demo 9-28
 - 9.5.1.1 Set-up for VRLite-1 Demo 9-31
 - 9.5.1.2 Procedure for VRLite-1 Demo 9-33
 - 9.5.1.2.1 Procedure for demo_vrlite1 9-33
 - 9.5.1.2.2 Procedure for demo_dial_vrlite1 9-34
 - 9.5.1.3 Demo Limitations 9-35



Because of an order from the United States International Trade Commission, BGA-packaged product lines and part numbers indicated here currently are not available from Freescale for import or sale in the United States prior to September 2010: DSP56852VFE

List of Tables

Table 3-1	DSP56852 OMR Configuration	3-6
Table 3-2	DSP56852 SR Configuration	3-6
Table 3-3	Format of Data in External Memory.	3-29
Table 3-4	Format of Data in SPI Data Flash.	3-31
Table 3-5	Format of Data for Boot from SCI	3-32
Table 4-1	SDK ISR Overhead for DSP56852 (Running From Internal Memory)	4-2
Table 4-2	SDK ISR Stack Requirements for DSP56852	4-2
Table 5-1	COP Driver Arguments - copInitialize()	5-5
Table 5-2	GPIO Driver Arguments - open	5-17
Table 5-3	GPIO Ports Available for the DSP56852 Chip.	5-17
Table 5-4	GPIO Driver Arguments - close	5-18
Table 5-5	GPIO Driver Arguments - ioctl	5-19
Table 5-6	GPIO ioctl Commands	5-19
Table 5-7	GPIO Driver Arguments - gpioOpen	5-22
Table 5-8	GPIO Driver Arguments - close	5-23
Table 5-10	gpiolock Commands.	5-24
Table 5-9	GPIO Driver Arguments.	5-24
Table 5-11	PCS Argument Values	5-29
Table 5-12	PCS Argument Values	5-33
Table 5-13	Quad Timers	5-36
Table 5-14	Quad Timer Data Structure Members.	5-38
Table 5-15	Quad Driver Arguments - open	5-43
Table 5-16	Quad Timer Driver Arguments - close	5-44
Table 5-17	Quad Timer Driver Arguments - ioctl	5-45
Table 5-18	Quad Timer ioctl / qtioclock Commands	5-45
Table 5-19	Quad Timer Driver Arguments - open	5-50
Table 5-20	Quad Timer Driver Arguments - close	5-51
Table 5-21	Quad Timer Driver Arguments - ioctl	5-52
Table 5-22	SCI Configuration Items for appconfig.h	5-54
Table 5-23	SCI Driver Arguments - open.	5-58
Table 5-24	SCI Devices for DSP56852 Chip	5-58
Table 5-25	SCI Driver Arguments - write	5-59
Table 5-26	SCI Driver Arguments - read	5-60
Table 5-27	SCI Driver Arguments - close	5-61

Table 5-28	SCI Driver Arguments - ioctl	5-62
Table 5-29	SCI ioctl Commands	5-62
Table 5-30	SCI Driver Arguments - sciOpen	5-66
Table 5-31	SCI Devices for DSP56852 Chip	5-66
Table 5-32	SCI Driver Arguments - sciWrite	5-67
Table 5-33	SCI Driver Arguments - sciRead	5-68
Table 5-34	SCI Driver Arguments - sciClose	5-69
Table 5-35	SCI Driver Arguments - sciIoctl	5-70
Table 5-36	SCI sciIoctl Commands	5-70
Table 5-37	SPI Configuration Items for appconfig.h	5-75
Table 5-38	SPI Driver Arguments - open	5-78
Table 5-39	SPI Driver Arguments - write	5-79
Table 5-40	SPI Driver Arguments - read	5-80
Table 5-41	SPI Driver Arguments - close	5-81
Table 5-42	SPI Driver Arguments - ioctl	5-82
Table 5-43	SPI ioctl Commands	5-82
Table 5-44	SPI Driver Arguments - spiOpen	5-85
Table 5-45	SPI Driver Arguments - spiWrite	5-86
Table 5-46	SPI Driver Arguments - spiRead	5-87
Table 5-47	SPI Driver Arguments - spiClose	5-88
Table 5-48	SPI Driver Arguments - spiIoctl	5-89
Table 5-49	SPI spiIoctl Commands	5-89
Table 5-50	SSI Configuration Items for appconfig.h	5-92
Table 5-51	SSI Driver Arguments - open	5-102
Table 5-52	SSI Devices for DSP56852 Chip	5-102
Table 5-53	SSI Driver Arguments - write	5-103
Table 5-54	SSI Driver Arguments - read	5-104
Table 5-55	SSI Driver Arguments - close	5-105
Table 5-56	SSI Driver Arguments - ioctl	5-106
Table 5-57	SSI ioctl Commands	5-106
Table 5-58	SSI Driver Arguments - ssiOpen	5-109
Table 5-59	SSI Devices for DSP56852 Chip	5-109
Table 5-60	SSI Driver Arguments - ssiWrite	5-110
Table 5-61	SSI Driver Arguments - ssiRead	5-111
Table 5-62	SSI Driver Arguments - ssiClose	5-112
Table 5-63	SSI Driver Arguments - ssiIoctl	5-113
Table 5-64	SSI ssiIoctl Commands	5-113
Table 6-1	Button Driver Arguments - open	6-4
Table 6-2	Availability of Button Devices for Specific Platforms	6-4



Table 6-3	Button Driver Arguments - close	6-5
Table 6-4	Button Driver Arguments - buttonOpen	6-7
Table 6-5	Button Driver Arguments - buttonClose	6-8
Table 6-6	Codec Configuration Items for appconfig.h	6-11
Table 6-7	Codec Driver Arguments - open	6-18
Table 6-8	Codec Driver Arguments - write	6-19
Table 6-9	Codec Driver Arguments - read	6-20
Table 6-10	Codec Driver Arguments - close	6-21
Table 6-11	Codec Driver Arguments - ioctl	6-22
Table 6-12	Codec ioctl Commands	6-22
Table 6-13	Codec Driver Arguments - codecOpen	6-27
Table 6-14	Codec Driver Arguments - codecWrite	6-28
Table 6-15	Codec Driver Arguments - codecRead	6-29
Table 6-16	Codec Driver Arguments - codecClose	6-30
Table 6-17	Codec Driver Arguments - codecIoctl	6-31
Table 6-18	Codec codecIoctl Commands	6-31
Table 6-19	File I/O Configuration Items for appconfig.h	6-37
Table 6-20	File I/O Driver Arguments - open	6-40
Table 6-21	File I/O Driver Arguments - write	6-41
Table 6-22	File I/O Driver Arguments - read	6-42
Table 6-23	File I/O Driver Arguments - close	6-43
Table 6-24	KEYPAD Driver Arguments - open	6-50
Table 6-25	Keypad Driver Arguments - ioctl	6-51
Table 6-26	Keypad Driver Arguments - close	6-52
Table 6-27	Keypad Driver Arguments - keypadOpen	6-54
Table 6-28	TDC1 Keypad Driver Arguments - keypadIoctl	6-55
Table 6-29	keypadIoctl Commands	6-55
Table 6-30	TDC1 Keypad Driver Arguments - keypadClose	6-58
Table 6-31	LCD Driver Arguments - open	6-65
Table 6-32	LCD Driver Arguments - write	6-66
Table 6-33	LCD Driver Arguments - ioctl	6-67
Table 6-34	LCD Driver Arguments - close	6-68
Table 6-35	LCD Driver Arguments - lcdOpen	6-70
Table 6-36	LCD Driver Arguments - lcdWrite	6-71
Table 6-37	LCD Driver Arguments - lcdIoctl	6-73
Table 6-38	lcdIoctl Commands	6-73
Table 6-39	LCD Driver Arguments - lcdClose	6-75
Table 6-40	LEDs and Peripheral Multiplexing	6-77
Table 6-41	LED Driver Arguments - open	6-80



Table 6-42	LED Driver Arguments - close.....	6-81
Table 6-43	LED Driver Arguments - ioctl	6-82
Table 6-44	LEDs for Specific EVM Boards (pParams - input parameter).....	6-82
Table 6-45	LED ioctl Commands.....	6-82
Table 6-46	Low-Level LED Driver Arguments - ledOpen.....	6-85
Table 6-47	LED Driver Arguments - ledClose.....	6-86
Table 6-48	Low-Level LED Driver Arguments	6-87
Table 6-49	PC Master Software appconfig.h Settings	6-89
Table 6-50	sPCMasterComm structure.....	6-92
Table 6-51	timeBase format	6-92
Table 6-52	pcmasterdrvInit parameters.....	6-94
Table 6-53	pcmasterdrvWriteAppCmdSts arguments	6-96
Table 6-54	Serial Data Flash Driver Arguments - open	6-103
Table 6-55	Serial Data Flash Driver Arguments - write.....	6-104
Table 6-56	Serial Data Flash Driver Arguments - read	6-105
Table 6-57	Serial Data Flash Driver Arguments - close.....	6-106
Table 6-58	Serial Data Flash Driver Arguments.....	6-107
Table 6-59	Serial Data Flash ioctl Commands	6-107
Table 6-60	TDC1 Configuration Items for appconfig.h	6-112
Table 6-61	TDC1 Driver Arguments - open.....	6-122
Table 6-62	TDC1 Driver Arguments - write.....	6-123
Table 6-63	TDC1 Driver Arguments - read	6-124
Table 6-64	TDC1 Driver Arguments - close.....	6-125
Table 6-65	TDC1 Driver Arguments - ioctl	6-126
Table 6-66	TDC1 Driver Arguments - tdc1Open	6-133
Table 6-67	TDC1 Driver Arguments - tdc1Write.....	6-134
Table 6-68	TDC1 Driver Arguments - tdc1Read	6-135
Table 6-69	TDC1 Driver Arguments - tdc1Close.....	6-136
Table 6-70	TDC1 Driver Arguments - tdc1Ioctl.....	6-137
Table 6-71	TDC1 tdc1Ioctl Commands	6-137
Table 7-1	Trigonometric Math Function Performance	7-2
Table 7-2	autoCorr Performance.....	7-3
Table 7-3	cbitrev Performance	7-4
Table 7-4	cfft Performance	7-4
Table 7-5	cifft Performance	7-5
Table 7-6	corr Performance.....	7-6
Table 7-7	iir Performance	7-7
Table 7-8	rfft Performance	7-7
Table 7-9	rifft Performance.....	7-8

Table 7-10	V.8bis Memory and MIPS Requirements for Internal Memory on the DSP56852	7-9
Table 7-11	MIPS on the DSP56852	7-10
Table 7-12	Frequencies Used in V.21	7-10
Table 7-13	V.21 Memory and MIPS Requirements for Internal Memory on the DSP56852	7-11
Table 7-14	V.22bis Memory and MIPS Requirements for Internal Memory on the DSP56852	7-12
Table 7-15	V.42bis Memory Requirements on the DSP56852	7-13
Table 7-16	V42bis Cycle Count for Internal Memory on the DSP56852	7-13
Table 7-17	DES Memory and MIPS Requirements for Internal Memory on the DSP56852	7-14
Table 7-18	DES Cycle Counts for Internal Memory on the DSP56852	7-15
Table 7-19	3DES Memory and MIPS Requirements for Internal Memory on the DSP56852	7-16
Table 7-20	3DES Cycle Count for Internal Memory on the DSP56852	7-17
Table 7-21	RSA Memory and MIPS Requirements for Internal Memory on the DSP56852	7-18
Table 7-22	RSA Cycle Count for Internal Memory on the DSP56852	7-18
Table 7-23	AEC Memory and MIPS Requirements for Internal Memory on the DSP56852	7-20
Table 7-24	AGC Memory and MIPS Requirements for Internal Memory on the DSP56852 for Small Memory Model	7-21
Table 7-25	AGC Memory and MIPS Requirements for Internal Memory on the DSP56852 for Large Memory Model	7-21
Table 7-26	CAS Electrical Characteristics	7-22
Table 7-27	CAS Detect Memory and MIPS Requirements for Internal Memory on the DSP56852	7-22
Table 7-28	Tone Specifications	7-23
Table 7-29	CPT Memory and MIPS Requirements for Internal Memory on the DSP56852	7-24
Table 7-30	Caller ID Memory and MIPS Requirements for Internal Memory on the DSP56852	7-25
Table 7-31	CTG Memory and MIPS Requirements for Internal Memory on the DSP56852 for Small Memory Model	7-26
Table 7-32	CTG Memory and MIPS Requirements for Internal Memory on the DSP56852 for Large Memory Model	7-26
Table 7-33	DTMF Detection Memory and MIPS for Internal Memory on the DSP56852	7-27
Table 7-34	MIPS and Memory Requirements for Internal Memory on the DSP56852 for Small Memory Model	7-28



Table 7-35	MIPS and Memory Requirements for Internal Memory on the DSP56852 for Large Memory Model	7-28
Table 7-36	G.165 Memory and MIPS Requirements for Internal Memory on the DSP56852	7-29
Table 7-37	G.168 Memory and MIPS Requirements for Internal Memory on the DSP56852	7-30
Table 7-38	G.711 Memory and MIPS Requirements for Internal Memory on the DSP56852 for Small Memory Model	7-31
Table 7-39	G.711 Memory and MIPS Requirements for Internal Memory on the DSP56852 for Large Memory Model	7-32
Table 7-40	G.726 Memory and MIPS Requirements for Internal Memory on the DSP56852	7-33
Table 7-41	MFCR2 Detection Memory and MIPS for Internal Memory on the DSP56852	7-34
Table 7-42	NS Memory and MIPS Requirements for Internal Memory on the DSP56852	7-34
Table 7-43	VAD Memory and MIPS Requirements for Internal Memory on the DSP56852 for Small Memory Model	7-35
Table 7-44	VAD Memory and MIPS Requirements for Internal Memory on the DSP56852 for Large Memory Model	7-36
Table 7-45	VRLite-1 Memory and MIPS Requirements for Internal Memory on the DSP56852	7-37
Table 7-46	Instruction Cycle Counts for Internal Memory on the DSP56852	7-37
Table 7-47	Generic Echo Canceller Memory and MIPS Requirements	7-38
Table 7-48	Type 1 and 2 Telephony Parser Library Memory and MIPS Requirements	7-39
Table 7-49	Full Duplex Speakerphone Memory and MIPS Requirements	7-40
Table 7-50	Type 1 Telephony Features Library Memory and MIPS Requirements	7-40
Table 7-51	Type 1 and 2 Telephony Features Memory and MIPS Requirements	7-41

List of Figures

Figure 1-1	Launch CodeWarrior Tools	1-3
Figure 1-2	Select File/Open	1-3
Figure 1-3	Select the ...\sdk\src\dsp56852evm\nos Directory	1-4
Figure 1-4	Select buildall.mcp Project	1-4
Figure 1-5	buildall.mcp Project	1-4
Figure 1-6	Execute Make Command	1-5
Figure 2-1	DSP56852 Directories	2-1
Figure 3-1	Set Up DSP56852 Memory Partitions	3-22
Figure 3-2	Programming Target Memory	3-23
Figure 3-3	Set up DSP56852 Memory Partitions	3-26
Figure 3-4	Programming Target Memory	3-27
Figure 3-5	DSP56852 Boot Sequence	3-28
Figure 3-6	DSP56852 Boot Sequence With Second Stage SPI Bootloader	3-35
Figure 3-7	SDK Clock Configuration for the DSP56852	3-40
Figure 6-1	Basic Set-up for File I/O Utilization	6-36
Figure 6-2	PC Master Software Application Window	6-100
Figure 7-1	Triple-DES: (a) Encryption and (b) Decryption	7-16
Figure 8-1	Test Set-up for V.21	8-20
Figure 8-2	V.22bis Set-up for Digital and Analog Loopback Tests	8-21
Figure 8-3	V.8bis Test Set-up	8-23
Figure 8-4	V.8bis Test Set-up with Speakers	8-24
Figure 9-1	DES Demo	9-3
Figure 9-2	DES Real-time Demo Set-up	9-4
Figure 9-3	Triple DES Demo	9-5
Figure 9-4	3DES Demo Set-up	9-6
Figure 9-5	RSA Demo	9-7
Figure 9-6	AGC Demo Set-up	9-8
Figure 9-7	CPT Detection Demo Set-up	9-10
Figure 9-8	DTMF Detection Demo Set-up	9-12
Figure 9-9	DTMF Generation Demo Set-up	9-13
Figure 9-10	G.165 Demo Set-up	9-15
Figure 9-11	G.168 Demo Set-up	9-16
Figure 9-12	G.711 Demo Set-up	9-17



Figure 9-13 G.726 ADPCM Codec Test Set-up..... 9-18

Figure 9-14 NS Demo Set-up..... 9-19

Figure 9-15 V.22bis Demo Set-up..... 9-22

Figure 9-16 V.42bis Demo Set-up..... 9-27

Figure 9-17 State Diagram of VRLite-1 Demo System..... 9-29

Figure 9-18 Flowchart for VRLite-1 Demo System..... 9-30

Figure 9-19 Demo Set-up for demo_vrlite1..... 9-32

Figure 9-20 Demo Set-up for demo_dial_vrlite1..... 9-33

List of Examples

Code Example 3-1	3DES - appconfig.h File	3-1
Code Example 3-2	AEC - appconfig.h File	3-2
Code Example 3-3	AGC - appconfig.h File	3-2
Code Example 3-4	BSP - appconfig.h File	3-2
Code Example 3-5	BUTTON - appconfig.h File	3-3
Code Example 3-6	CALLER_ID - appconfig.h File	3-3
Code Example 3-7	CAS_DETECT - appconfig.h File	3-3
Code Example 3-8	CODEC - appconfig.h File	3-4
Code Example 3-9	COP - appconfig.h File	3-4
Code Example 3-10	CPT - appconfig.h File	3-4
Code Example 3-11	CTG - appconfig.h File	3-5
Code Example 3-12	Cycle Count - appconfig.h File	3-5
Code Example 3-13	DES - appconfig.h File	3-5
Code Example 3-14	DSPFUNC - appconfig.h File	3-6
Code Example 3-15	DTMF_DET - appconfig.h File	3-6
Code Example 3-16	DTMF_GEN - appconfig.h File	3-7
Code Example 3-17	FILEIO - appconfig.h File	3-7
Code Example 3-18	G.165 - appconfig.h File	3-7
Code Example 3-19	G.168 - appconfig.h File	3-8
Code Example 3-20	G.711 - appconfig.h File	3-8
Code Example 3-21	G.726 - appconfig.h File	3-8
Code Example 3-22	GPIO - appconfig.h File	3-9
Code Example 3-23	IO - appconfig.h File	3-9
Code Example 3-24	ITCN - appconfig.h File	3-9
Code Example 3-25	Keypad - appconfig.h File	3-10
Code Example 3-26	LCD - appconfig.h File	3-10
Code Example 3-27	LED - appconfig.h File	3-10
Code Example 3-28	MEMORY - appconfig.h File	3-11
Code Example 3-29	Noise Suppresion - appconfig.h File	3-11
Code Example 3-30	PC Master Software - appconfig.h File	3-11
Code Example 3-31	PCS - appconfig.h File	3-12
Code Example 3-32	PLL - appconfig.h File	3-12
Code Example 3-33	QUAD_TIMER - appconfig.h File	3-12
Code Example 3-34	RSA- appconfig.h File	3-13

Code Example 3-35	SCI - appconfig.h File	3-13
Code Example 3-36	Serial Data Flash - appconfig.h File	3-13
Code Example 3-37	SIM - appconfig.h File	3-14
Code Example 3-38	SPI - appconfig.h File	3-14
Code Example 3-39	SSI - appconfig.h File	3-14
Code Example 3-40	STACK_CHECK - appconfig.h File	3-15
Code Example 3-41	TDC1 - appconfig.h File	3-15
Code Example 3-42	TIMER - appconfig.h File	3-15
Code Example 3-43	UCOS - appconfig.h File	3-16
Code Example 3-44	VAD - appconfig.h File	3-16
Code Example 3-45	V.8BIS - appconfig.h File	3-16
Code Example 3-46	V.21 - appconfig.h File	3-17
Code Example 3-47	V.22BIS - appconfig.h File	3-17
Code Example 3-48	V.42BIS - appconfig.h File	3-17
Code Example 3-49	VR Lite-1- appconfig.h File	3-18
Code Example 3-50	Linker.cmd, External Memory	3-19
Code Example 3-51	Linker.cmd, Internal Memory	3-24
Code Example 3-52	PLL_MUL / appconfig.h File	3-41
Code Example 5-1	copInitialize	5-5
Code Example 5-2	Overwriting Default COP Driver Settings in appconfig.h	5-8
Code Example 5-3	COP Driver API usage	5-9
Code Example 5-4	GPIO Driver Usage.	5-20
Code Example 5-5	GPIO Driver Usage.	5-25
Code Example 5-6	appconfig.h file/Chip Select 3	5-31
Code Example 5-7	main.c file/Chip Select 3.	5-32
Code Example 5-8	appconfig.h file/EMI.	5-34
Code Example 5-9	main.c file/Chip Select 3.	5-34
Code Example 5-10	Timer Configuration	5-35
Code Example 5-11	qt_sState Data Structure	5-40
Code Example 5-12	Quad Timer Driver Usage.	5-46
Code Example 5-13	Low-Level Quad Timer Driver Usage	5-52
Code Example 5-14	SCI Driver Usage	5-63
Code Example 5-15	SCI Driver Usage	5-71
Code Example 5-16	SPI Driver Settings in appconfig.h	5-81
Code Example 5-17	SPI Driver Usage	5-81
Code Example 5-18	SPI Driver Usage	5-83
Code Example 5-19	SPI Driver Usage with ioctl call	5-88
Code Example 5-20	SPI Driver Usage	5-90
Code Example 5-21	SSI Driver Usage	5-106

Code Example 5-22	SSI Driver Usage	5-113
Code Example 6-1	Button Driver Example	6-5
Code Example 6-2	Low Level Button Driver Example	6-8
Code Example 6-3	Sample appconfig.h code	6-15
Code Example 6-4	Codec Driver IO Layer API Example	6-25
Code Example 6-5	Codec Driver Low Level API Example	6-34
Code Example 6-6	Sample appconfig.h code	6-37
Code Example 6-7	File I/O Driver Usage	6-44
Code Example 6-8	Use of open	6-50
Code Example 6-9	Use of ioctl	6-51
Code Example 6-10	Use of close	6-52
Code Example 6-11	Use of keypadOpen	6-54
Code Example 6-12	Use of keypadIoctl	6-57
Code Example 6-13	use of keypadClose	6-58
Code Example 6-14	Use of open	6-65
Code Example 6-15	Use of write	6-66
Code Example 6-16	Use of ioctl	6-67
Code Example 6-17	Use of close	6-68
Code Example 6-18	Use of lcdOpen	6-70
Code Example 6-19	Use of lcdWrite	6-71
Code Example 6-20	Use of lcdIoctl	6-74
Code Example 6-21	use of lcdClose	6-75
Code Example 6-22	LED Driver Usage	6-83
Code Example 6-23	Low-Level LED Driver Usage	6-87
Code Example 6-24	Serial Data Flash Driver Usage	6-108
Code Example 6-25	Sample appconfig.h code	6-119
Code Example 6-26	TDC1 Driver Device-Independent API Example	6-127
Code Example 6-27	TDC1 Driver Low Level API Example	6-141
Code Example 8-1	GEC testapp.c	8-26
Code Example 8-2	Use of CIDMessageParser Interface	8-28
Code Example 8-3	Speakerphone testapp.c	8-32
Code Example 8-4	Use of Type1CID Interface	8-35
Code Example 8-5	Use of Type12CID Interface	8-40



About This Book

This manual describes how to use the Motorola Embedded Software Development Kit (SDK) to develop software for Motorola's DSP56852 processor.

Audience

This document targets software developers using Motorola's Embedded Software Development Kit to develop software for Motorola's DSP56852 processor.

Organization

- **Chapter 1, Introduction**,—provides a brief overview of this document
- **Chapter 2, Directory Structure**,—provides a description of domain-specific directories
- **Chapter 3, Target Configuration for DSP56852**,—explains how to change SDK configurations
- **Chapter 4, Interrupt Processing for the DSP56852**,—describes all of the interrupt processing functions
- **Chapter 5, On-Chip Drivers**,—details the DSP56852 on chip device drivers
- **Chapter 6, Off-Chip Drivers**,—discusses specific evaluation module hardware peripheral drivers included in the DSP56852
- **Chapter 7, Libraries**,—describes all of the SDK libraries that are not part of the core libraries and are part of the SDK DSP56852 release
- **Chapter 8, Library Tests**,—describes the tests available to confirm that SDK libraries are operating properly
- **Chapter 9, DSP56852 Applications**,—details DSP56852 applications

Suggested Reading

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

- *DSP56800E Reference Guide*, DSP56800ERM/D
- *DSP56852 16-Bit Digital Signal Processor User's Manual*, DSP56852UM/AD
- *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
<i>Italic</i>	Directory names, project names, calls, functions, statements, procedures, routines, arguments, file names, applications, variables, directives, code snippets in text	...and contains these core directories: <i>applications</i> contains applications software... ...CodeWarrior project, <i>3des.mcp</i> is... ...the <i>pConfig</i> argument... ...defined in the C header file, <i>aec.h</i> ...
Bold	Reference sources, paths, emphasis	...refer to the Targeting DSP56F80x Platform manual... ...see: C:\Program Files\Motorola\Embedded SDK\help\tutorials
Blue Text	Linkable on-line	...refer to Chapter 7 , License....
Number	Any number is consid- ered a positive value, unless preceded by a minus symbol to signify a negative value	3V -10 DES ⁻¹
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"...

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.



Freescal Semiconductor, Inc.

ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

3DES	Triple Data Encryption Standard
ADPCM	Adaptive Differential Pulse Code Modulation
AEC	Acoustic Echo Canceller
BSP	Board Specific Peripheral
CAS	Customer premises equipment Alerting Signal
CBC	Cipher Clock Chaining Mode
CFB	Cipher Feedback Mode
CFFT	Complex Fast Fourier Transform
CID	Caller ID
CODEC	Coder/Decoder
COP	Computer Operating Properly
CPE	Customer Premises Equipment
CPT	Call Progress Tones
DCE	Data Circuit terminating Equipment
DES	Data Encryption Standard
DSP	Digital Signal Processor or Digital Signal Processing
DTE	Data Terminal Equipment
DTMF	Dual Tone Multiple Frequency
ERLE	Echo Return Loss Enhancement
EVM	Evaluation Module
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
I/O	Input/Output
IDE	Integrated Development Environment
IIR	Infinite Impulse Response
ITCN	Interrupt Controller
LSB	Least Significant Bit
MAC	Multiply/Accumulate
MIPS	Million Instructions Per Second
MSB	Most Significant Bit
NLMS	Normalized Mean Squares
OnCE™	On-Chip Emulation
OMR	Operating Mode Register
PC	Personal Computer
PCM	Pulse Code Modulation
PLL	Phase Lock Loop
PSTN	Public Switched Telephone Network
RFFT	Real Fast Fourier Transform
RSA	Rivest, Shamir and Adleman

SAS	Subscriber Alerting Signal
SCI	Serial Communication Interface
SDK	Software Development Kit
SIM	System Integration Module
SP	Stack Pointer
SPCS	Stored Program Controlled-switching System
SPI	Serial Peripheral Interface
SR	Status Register
SRC	Source
SSI	Synchronous Serial Interface
VAD	Voice Activity Detection
VR	Voice Recognition

References

The following sources were used to produce this book:

1. *DSP56800E Reference Manual*, DSP56800ERM/D
2. *DSP56852 16-Bit Digital Signal Processor User's Manual*, DSP56852UM/AD
3. *Embedded SDK Programmer's Guide*, SDK101/D
4. *Digital Signal Processing (DSP) Function Library*, SDK107/D
5. *V.8bis Library*, SDK120/D
6. *V.22bis Library*, SDK108/D
7. *V.42bis Library*, SDK121/D
8. *Data Encryption Standard (DES) Library*, SDK110/D
9. *Triple Data Encryption Standard (3 DES) Library*, SDK119/D
10. *RSA Library*, SDK128/D
11. *Caller ID Detection Library*, SDK112/D
12. *DTMF Generation Library*, SDK114/D
13. *DTMF Detection Library*, SDK113/D
14. *G.165 Line Echo Canceller Library*, SDK115/D
15. *G.711 Log-PCM Library*, SDK117/D
16. *G.726 Speech Codec Library*, SDK118/D
17. *Customer Premises Equipment Alerting Signal (CAS) Detect Library*, SDK124/D
18. *CPT Detection Library*, SDK123/D
19. *Acoustic Echo Canceller Library*, SDK125/D
20. *Feature Phone Application for the DSP56852*, SDK147/D

Chapter 1

Introduction

This manual describes how to use Motorola's Embedded Software Development Kit (SDK) to develop software for the Motorola DSP56852 processor.

1.1 Before You Start

This manual contains information specific to the SDK only as it applies to Motorola DSP56852 software development. The user must be familiar with the SDK in general, as described in the **SDK Programmer's Guide**, before continuing.

Motorola's Embedded SDK is designed for and fully integrated with Metrowerks CodeWarrior development tools. Before you can start exploring the full feature set of the SDK, you must first install and become familiar with the CodeWarrior development environment.

1.2 Quick Start

This chapter gives you the information you need to install and run the SDK and CodeWarrior.

1.2.1 Install CodeWarrior Development Tools

As previously mentioned, Motorola's Embedded SDK is designed for and fully integrated with Metrowerks CodeWarrior development tools. Using CodeWarrior tools, users can build SDK libraries, applications and other software included as part of the SDK release. Once the software is built, CodeWarrior tools will allow SDK users to download executable images into the target platform and run/debug downloaded code.

To start the installation process, follow these steps:

Step 1.

- Insert the CodeWarrior CD-ROM into your computer's CD-ROM drive
- If Auto Install is disabled on your computer, click the *Start* button, select *Run*, and type the CD-ROM's drive letter and *\Setup.exe* in the *Open: text box*. (e.g. *D:\Setup.exe*)
- Follow the CodeWarrior software installation instructions on your screen

NOTE: After installing CodeWarrior, remember to restart the computer. This restart ensures that newly installed drivers will be available for use.

Step 2. Register CodeWarrior

- Click the *Start* button, then select *CodeWarrior Registration* from the CodeWarrior group. This runs the *MWRegister.exe* program.
- Enter your registration number and contact information. If you do not have a registration number, leave the Registration Number text box blank.
- Click *OK* and send the resulting text file (e.g., *MWRegistration.txt*) to *license@metrowerks.com*. A license key will be sent to you via Email.

Step 3. Install the license key

- Locate the *license.dat* file in the CodeWarrior installation directory and open this file with any standard text editor (for example, Notepad)
- Copy or type the key, starting on a new line at the bottom of the *license.dat* file. For more detailed instructions, see the *License_Install.txt* file in the Licensing directory.
- Save the *license.dat* file

1.2.2 Install DSP56852EVM Hardware

The SDK for DSP56852 has been designed and tested with DSP56852EVM target hardware. If you want to exercise the software applications included with SDK quickly, DSP56852EVM hardware must be installed.

The DSP56852EVM installation information is provided with CodeWarrior installation and can be found in the following document, located in the CodeWarrior installation directory:

<...>\CodeWarrior Manuals\Pdf\Targeting_DSP56800E.pdf

It is recommended that all SDK users read through this document before proceeding with software development.

1.2.3 Install SDK

In order for the SDK to integrate itself with development tools, CodeWarrior tools must be installed prior to the SDK installation (see [Section 1.2.1](#)). If the SDK is installed while CodeWarrior is not present, users can only browse installed SDK software, but will not be able to build, download and run released code.

NOTE: If CodeWarrior is installed after the SDK, the SDK must be removed and reinstalled after CodeWarrior installation, in order for the SDK to function properly. To start the installation process, perform the following steps:

Step 1.

- Execute *Setup.exe* from the CD

Step 2.

- Follow the SDK software installation instructions on your screen

1.2.4 Build Released Software

Once the SDK is installed, users can build all released software for DSP56852 by opening and building the top level project, *buildall.mcp*, using the CodeWarrior development environment.

Step 1: Launch Metrowerks' CodeWarrior IDE, as shown in [Figure 1-1](#).

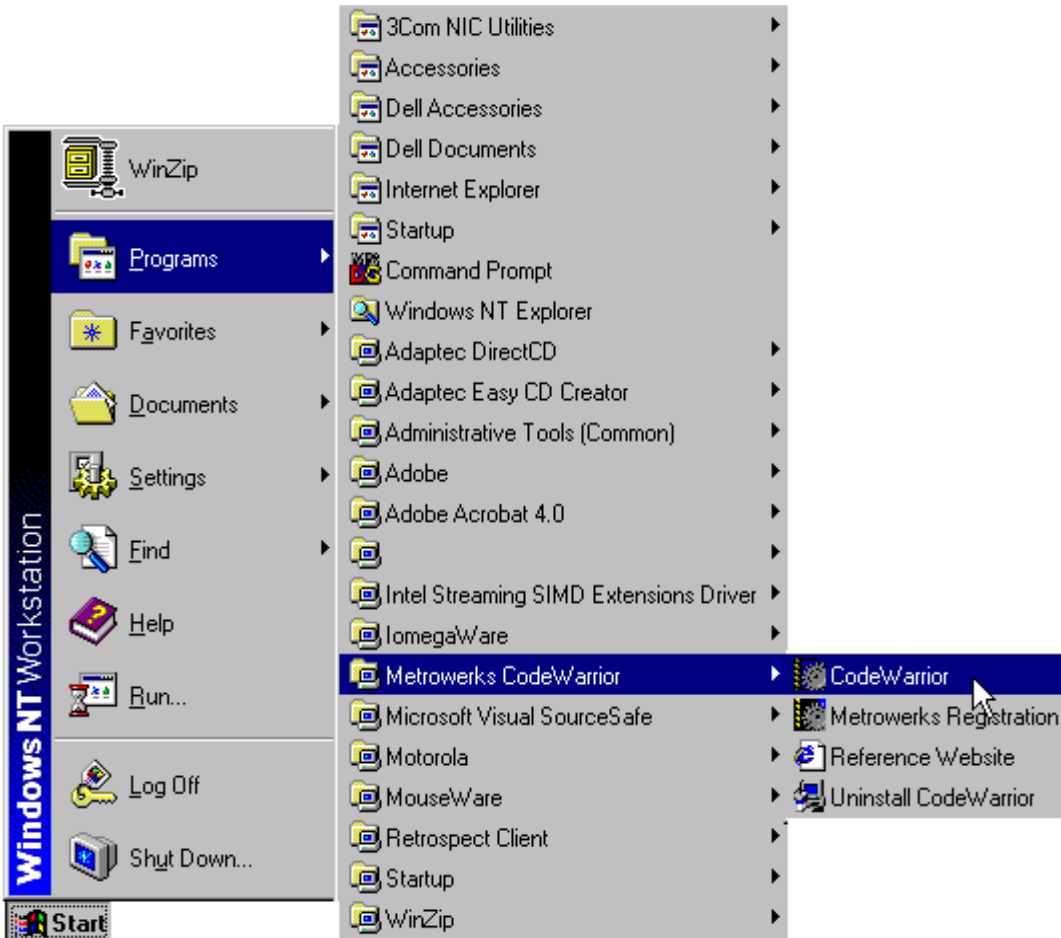


Figure 1-1. Launch CodeWarrior Tools

Step 2: Using the *File/Open* command ([Figure 1-2](#)), open the *buildall.mcp* project by selecting the *...\sdk\src\dsp56852evm\nos* directory ([Figure 1-3](#)), then choosing the *buildall.mcp* folder ([Figure 1-4](#)). The *buildall.mcp* project is shown in [Figure 1-5](#).

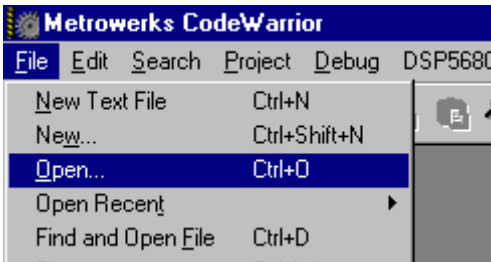


Figure 1-2. Select *File/Open*

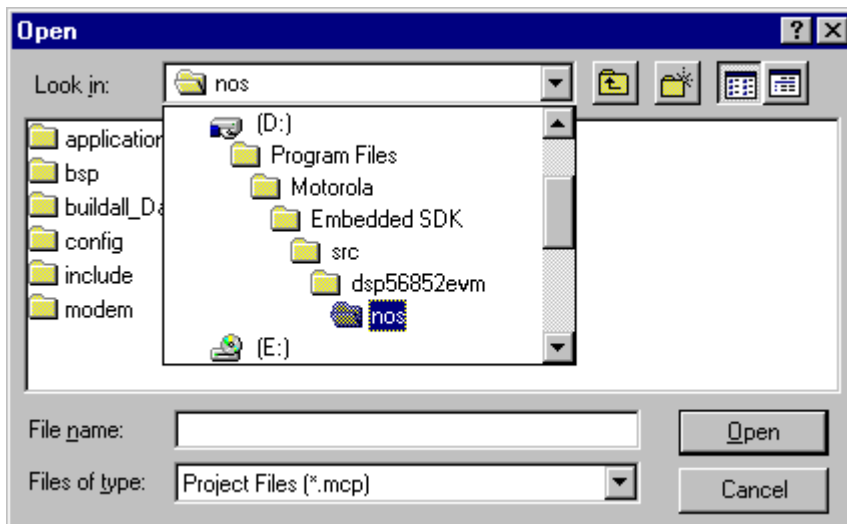


Figure 1-3. Select the ...\\sdk\\src\\dsp56852evm\\nos Directory

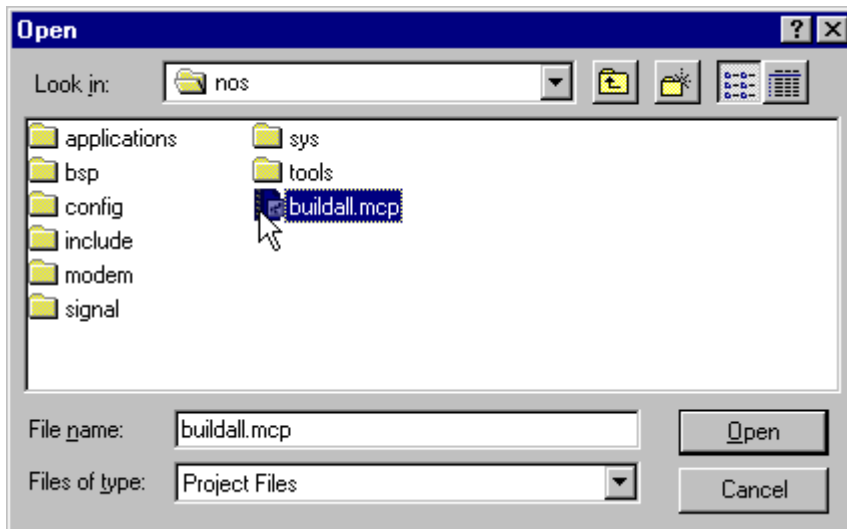


Figure 1-4. Select *buildall.mcp* Project

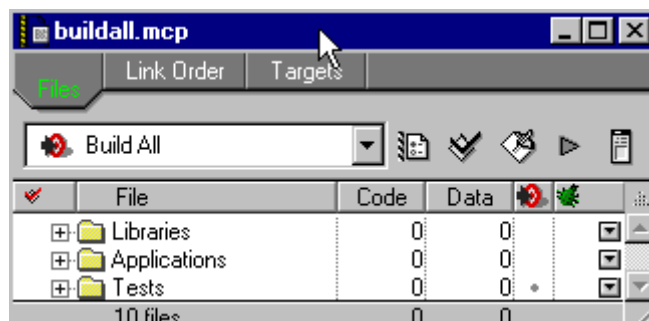


Figure 1-5. *buildall.mcp* Project

Step 3: Execute the build by pressing function key [F7] or by choosing the *Make* command from the menu.

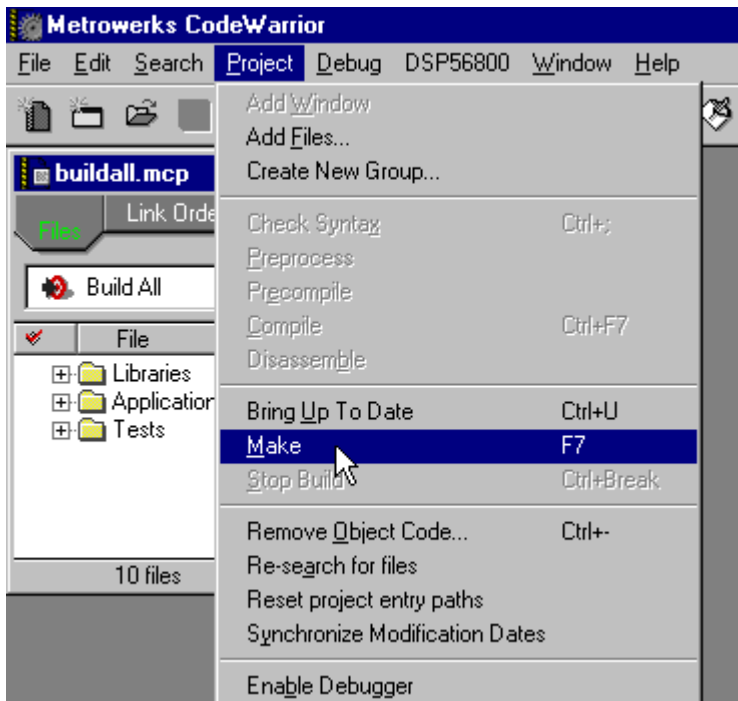


Figure 1-6. Execute *Make* Command

At this point, the *buildall.mcp* project will build every library and application that is part of the SDK DSP56852 release.



Chapter 2

Directory Structure

2.1 DSP56852 SDK Directory Structure

Figure 2-1 illustrates the directory structure for DSP56852EVM.

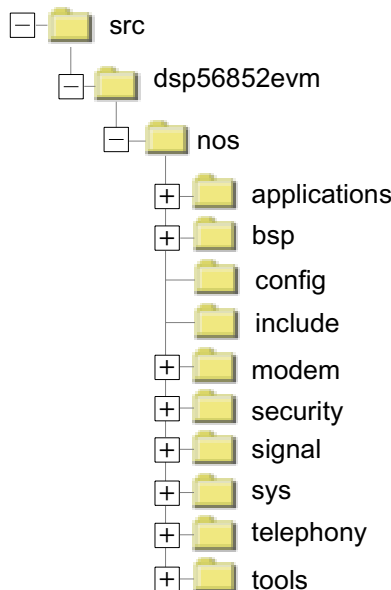


Figure 2-1. DSP56852 Directories

In the SDK DSP56852EVM release, there are several domain-specific directories:

- ***applications*** contains demo projects
- ***bsp*** contains on and off chip drivers
- ***config*** contains SDK configuration files
- ***include*** contains public header files for drivers and libraries
- ***modem*** contains modem-specific libraries
- ***security*** contains security-specific libraries
- ***signal*** contains a variety of signal processing libraries
- ***sys*** contains code SDK services
- ***telephony*** contains telephony-specific libraries
- ***tools*** contains software tools



Chapter 3

Target Configuration for DSP56852

The SDK is a scalable software infrastructure that defines and implements such services as Board Support Package; Interrupt Handling; Memory Management; Input and Output; and Timers. These services are the building blocks of the SDK, and each service has its own configuration and start-up requirements. While the **Embedded SDK Programmer's Guide** covers target configuration in great detail, this section acts as an extension of that manual, focusing on the DSP56852's specifics. The SDK user must be familiar with Target Configuration in general, as described in **Chapter 6** of the **SDK Programmer's Guide**, before continuing.

3.1 SDK Configuration

The *appconfig.h* configuration file contains the following list of SDK components and services which the user can include in the executable program: **3DES**, **AEC**, **BSP**, **BUTTON**, **CALLER_ID**, **CAS_DETECT**, **CODEC**, **COP**, **CPT**, **DES**, **DSPFUNC**, **DTMF_DET**, **DTMF_GEN**, **FILEIO**, **G.165**, **G.168**, **G.711**, **G.726**, **GPIO**, **IO**, **ITCN**, **Keypad**, **LCD**, **LED**, **MEMORY**, **PCS**, **PLL**, **QUAD_TIMER**, **RSA**, **SCI**, **SIM**, **SPI**, **SSI**, **STACK_CHECK**, **TIMER**, **VAD**, **V.8bis**, **V.21**, **V.22bis**, **VR Lite-1**. Each of these components is described below.

3.1.1 3DES

Macro(s):

`INCLUDE_3DES`

Description: When *INCLUDE_3DES* is defined and included in the *appconfig.h* file as shown below, the **Triple DES Library** will be included in the application.

Default: Undefined and not included.

Code Example 3-1. 3DES - *appconfig.h* File

```
#define INCLUDE_3DES
```

3.1.2 AEC

Macro(s):

INCLUDE_AEC

Description: When *INCLUDE_AEC* is defined and included in the *appconfig.h* file as shown below, the **Acoustic Echo Canceller Library** will be included in the application.

Default: Undefined and not included.

Code Example 3-2. AEC - *appconfig.h* File

```
#define INCLUDE_AEC
```

3.1.3 AGC

Macro(s):

INCLUDE_AGC

Description: When *INCLUDE_AGC* is defined and included in the *appconfig.h* file as shown below, the **Automatic Gain Control (AGC) Library** will be included in the application.

Default: Undefined and not included.

Code Example 3-3. AGC - *appconfig.h* File

```
#define INCLUDE_AGC
```

3.1.4 BSP

Macro(s):

INCLUDE_BSP

Description: When *INCLUDE_BSP* is defined and included in the *appconfig.h* file as shown below, the Board Specific Peripheral drivers are included in the application.

Default: Included in *appconfig.h*.

Special Issues: When the BSP component is used, **PLL**, **SIM**, **COP**, and **ITCN** components will automatically be included in the application.

Code Example 3-4. BSP - *appconfig.h* File

```
#define INCLUDE_BSP
```

3.1.5 BUTTON

Macro(s):

INCLUDE_BUTTON

Description: When *INCLUDE_BUTTON* is defined and included in the *appconfig.h* file as shown below, the **Button Driver** will be included in the application.

Default: Undefined and not included.

Special Issues: When the BUTTON component is used, the **TIMER** component will automatically be included in the application. Also, when *INCLUDE_IO* is used, IO service is automatically included in the application to provide device-independent interface for the **Button Driver**.

Code Example 3-5. BUTTON - *appconfig.h* File

```
#define INCLUDE_BUTTON
```

3.1.6 CALLER_ID

Macro(s):

INCLUDE_CALLER_ID

Description: When *INCLUDE_CALLER_ID* is defined and included in the *appconfig.h* file as shown below, the **Caller ID Library** will be included in the application.

Default: Undefined and not included.

Code Example 3-6. CALLER_ID - *appconfig.h* File

```
#define INCLUDE_CALLER_ID
```

3.1.7 CAS_DETECT

Macro(s):

INCLUDE_CAS_DETECT

Description: When *INCLUDE_CAS_DETECT* is defined and included in the *appconfig.h* file as shown below, the **CAS Detect Library** will be included in the application.

Default: Undefined and not included.

Code Example 3-7. CAS_DETECT - *appconfig.h* File

```
#define INCLUDE_CAS_DETECT
```

3.1.8 CODEC

Macro(s):

INCLUDE_CODEC

Description: When *INCLUDE_CODEC* is defined and included in the *appconfig.h* file as shown below, the **Codec Driver** will be included in the application.

Default: Undefined and not included.

Special Issues: When the CODEC component is used, the **SSI** component will automatically be included in the application. Also, when *INCLUDE_IO* is used, IO service is automatically included in the application to provide device-independent interface for the **Codec Driver**.

Code Example 3-8. CODEC - *appconfig.h* File

```
#define INCLUDE_CODEC
```

3.1.9 COP

Macro(s):

INCLUDE_COP

Description: When *INCLUDE_COP* is defined and included in the *appconfig.h* file as shown below, **Computer Operating Property (COP) Driver** will be included in the application.

Default: Undefined and not included.

Special Issues: This component is automatically included if the **BSP** component is defined.

Code Example 3-9. COP - *appconfig.h* File

```
#define INCLUDE_COP
```

3.1.10 CPT

Macro(s):

INCLUDE_CPT

Description: When *INCLUDE_CPT* is defined and included in the *appconfig.h* file as shown below, the **Call Progress Tones Detect Library** will be included in the application.

Default: Undefined and not included.

Code Example 3-10. CPT - *appconfig.h* File

```
#define INCLUDE_CPT
```

3.1.11 CTG

Macro(s):

INCLUDE_CTG

Description: When *INCLUDE_CTG* is defined and included in the *appconfig.h* file as shown below, the **Common Tone Generation (CTG) Library** will be included in the application.

Default: Undefined and not included.

Code Example 3-11. CTG - *appconfig.h* File

```
#define INCLUDE_CTG
```

3.1.12 CYCLE_COUNT

Macro(s):

INCLUDE_CYCLECOUNT

Description: When *INCLUDE_CYCLECOUNT* is defined and included in the *appconfig.h* file as shown below, SDK cycle count services will be included in the application. Refer to the **SDK Programmer's Guide** for more details about this service.

Default: Undefined and not included.

Code Example 3-12. Cycle Count - *appconfig.h* File

```
#define INCLUDE_CYCLECOUNT
```

3.1.13 DES

Macro(s):

INCLUDE_DES

Description: When *INCLUDE_DES* is defined and included in the *appconfig.h* file as shown below, the **DES Library** will be included in the application.

Default: Undefined and not included.

Code Example 3-13. DES - *appconfig.h* File

```
#define INCLUDE_DES
```

3.1.14 DSPFUNC

Macro(s):

INCLUDE_DSPFUNC

Description: When *INCLUDE_DSPFUNC* is defined and included in the *appconfig.h* file as shown below, the [DSP Function Library](#) will be included in the application.

Default: Undefined and not included.

Special Issues: The following DSP56852 core registers will be changed:

Table 3-1. DSP56852 OMR Configuration

Operating Mode Register	Definition
Saturation bit (SA) bit 4 Rounding bit (R) bit 5	Enables automatic saturation on 32-bit arithmetic result Enables two's complement rounding (round up)

Table 3-2. DSP56852 SR Configuration

Status Register	Definition
Limit (L) bit 6	Resets or clears the limit bit

Code Example 3-14. DSPFUNC - *appconfig.h* File

```
#define INCLUDE_DSPFUNC
```

3.1.15 DTMF_DET

Macro(s):

INCLUDE_DTMF_DET

Description: When *INCLUDE_DTMF_DET* is defined and included in the *appconfig.h* file as shown below, the [DTMF Detection Library](#) will be included in the application.

Default: Undefined and not included.

Code Example 3-15. DTMF_DET - *appconfig.h* File

```
#define INCLUDE_DTMF_DET
```


3.1.16 DTMF_GEN

Macro(s):

INCLUDE_DTMF_GEN

Description: When *INCLUDE_DTMF_GEN* is defined and included in the *appconfig.h* file as shown below, the **DTMF Generation Library** will be included in the application.

Default: Undefined and not included.

Code Example 3-16. DTMF_GEN - *appconfig.h* File

```
#define INCLUDE_DTMF_GEN
```

3.1.17 FILEIO

Macro(s):

INCLUDE_FILEIO

Description: When *INCLUDE_FILEIO* is defined and included in the *appconfig.h* file as shown below, the **FILE I/O Driver** will be included in the application.

Default: Undefined and not included.

Special Issues: When the FILEIO component is used, **SCI** and **MEMORY** components will automatically be included in the application.

Code Example 3-17. FILEIO - *appconfig.h* File

```
#define INCLUDE_FILEIO
```

3.1.18 G.165

Macro(s):

INCLUDE_G165

Description: When *INCLUDE_G165* is defined and included in the *appconfig.h* file as shown below, the **G.165 Library** will be included in the application.

Default: Undefined and not included.

Code Example 3-18. G.165 - *appconfig.h* File

```
#define INCLUDE_G165
```

3.1.19 G.168

Macro(s):

INCLUDE_G168

Description: When *INCLUDE_G168* is defined and included in the *appconfig.h* file as shown below, the [G.168 Library](#) will be included in the application.

Default: Undefined and not included.

Code Example 3-19. G.168 - *appconfig.h* File

```
#define INCLUDE_G168
```

3.1.20 G.711

Macro(s):

INCLUDE_G711

Description: When *INCLUDE_G711* is defined and included in the *appconfig.h* file as shown below, the [G.711 Library](#) will be included in the application.

Default: Undefined and not included.

Code Example 3-20. G.711 - *appconfig.h* File

```
#define INCLUDE_G711
```

3.1.21 G.726

Macro(s):

INCLUDE_G726

Description: When *INCLUDE_G726* is defined and included in the *appconfig.h* file as shown below, the [G.726 Library](#) will be included in the application.

Default: Undefined and not included.

Code Example 3-21. G.726 - *appconfig.h* File

```
#define INCLUDE_G726
```

3.1.22 GPIO

Macro(s):

INCLUDE_GPIO

Description: When *INCLUDE_GPIO* is defined and included in the *appconfig.h* file as shown below, the **GPIO Driver** will be included in the application.

Default: Undefined and not included.

Special Issues: When *INCLUDE_IO* is used, IO service is automatically included in the application to provide device-independent interface for the **GPIO Driver**. Also, the GPIO component is automatically included if the **LED** component is defined.

Code Example 3-22. GPIO - *appconfig.h* File

```
#define INCLUDE_GPIO
```

3.1.23 IO

Macro(s):

INCLUDE_IO

Description: When *INCLUDE_IO* is defined and included in the *appconfig.h* file as shown below, SDK IO services will be included in the application. For more details about this service, refer to [Chapter 5, “On-Chip Drivers,”](#) [Chapter 6, “Off-Chip Drivers,”](#) and the **SDK Programmer’s Guide**.

Default: Undefined and not included.

Code Example 3-23. IO - *appconfig.h* File

```
#define INCLUDE_IO
```

3.1.24 ITCN

Macro(s):

INCLUDE_ITCN

Description: When *INCLUDE_ITCN* is defined and included in the *appconfig.h* file as shown below, the ITCN initialization will be included in the application.

Default: Undefined and not included.

Special Issues: This component is automatically included if the **BSP** component is defined.

Code Example 3-24. ITCN - *appconfig.h* File

```
#define INCLUDE_ITCN
```

3.1.25 Keypad

Macro(s):

INCLUDE_KEYPAD

Description: When *INCLUDE_KEYPAD* is defined and included in the *appconfig.h* file as shown below, the **Keypad Driver** will be included in the application.

Default: Undefined and not included.

Special Issues: When *INCLUDE_IO* is used, IO service is automatically included in the application to provide device-independent interface for the **Keypad Driver**.

Code Example 3-25. Keypad - *appconfig.h* File

```
#define INCLUDE_KEYPAD
```

3.1.26 LCD

Macro(s):

INCLUDE_LCD

Description: When *INCLUDE_LCD* is defined and included in the *appconfig.h* file as shown below, the **LCD Driver** will be included in the application.

Default: Undefined and not included.

Special Issues: When *INCLUDE_IO* is used, IO service is automatically included in the application to provide device-independent interface for the **LCD Driver**.

Code Example 3-26. LCD - *appconfig.h* File

```
#define INCLUDE_LCD
```

3.1.27 LED

Macro(s):

INCLUDE_LED

Description: When *INCLUDE_LED* is defined and included in the *appconfig.h* file as shown below, the **LED Driver** will be included in the application.

Default: Undefined and not included.

Special Issues: When the LED component is used, the **GPIO** component will automatically be included in the application. Also, when *INCLUDE_IO* is used, IO service is automatically included in the application to provide device-independent interface for the **LED Driver**.

Code Example 3-27. LED - *appconfig.h* File

```
#define INCLUDE_LED
```

3.1.28 MEMORY

Macro(s):

INCLUDE_MEMORY

Description: When *INCLUDE_MEMORY* is defined and included in the *appconfig.h* file as shown below, SDK Memory services will be included in the application. Refer to the **SDK Programmer's Guide** for more details about this service.

Default: Undefined and not included.

Special Issues: None

Code Example 3-28. MEMORY - *appconfig.h* File

```
#define INCLUDE_MEMORY
```

3.1.29 Noise Suppression

Macro(s):

INCLUDE_NS

Description: When *INCLUDE_NS* is defined and included in the *appconfig.h* file as shown below, the [Noise Suppression Library](#) will be included in the application.

Default: Undefined and not included.

Code Example 3-29. Noise Suppression - *appconfig.h* File

```
#define INCLUDE_NS
```

3.1.30 PC Master Software

Macro(s):

INCLUDE_PCMaster

Description: When *INCLUDE_PCMaster* is defined and included in the *appconfig.h* file as shown below, the [PC Master Software SCI Communication Driver](#) will be included in the application.

Default: Undefined and not included.

Special Issues: None

Code Example 3-30. PC Master Software - *appconfig.h* File

```
#define INCLUDE_PCMaster
```

3.1.31 PCS

Macro(s):`INCLUDE_PCS`

Description: When `INCLUDE_PCS` is defined and included in the `appconfig.h` file as shown below, the **Programmable Chip Select (PCS) Driver** will be included in the application.

Default: Undefined and not included.

Special Issues: None

Code Example 3-31. PCS - `appconfig.h` File

```
#define INCLUDE_PCS
```

3.1.32 PLL

Macro(s):`INCLUDE_PLL`

Description: When `INCLUDE_PLL` is defined and included in the `appconfig.h` file as shown below, the PLL initialization will be included in the application. See [Section 3.4](#), Clock (PLL) Configuration, for more information about PLL.

Default: Undefined and not included.

Special Issues: This component is automatically included if the **BSP** component is defined.

Code Example 3-32. PLL - `appconfig.h` File

```
#define INCLUDE_PLL
```

3.1.33 QUAD_TIMER

Macro(s):`INCLUDE_QUAD_TIMER`

Description: When `INCLUDE_QUAD_TIMER` is defined and included in the `appconfig.h` file as shown below, the **Quad Timer Driver** will be included in the application.

Default: Undefined and not included.

Special Issues: When `INCLUDE_IO` is used, IO service is automatically included in the application to provide device-independent interface for the **Quad Timer Driver**. Also, QUAD_TIMER component is automatically included if the **TIMER** component is defined.

Code Example 3-33. QUAD_TIMER - `appconfig.h` File

```
#define INCLUDE_QUAD_TIMER
```

3.1.34 RSA

Macro(s):`INCLUDE_RSA`

Description: When *INCLUDE_RSA* is defined and included in the *appconfig.h* file as shown below, the **RSA Library** will be included in the application.

Default: Undefined and not included.

Code Example 3-34. RSA- *appconfig.h* File

```
#define INCLUDE_RSA
```

3.1.35 SCI

Macro(s):`INCLUDE_SCI`

Description: When *INCLUDE_SCI* is defined and included in the *appconfig.h* file as shown below, the **SCI Driver** will be included in the application.

Default: Undefined and not included.

Special Issues: When *INCLUDE_IO* is used, IO service is automatically included in the application to provide device-independent interface for the **SCI Driver**.

Code Example 3-35. SCI - *appconfig.h* File

```
#define INCLUDE_SCI
```

3.1.36 SERIAL_DATA_FLASH

Macro(s):`INCLUDE_SERIAL_DATA_FLASH`

Description: When *INCLUDE_SERIAL_DATA_FLASH* is defined and included in the *appconfig.h* file as shown below, the **Serial Data Flash Driver** will be included in the application.

Default: Undefined and not included.

Special Issues: When the Serial Data Flash component is used, the **GPIO** and **SPI** components will automatically be included in the application. Also, when *INCLUDE_IO* is used, IO service is automatically included in the application to provide device-independent interface for the **Serial Data Flash Driver**.

Code Example 3-36. Serial Data Flash - *appconfig.h* File

```
#define INCLUDE_SERIAL_DATA_FLASH
```

3.1.37 SIM

Macro(s):

INCLUDE_SIM

Description: When *INCLUDE_SIM* is defined and included in the *appconfig.h* file as shown below, the SIM initialization will be included in the application.

Default: Undefined and not included.

Special Issues: This component is automatically included if the **BSP** component is defined.

Code Example 3-37. SIM - appconfig.h File

```
#define INCLUDE_SIM
```

3.1.38 SPI

Macro(s):

INCLUDE_SPI

Description: When *INCLUDE_SPI* is defined and included in the *appconfig.h* file as shown below, the **SPI Driver** will be included in the application.

Default: Undefined and not included.

Special Issues: When *INCLUDE_IO* is used, IO service is automatically included in the application to provide device-independent interface for the **SPI Driver**.

Code Example 3-38. SPI - appconfig.h File

```
#define INCLUDE_SPI
```

3.1.39 SSI

Macro(s):

INCLUDE_SSI

Description: When *INCLUDE_SSI* is defined and included in the *appconfig.h* file as shown below, the **SSI Driver** will be included in the application.

Default: Undefined and not included.

Special Issues: The SSI component is automatically included if the **CODEC** component is defined.

Code Example 3-39. SSI - appconfig.h File

```
#define INCLUDE_SSI
```


3.1.40 STACK_CHECK

Macro(s):

INCLUDE_STACK_CHECK

Description: When *INCLUDE_STACK_CHECK* is defined and included in the *appconfig.h* file as shown below, SDK stack check services will be included in the application. Refer to the **SDK Programmer's Guide** for more details about this service.

Default: Undefined and not included.

Code Example 3-40. STACK_CHECK - appconfig.h File

```
#define INCLUDE_STACK_CHECK
```

3.1.41 TDC1

Macro(s):

INCLUDE_TDC1

Description: When *INCLUDE_TDC1* is defined and included in the *appconfig.h* file as shown below, the **TDC1 Driver** will be included in the application.

Default: Undefined and not included.

Code Example 3-41. TDC1 - appconfig.h File

```
#define INCLUDE_TDC1
```

3.1.42 TIMER

Macro(s):

INCLUDE_TIMER

Description: When *INCLUDE_TIMER* is defined and included in the *appconfig.h* file as shown below, SDK Timer services will be included in the application. Refer to the **SDK Programmer's Guide** for more details about this service.

Default: Undefined and not included.

Special Issues: When the TIMER component is used, the **QUAD_TIMER** component will be automatically included in the application. Also, the TIMER component is automatically included if the **BUTTON** component is defined.

Code Example 3-42. TIMER - appconfig.h File

```
#define INCLUDE_TIMER
```

3.1.43 UCOS

Macro(s):

INCLUDE_UCOS

Description: When *INCLUDE_UCOS* is defined and included in the *appconfig.h* file as shown below, support for UCOS will be included in the application.

Default: Undefined and not included.

Code Example 3-43. UCOS - *appconfig.h* File

```
#define INCLUDE_UCOS
```

3.1.44 VAD

Macro(s):

INCLUDE_VAD

Description: When *INCLUDE_VAD* is defined and included in the *appconfig.h* file as shown below, the **Voice Activity Detector Library** will be included in the application.

Default: Undefined and not included.

Code Example 3-44. VAD - *appconfig.h* File

```
#define INCLUDE_VAD
```

3.1.45 V.8bis

Macro(s):

INCLUDE_V8BIS

Description: When *INCLUDE_V8BIS* is defined and included in the *appconfig.h* file as shown below, the **V.8bis Library** will be included in the application.

Default: Undefined and not included.

Code Example 3-45. V.8BIS - *appconfig.h* File

```
#define INCLUDE_V8BIS
```

3.1.46 V.21

Macro(s):

INCLUDE_V21

Description: When *INCLUDE_V21* is defined and included in the *appconfig.h* file as shown below, the **V.21 Library** will be included in the application.

Default: Undefined and not included.

Code Example 3-46. V.21 - *appconfig.h* File

```
#define INCLUDE_V21
```

3.1.47 V.22bis

Macro(s):

INCLUDE_V22BIS

Description: When *INCLUDE_V22BIS* is defined and included in the *appconfig.h* file as shown below, the **V.22bis Library** will be included in the application.

Default: Undefined and not included.

Code Example 3-47. V.22BIS - *appconfig.h* File

```
#define INCLUDE_V22BIS
```

3.1.48 V.42bis

Macro(s):

INCLUDE_V42BIS

Description: When *INCLUDE_V42BIS* is defined and included in the *appconfig.h* file as shown below, the **V.42bis Library** will be included in the application.

Default: Undefined and not included.

Code Example 3-48. V.42BIS - *appconfig.h* File

```
#define INCLUDE_V42BIS
```



3.1.49 VR Lite-1

Macro(s):

INCLUDE_VRLITE1

Description: When *INCLUDE_VRLITE1* is defined and included in the *appconfig.h* file as shown below, the [VRLite-1 Library](#) will be included in the application.

Default: Undefined and not included.

Code Example 3-49. VR Lite-1- *appconfig.h* File

```
#define INCLUDE_VRLITE1
```

3.2 Memory Configuration

As explained in the **SDK Programmer's Guide**, the SDK supports different memory configurations by providing different Linker Command Files with directives targeted for a particular memory model. For the DSP56852 platform, the SDK includes Linker Command Files configured for the following memory models: External Memory Operation and Internal Memory Operation.

3.2.1 External Memory Operation

Section 6.3, Memory Configuration, of the **SDK Programmer's Guide** gives an explanation and example of how memory configuration is performed using the Linker Command File. This section documents only the differences that are specific to the DSP56852 platform.

3.2.1.1 Linker Command File

Code Example 3-50. *Linker.cmd*, External Memory

```

*****
#
# Linker.cmd file for DSP56852 Internal RAM
#   using both internal and external program
#   and data memory.
#
*****

MEMORY {

    .pInterruptVector    (RwX) : ORIGIN = 0x000000, LENGTH = 0x000082
    .pInRAM              (RwX) : ORIGIN = 0x000082, LENGTH = 0x00177e
    .pExtRAM             (RwX) : ORIGIN = 0x001800, LENGTH = 0x1EE800

    .pInROM              (RX)  : ORIGIN = 0x1F0000, LENGTH = 0x000400

    .xInRAM              (Rw)  : ORIGIN = 0x000000, LENGTH = 0x000800
    .xInRAM_DynamicMem   (Rw)  : ORIGIN = 0x000800, LENGTH = 0x000800

    .xStack              (Rw)  : ORIGIN = 0x001000, LENGTH = 0x000800
    .xExtRAM_DynamicMem   (Rw)  : ORIGIN = 0x001800, LENGTH = 0x001000
    .xExtRAM             (Rw)  : ORIGIN = 0x002800, LENGTH = 0x1FD400

    .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

#*****

    .ApplicationData :
    {
        # Define variables for C initialization code

        F_Xdata_start_addr_in_ROM = .;
        F_StackAddr                = ADDR(.xStack);
        F_StackEndAddr             = ADDR(.xStack) + SIZEOFW(.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 = 1; # 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)*2);
    WRITEH(SIZEOF(.xIntRAM_DynamicMem)*1);
FmemEMpartitionList = .;
    WRITEH(ADDR(.xExtRAM_DynamicMem)*2);
    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);
```

```
}
```

3.2.1.2 Linking: Set Up Memory Partitions

Using memory directives in the Linker Command File, we are setting up the DSP56852 to use External Program RAM for code and any data that must be put into Program memory, and to use a combination of External and Internal Data RAM for initialized and uninitialized data. [Figure 3-1](#) details the setup of memory partitions.

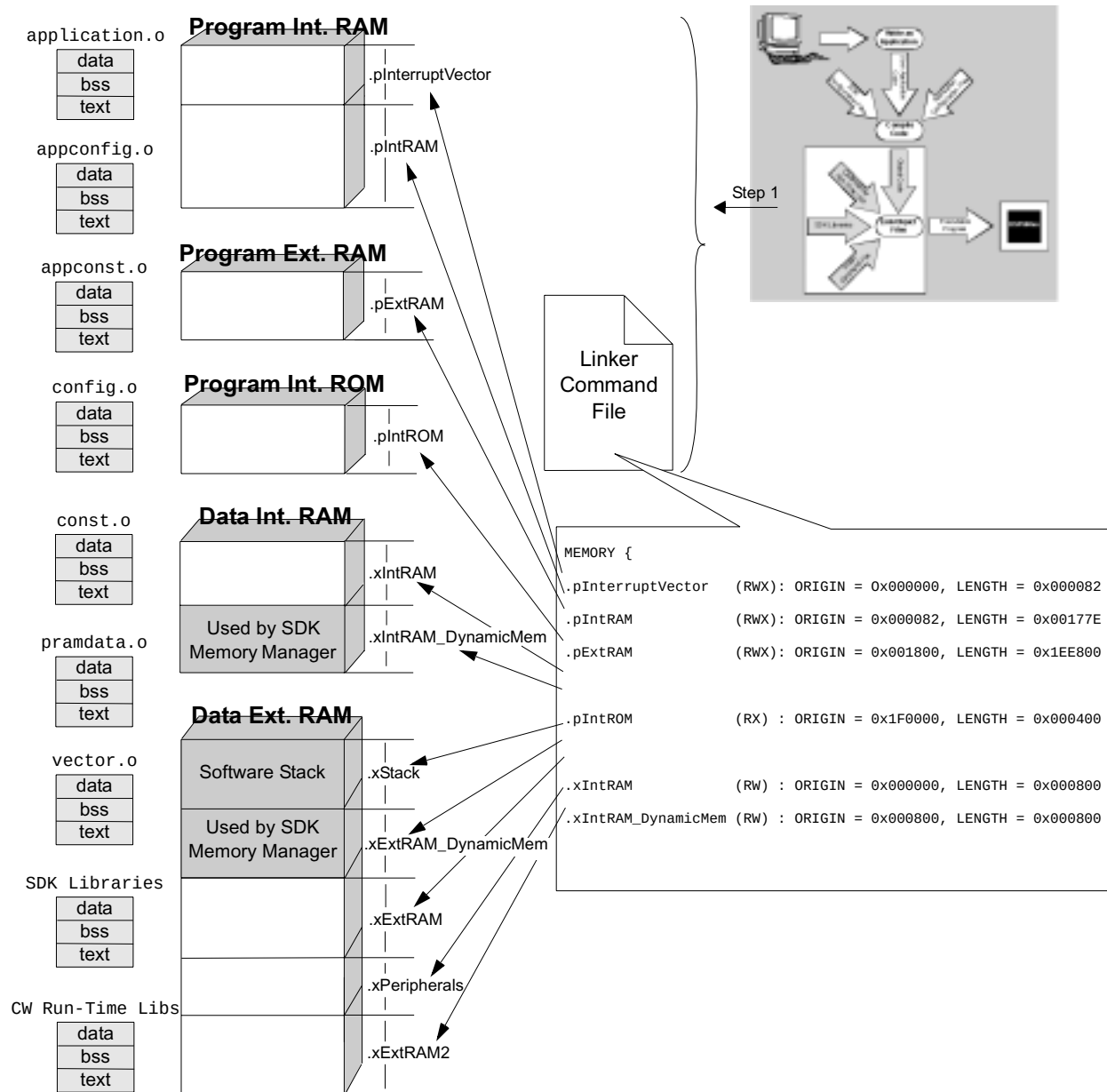


Figure 3-1. Set Up DSP56852 Memory Partitions

3.2.1.3 Programming Target Memory

[Figure 3-2](#) demonstrates how an executable file (which is configured for external memory operation) will be mapped into physical memory on the DSP56852 target.

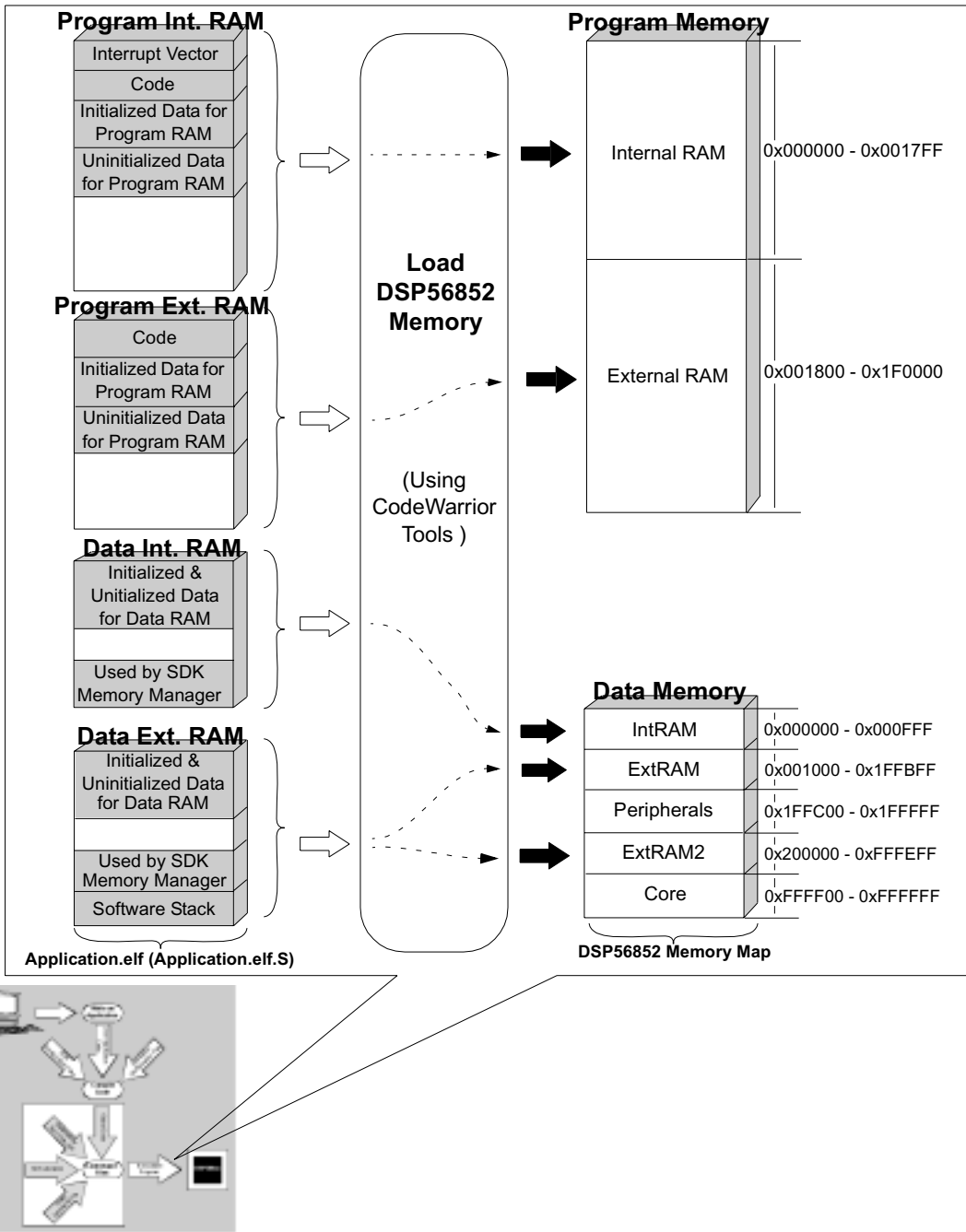


Figure 3-2. Programming Target Memory

3.2.2 Internal Memory Operation

Section 6.3, Memory Configuration, in the SDK Programmer's Guide gives an explanation and example of how memory configuration is performed using the Linker Command File. This section documents only the differences that are specific to the DSP56852 platform.

3.2.2.1 Linker Command File**Code Example 3-51. *Linker.cmd*, Internal Memory**

```

#*****
#
# Linker.cmd file for DSP56852 Internal RAM
#   using only internal program and data memory.
#
#*****

MEMORY {

    .pInterruptVector    (RwX) : ORIGIN = 0x000000, LENGTH = 0x000082
    .pIntRAM              (RwX) : ORIGIN = 0x000082, LENGTH = 0x00177e

    .pIntROM              (RX)  : ORIGIN = 0x1F0000, LENGTH = 0x000400

    .xStack               (Rw)  : ORIGIN = 0x000000, LENGTH = 0x000200
    .xIntRAM               (Rw)  : ORIGIN = 0x000200, LENGTH = 0x000E00

    .xPeripherals         (Rw)  : ORIGIN = 0x1FFC00, LENGTH = 0x000400

    .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;

```

```

} > .pIntrAM

```

```

#*****

```

```

.ApplicationData :
{

```

```

    # Define variables for C initialization code

```

```

F_Xdata_start_addr_in_ROM = .;
F_StackAddr                = ADDR(.xStack);
F_StackEndAddr              = ADDR(.xStack) + SIZEOFW(.xStack) - 1;
F_Xdata_start_addr_in_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

```

```

} > .xIntrAM

```

```

#*****

```

```

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

```

```

}

```

3.2.2.2 Linking: Set Up Memory Partitions

Using memory directives in the Linker Command File, we are setting up the DSP56852 to use Internal Program RAM for code and any data that must be put into Program memory, and to use Internal Data RAM for initialized and uninitialized data. See [Figure 3-3](#) for details on setting up memory partitions.

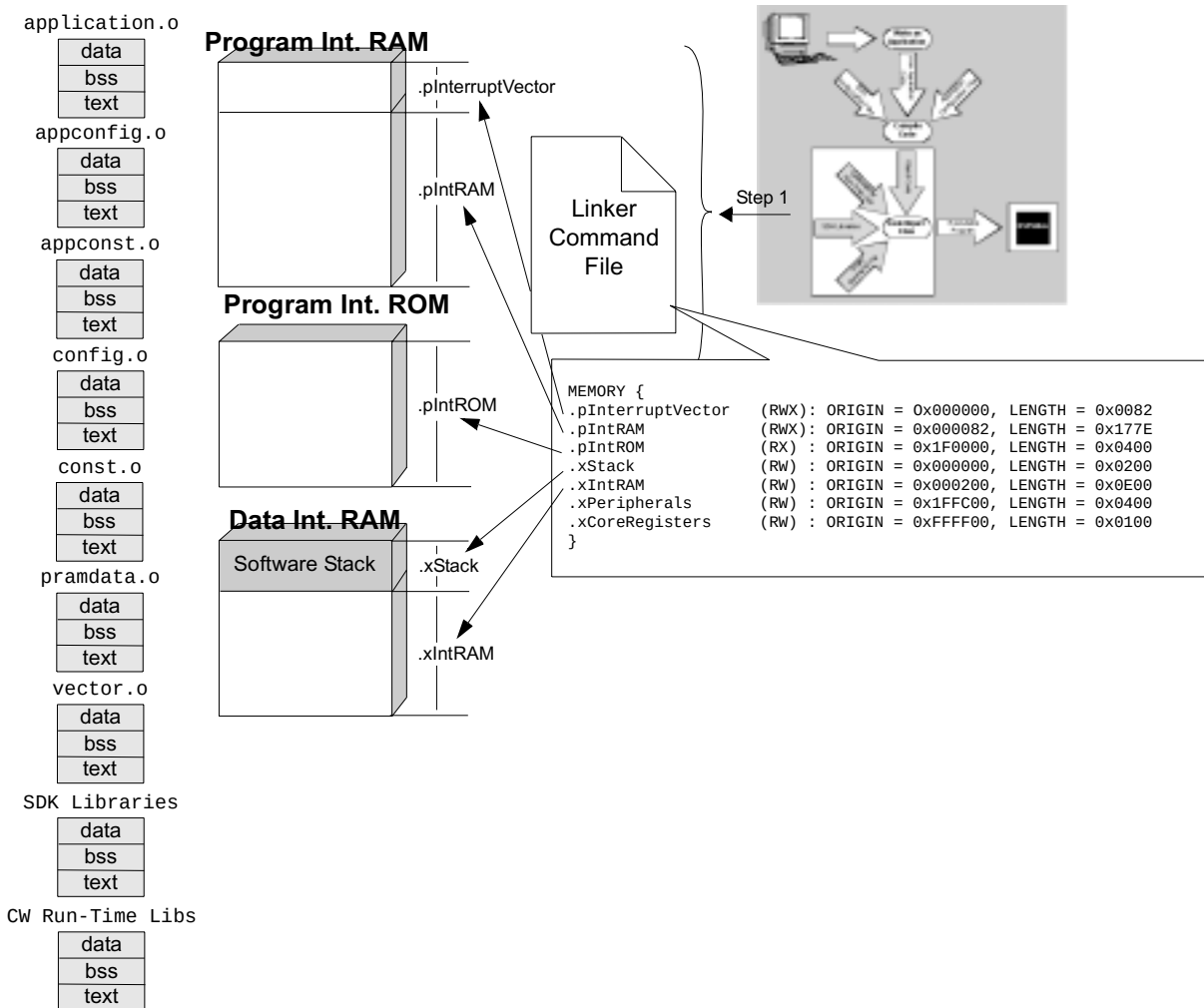
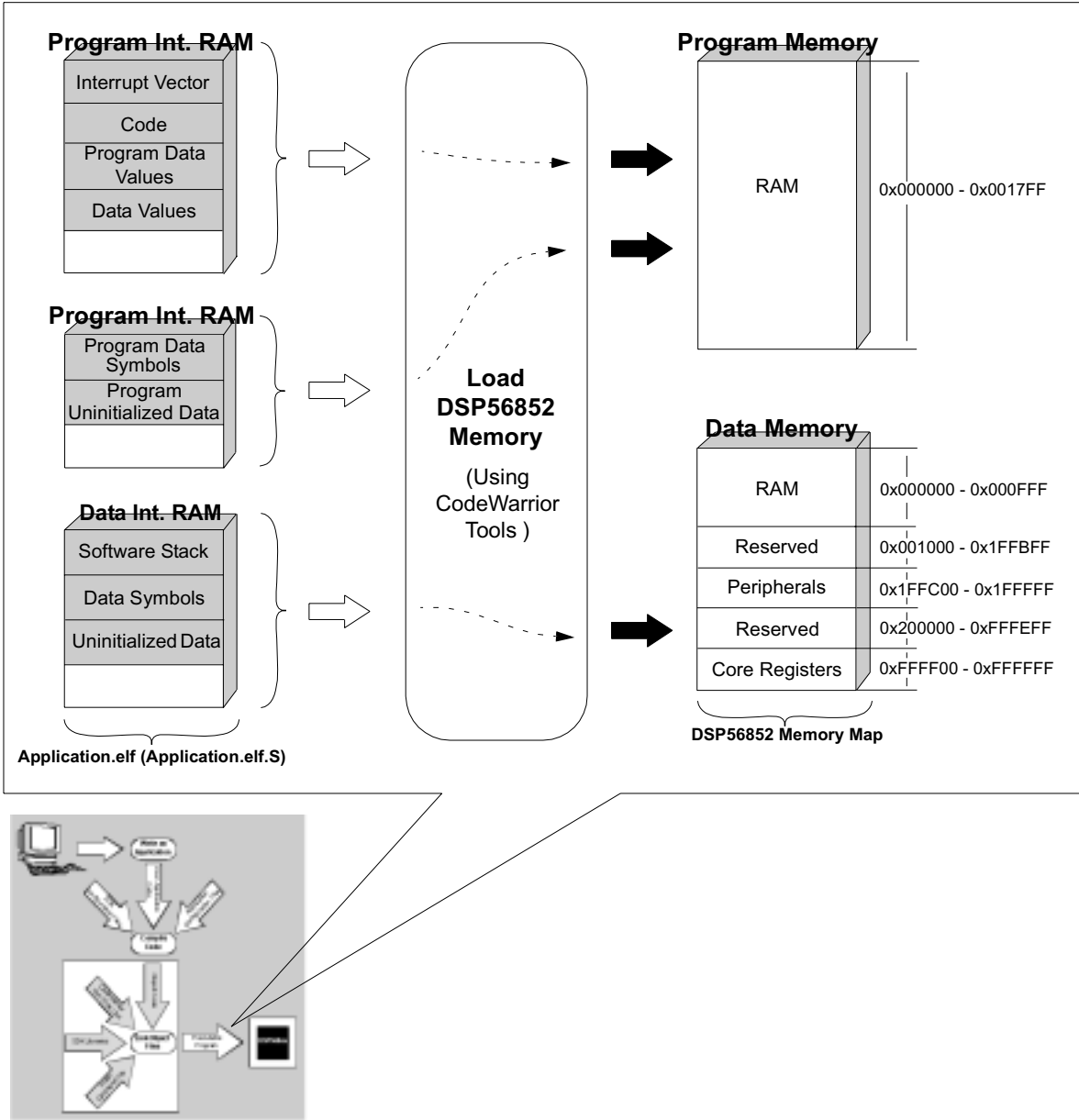


Figure 3-3. Set up DSP56852 Memory Partitions

3.2.2.3 Programming Target Memory

[Figure 3-4](#) demonstrates how an executable file (which is configured for internal memory operation) will be mapped into physical memory on the DSP56852 target.



3.3 DSP56852 Boot Sequence Configuration

In order to execute an application on the DSP56852, the code must first be loaded into memory and data must be initialized. When using the CodeWarrior IDE, this is done automatically. This is acceptable for developing and debugging code, but it is not acceptable for production-ready code. Production-ready code must be loaded and started using the Boot ROM code (First Stage Bootloader) of the DSP5658x and a Second Stage Bootloader provided in the SDK. [Figure 3-5](#) shows the process of loading a User Application.

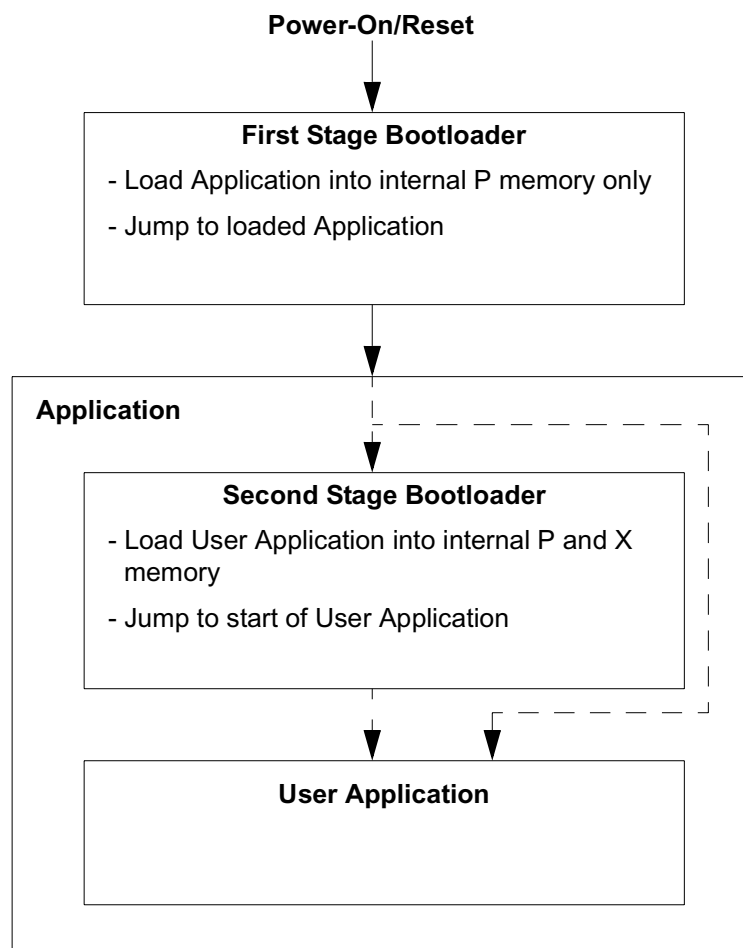


Figure 3-5. DSP56852 Boot Sequence

The First Stage Bootloader is not capable of loading an application that relies on initialized data since it initializes only program memory, so a Second Stage Bootloader is required. The SDK offers Second Stage Bootloaders that can be loaded by the First Stage Bootloader and are capable of loading an application's program and data memory.

3.3.1 First Stage Bootloader

As part of the Power-On/Reset processing, the DSP56852 executes its internal boot ROM code (First Stage Bootloader) in order to load the program memory with code. The DSP56852 can be configured to read code from byte-wide external memory (i.e., Flash), the SPI port, the SCI port, or the Host Interface. It can also be configured to jump directly to external program memory and start executing. The mode in which the DSP56852 boots up depends on the settings of the three MOD pins (MODA, MODB, and MODC) connected to the DSP (please see the **DSP56852 16-Bit Digital Signal Processor User's Manual** to determine how to connect these). These mode bits are latched into the SIM register at Power-On Reset. The First Stage Bootloader reads the SIM register to determine which mode to boot up in. The First Stage Bootloader code can be found in:

...\nos\applications\bootloaders\BootRomTester\boot_ROM_code.asm

Note that this code is masked into the DSP56852 and cannot be changed by the user.

3.3.1.1 Boot from Byte-Wide External Memory (Mode 0)

When configured to boot from byte-wide external memory, the First Stage Bootloader will load code from external data memory starting at location \$040000 using chip select 0.

Table 3-3 shows how the external memory must be organized. The first four bytes read is the length of the code in words (LSB first, MSB last). The next four bytes read is the program address where to start storing the code (LSB first, MSB last). From then on, the next two bytes read will be the next word to store in program memory.

Table 3-3. Format of Data in External Memory

Address	Data Content (8-bit wide)
0x040000	length[0] (LSB)
0x040001	length[1]
0x040002	length[2]
0x040003	length[3] (MSB)
0x040004	ProgStartAddr[0] (LSB)
0x040005	ProgStartAddr[1]
0x040006	ProgStartAddr[2]
0x040007	ProgStartAddr[3] (MSB)
0x040008	prog[0] (LSB)
0x040009	prog[0] (MSB)
0x04000A	prog[1] (LSB)
0x04000B	prog[1] (MSB)

Table 3-3. Format of Data in External Memory

Address	Data Content (8-bit wide)
0x04000C	prog[2] (LSB)
0x04000D	prog[2] (MSB)
...	...
0x040006 + length*2	prog[length-1] (LSB)
0x040007 + length*2	prog[length-1] (MSB)

Example

If length = \$87654321, then:

length[3] = \$87

length[2] = \$65

length[1] = \$43

length[0] = \$21

If ProgStartAddr = \$87654321, then:

ProgStartAddr[3] = \$87

ProgStartAddr[2] = \$65

ProgStartAddr[1] = \$43

ProgStartAddr[0] = \$21

After loading the code, execution will jump to the start of the loaded code. Typically, this code will be a Second Stage Bootloader. Note that the SDK does not currently support a Second Stage Bootloader for this mode.

3.3.1.2 Boot from SPI (Mode 1)

When configured to boot from the SPI port (mode 1), the First Stage Bootloader loads code from the SPI starting at address 0x000000. GPIO Port F Pin 3 is used to chip select the serial device by driving it low. This mode is compatible with the ATMEL AT25xxx and AT45xxx series serial devices. The DSP56852 is the master during the bootstrap process.

Table 3-4 shows how the SPI Data Flash must be organized. The first four bytes must be the string “BOOT”. Since the ATMEL AT25xxx and AT45xxx devices use different protocols, the First Stage Bootloader tries different protocols until it recognizes “BOOT”, or until all protocols have been tried. The next four bytes read is the length of the code in words (LSB first, MSB last). The next four bytes read is the program address where to start storing the code (LSB first, MSB last). From then on, the next two bytes read will be the next word to store in program memory.

Table 3-4. Format of Data in SPI Data Flash

Address	Data Content (8-bit wide)
0x000000	\$42 ('B')
0x000001	\$4F ('O')
0x000002	\$4F ('O')
0x000003	\$54 ('T')
0x000004	length[0] (LSB)
0x000005	length[1]
0x000006	length[2]
0x000007	length[3] (MSB)
0x000008	ProgStartAddr[0] (LSB)
0x000009	ProgStartAddr[1]
0x00000A	ProgStartAddr[2]
0x00000B	ProgStartAddr[3] (MSB)
0x00000C	prog[0] (LSB)
0x00000D	prog[0] (MSB)
0x00000E	prog[1] (LSB)
0x00000F	prog[1] (MSB)
0x000010	prog[2] (LSB)
0x000011	prog[2] (MSB)
...	...
0x00000A + length*2	prog[length-1] (LSB)
0x00000B + length*2	prog[length-1] (MSB)

Example

If length = \$87654321, then:

length[3] = \$87

length[2] = \$65

length[1] = \$43

length[0] = \$21

If ProgStartAddr = \$87654321, then:

```
ProgStartAddr[3] = $87
ProgStartAddr[2] = $65
ProgStartAddr[1] = $43
ProgStartAddr[0] = $21
```

After loading the code, execution will jump to the start of the loaded code. Typically, this code will be the Second Stage Bootloader. The SDK does provide a Second Stage SPI Bootloader.

3.3.1.3 Boot from Normal Expanded Mode (Mode 2)

When configured to boot using normal expanded mode, the First Stage Bootloader simply vectors to external program memory location P:\$040000 using chip select 0.

3.3.1.4 Boot from Development Expanded Mode (Mode 3)

When configured to boot using development expanded mode, the First Stage Bootloader disables internal PRAM and then vectors to external program memory location P:\$000000 using chip select 0.

3.3.1.5 Boot from SCI (Mode 6)

When configured to boot from the SCI port, the First Stage Bootloader loads code from the SCI. If a 4MHz crystal is being used, then the data rate of the SCI port is 38400bps. If a 2MHz crystal is being used, then the data rate of the SCI port is 19200bps. The format of each character is one start bit, 8 data bits, no parity bit, and one stop bit. There is no flow control since the DSP is able to process all data in real time.

Table 3-5 shows how the data coming from the SCI must be organized. The first four bytes read is the length of the code in words (LSB first, MSB last). The next four bytes read is the program address where to start storing the code (LSB first, MSB last). From then on, the next two bytes read will be the next word to store in program memory.

Table 3-5. Format of Data for Boot from SCI

Address	Data Content (8-bit wide)
0x000000	length[0] (LSB)
0x000001	length[1]
0x000002	length[2]
0x000003	length[3] (MSB)
0x000004	ProgStartAddr[0] (LSB)
0x000005	ProgStartAddr[1]
0x000006	ProgStartAddr[2]

Table 3-5. Format of Data for Boot from SCI

Address	Data Content (8-bit wide)
0x000007	ProgStartAddr[3] (MSB)
0x000008	prog[0] (LSB)
0x000009	prog[0] (MSB)
0x00000A	prog[1] (LSB)
0x00000B	prog[1] (MSB)
0x00000C	prog[2] (LSB)
0x00000D	prog[2] (MSB)
...	...
0x000006 + length*2	prog[length-1] (LSB)
0x000007 + length*2	prog[length-1] (MSB)

Example

If length = \$87654321, then:

length[3] = \$87
 length[2] = \$65
 length[1] = \$43
 length[0] = \$21

If ProgStartAddr = \$87654321, then:

ProgStartAddr[3] = \$87
 ProgStartAddr[2] = \$65
 ProgStartAddr[1] = \$43
 ProgStartAddr[0] = \$21

After loading the code, execution will jump to the start of the loaded code. Typically, this code will be a Second Stage Bootloader. Note that the SDK does not currently support a Second Stage Bootloader for this mode.

3.3.1.6 Unused Boot Modes (Modes 4, 5, and 7)

If configured for Modes 4, 5, or 7, the First Stage Bootloader executes a *debughlt* and continuous loop. If the developer has a debugger attached, then the *debughlt* instruction will cause the code to stop executing, allowing the developer to determine why the code is not booting. Otherwise, the code will enter a continuous loop.

3.3.2 Second Stage Bootloader

Since the First Stage Bootloader has limited capability, the SDK offers Second Stage Bootloaders that load applications into program and data memory. Second Stage Bootloaders can be loaded by the First Stage Bootloader because they do not contain any initialized data.

Using a Second Stage Bootloader instead of the First Stage Bootloader offers these benefits:

- a Second Stage Bootloader loads program and data memory
- a Second Stage Bootloader reads S-Record formatted data, which is generated by CodeWarrior

The First Stage Bootloader loads only Program Memory and the format of the data must be in a format that is defined by the First Stage Bootloader, which is not generated by CodeWarrior.

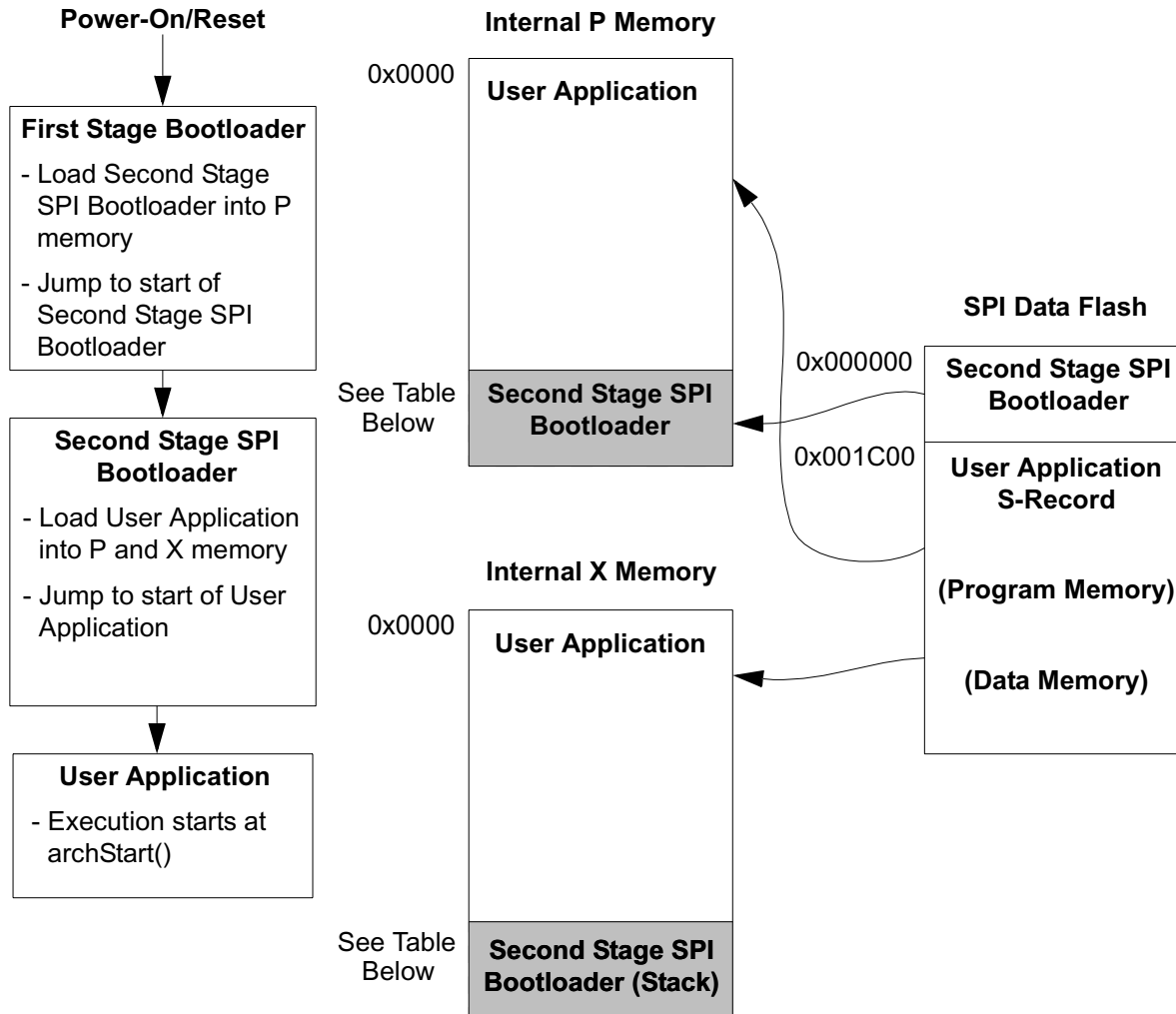
One disadvantage of using a Second Stage Bootloader is that it cannot occupy the same program and data space as the user's application, so the user's application must reserve program and data memory for a Second Stage Bootloader.

For the DSP56852 platform, the SDK offers the following Second Stage Bootloaders:

- **Second Stage SPI Bootloader**

3.3.2.1 Second Stage SPI Bootloader

The Second Stage SPI Bootloader loads a user application from SPI Data Flash into internal program and data memory of the DSP, then jumps to the start of that application. **Figure 3-6** shows how the SDK application is loaded. The Second Stage SPI Bootloader must reside in internal memory due to limitations of the First Stage Bootloader. The Second Stage SPI Bootloader requires 0x01C9 program memory words and 0x0030 data memory words (software stack). Note that the *linker.cmd* file provided with the SDK for the Second Stage SPI Bootloader is for the DSP56852 target. The table in **Figure 3-6** shows where the bootloader should be located for the DSP56852 target.



Target	Second Stage SPI Bootloader Program Memory		Second Stage SPI Bootloader Data Memory	
	Start	End	Start	End
DSP56852	0x1630	0x17FF	0x0FD0	0x0FFF

Figure 3-6. DSP56852 Boot Sequence With Second Stage SPI Bootloader

The Second Stage SPI Bootloader reads S-Record data from the SPI Data Flash and loads the internal program and data memory of the DSP. If the user application requires programming external program or data memory, then the user must modify the Second Stage SPI Bootloader to configure the appropriate chip selects, or the user could develop a Third Stage Bootloader that programs external memory. Note that the application that is loaded by the Second Stage SPI Bootloader cannot reside in the same memory as the Second Stage SPI Bootloader. The Second Stage SPI Bootloader does not have any checks to prevent it from being overwritten, so the user must guarantee this when linking his application.

The Second Stage SPI Bootloader doesn't require initialized data, so it can be loaded by the First Stage Bootloader. The Second Stage SPI Bootloader must be stored at address 0x000000 of the SPI Data Flash as described in [Section 3.3.1.2](#).

The Second Stage SPI Bootloader was not developed using the SDK, because it cannot contain initialized data and the starting address is not 0x000000. However, some functions within the SDK were copied and modified for use in the Second Stage SPI Bootloader. The Second Stage SPI Bootloader uses the following files from the `..\nos\include` directory located in the SDK:

port.h
arch.h
periph.h

Once the user's application has been loaded into memory, the Second Stage SPI Bootloader jumps to the starting address of the application. The starting address is defined in the linker.cmd file of the Second Stage SPI Bootloader. The default is 0x000000, the starting address of all applications that use the SDK.

3.3.2.1.1 Second Stage SPI Bootloader Demo

The Second Stage SPI Bootloader Demo shows how to use the Second Stage SPI Bootloader for the DSP56852EVM board. This demo requires the Serial Data Flash Programmer (see [Section 3.3.4](#)) for programming the Second Stage SPI Bootloader and a user application into the SPI Data Flash.

This demo explains the process of programming the DSP56852EVM SPI Data Flash for use with the Second Stage SPI Bootloader. Once complete, the EVM may be run in stand-alone mode. At Power-On Reset, it will load the Timers application and execute it.

3.3.2.1.1.1 Set-up for Second Stage SPI Bootloader Demo

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**, except :

- Set the Boot MODE (S5) dip-switch to mode 1 (bootstrap from SPI)
- Switch 1 should be Off
- Switches 2 and 3 should be On

The demo set-up requires a serial cable and parallel cable connecting the PC to the DSP56852EVM board.

3.3.2.1.1.2 Procedure for Second Stage SPI Bootloader Demo

- Execute HyperTerminal on the PC
 - Configure HyperTerminal for:

Baud Rate: 115200
 Data Bits: 8
 Parity: None
 Stop Bit: 1
 Flow Control Xon/Xoff

- Using CodeWarrior, open:
`...\nos\applications\bootloaders\serial_flash_prog\serial_flash_prog.mcp`
- Build and download the project; when the debugger reaches *main()*, it will stop
- Start HyperTerminal
- Select *Run* in the debugger to continue executing the Serial Data Flash Programmer
- The terminal will display a menu:

```
(c) 2002 Motorola Inc. 5685x Serial Data Flash Programmer. Version 1.0

Select :
1. Read 2nd Stage Bootloader S-Record and Store in Data Flash
2. Read Application S-Record and Store in Data Flash
3. Exit
->
```
- Press 1
- The terminal will display:
Waiting for 2nd Stage Bootloader S-Record
- Select HyperTerminal menu *Transfer\Send Text File*
- Select the file:
`...\nos\applications\bootloaders\2nd_stage_bootloader\Debug\InternalRAM.elf.S`
- Wait for next HyperTerminal message:
Loaded 0x01c9 Program and 0x0000 Data words.

```
(c) 2002 Motorola Inc. 5685x Serial Data Flash Programmer. Version 1.0

Select :
1. Read 2nd Stage Bootloader S-Record and Store in Data Flash
2. Read Application S-Record and Store in Data Flash
3. Exit
->
```
- The Second Stage SPI Bootloader is now programmed into the SPI Data Flash
- To program the user application into SPI Data Flash, press 2
- The terminal will display:
Waiting for Application S-Record
- Select HyperTerminal menu *Transfer\Send Text File*
- Select any SDK application S-Record file for internal memory
 - For this example, use:
`...\nos\applications\bsp\timers\Debug\InternalRAM.elf.S`
 - If this file does not exist, open the *timers.mcp* project
 - Change the project settings to generate an S-Record
 - Rebuild the *timers.mcp* project
- Wait for the next HyperTerminal message:
Loaded 0x0744 Program and 0x0062 Data words.

```
(c) 2002 Motorola Inc. 5685x Serial Data Flash Programmer. Version 1.0

Select :
1. Read 2nd Stage Bootloader S-Record and Store in Data Flash
2. Read Application S-Record and Store in Data Flash
3. Exit
->
```


- The Timers Application is now programmed into the SPI Data Flash
- To exit the Serial Data Flash Programmer, press 3
- Kill the debugger session for the Serial Data Flash Programmer
- Disconnect the Parallel Port Cable from the DSP56852EVM
- Remove power from the DSP56852EVM
- Re-apply power to the DSP56852EVM
- The LEDs should be blinking

3.3.3 User Application

The SDK start-up code for all SDK user applications, starting at *archStart()*, is described in **Section 6.4**, Boot Sequence Configuration, of the **SDK Programmer's Guide**. Note that this is a description of the SDK boot sequence and does not include the execution of the DSP56852 Boot ROM code.

3.3.4 DSP56852EVM Serial Data Flash Programmer

The SDK offers a Serial Data Flash Programmer that programs a Second Stage SPI Bootloader and a User Application into the Serial Data Flash. The Serial Data Flash Programmer supports the easiest serial protocol, so a standard serial terminal program can be used on the host PC.

The Serial Data Flash Programmer must be loaded into memory using the CodeWarrior IDE. Once the Second Stage SPI Bootloader and User Application are loaded into serial data flash and the DSP56852 is configured to boot from SPI mode, the User's Application can be executed without the use of CodeWarrior.

A demo of the Serial Data Flash Programmer is described in the demo of the Second Stage SPI Bootloader. Please see [Section 3.3.2.1.1](#).

3.3.4.1 Specification

The Serial Data Flash Programmer reads the S-Record file of the user application (for example, generated by CodeWarrior) via the serial interface, parses this S-Record file and stores needed data in the Serial Data Flash. If any error occurs during the loading of the S-Record file, the Serial Data Flash Programmer outputs an error message containing an error number via the serial interface. The errors that may be generated are:

INDICATE_ERROR_RECEIVE	1
INDICATE_ERROR_CHARACTER	2
INDICATE_ERROR_FORMAT	3
INDICATE_ERROR_CHECKSUM	4
INDICATE_ERROR_OVERRUN	5
INDICATE_ERROR_FLASH	6
INDICATE_ERROR_INTERNAL	7
INDICATE_ERROR_PARITY	8
INDICATE_ERROR_LOAD	9

If an error occurs, the user should note the error, kill the debugger session, exit HyperTerminal, then restart.

The user can choose one of the following actions for the Serial Data Flash Programmer to perform:

- Read Second Stage Bootloader S-Record and Store in Data Flash
This stores the loaded S-Record in the serial data flash at address 0x0000 in the format that can be recognized by the First Stage Bootloader when configured to boot using Mode 1 (boot from SPI).
- Read Application S-Record and Store in Data Flash
This stores the loaded S-Record in the Serial Data Flash starting at address 0x1C00 in the format that can be recognized by the Second Stage SPI Bootloader.
- Exit
This exits the Serial Data Flash Programmer.

3.4 Clock (PLL) Configuration

SDK provides a fast and easy way for configuring DSP56852 clocks. By modifying the SDK default configuration parameters, a user can change the clock frequency for the DSP56852 core and peripherals.

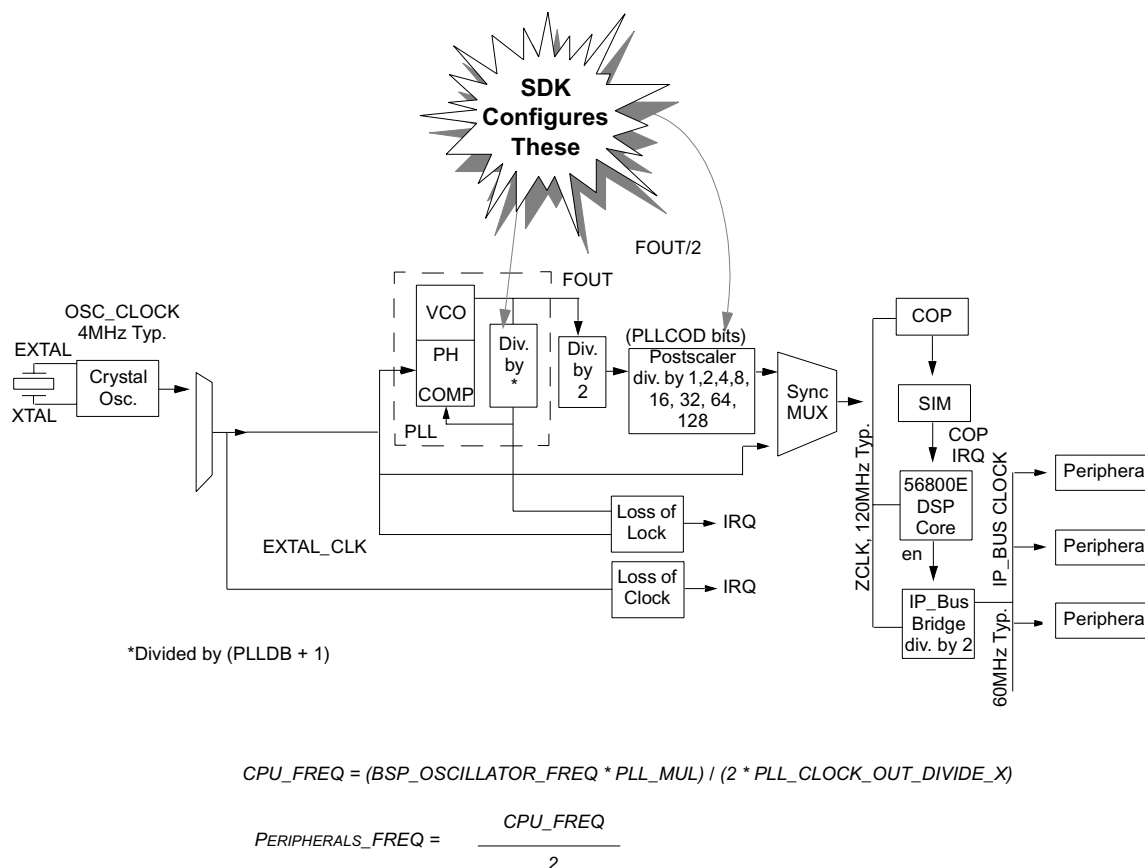


Figure 3-7. SDK Clock Configuration for the DSP56852

Figure 3-7 shows the SDK configuration parameters that are available to a user.

BSP_OSCILLATOR_FREQ defines the frequency of the external oscillator and is located in the *config.h* configuration file.

PLL_CLOCK_OUT_DIVIDE_BY_X defines a value of the PLL postscaler, and can be set to:

- *PLL_CLOCK_OUT_DIVIDE_BY_1*
- *PLL_CLOCK_OUT_DIVIDE_BY_2*
- *PLL_CLOCK_OUT_DIVIDE_BY_4*
- *PLL_CLOCK_OUT_DIVIDE_BY_8*
- *PLL_CLOCK_OUT_DIVIDE_BY_16*
- *PLL_CLOCK_OUT_DIVIDE_BY_32*
- *PLL_CLOCK_OUT_DIVIDE_BY_64*
- *PLL_CLOCK_OUT_DIVIDE_BY_128*

PLL_MUL can be any number in the range from 1 to 128.

Application developers can use these configuration parameters to select desired CPU speeds. For example, to set a CPU speed to 9MHz, we will use the equation from [Figure 3-7](#), and redefine the default configuration parameters (from *config.h*) in the *appconfig.h* file as follows:

Code Example 3-52. PLL_MUL / *appconfig.h* File

```
#define PLL_MUL 18

#define PLL_DIVIDE_BY_REG (( PLL_MUL - 1 ) \
                          | PLL_CLOCK_OUT_DIVIDE_BY_4)
```

In [Code Example 3-52](#), we will be using *BSP_OSCILLATOR_FREQ* (with a default value set to 4000000 in the *config.h* file); *PLL_MUL* (with a user-set value of 18 in the *appconfig.h* file); and *PLL_CLOCK_OUT_DIVIDE_BY_4* to define the CPU clock to $4000000 * 18 / 2 / 4 = 9000000\text{Hz}$.

NOTE: The default SDK setting for CPU clock is 120MHz (*BSP_OSCILLATOR_FREQ* = 4000000, *PLL_MUL* = 60, *PLL_CLOCK_OUT_DIVIDE_BY_1*)



Chapter 4

Interrupt Processing for the DSP56852

This section describes specifically how to implement interrupts on the DSP56852 using the SDK. Please refer to the **Embedded SDK Programmer's Guide** for a high-level description of interrupts for the DSP568xx family of DSPs.

4.1 Interrupt Dispatcher

The SDK for the DSP56852 platform comes with interrupt dispatchers that provide normal and fast context saves.

4.1.1 Super Fast Interrupt Dispatcher

A super fast interrupt dispatcher is not required, since the interrupt dispatcher is bypassed altogether when using a super fast ISR.

4.1.2 Fast Interrupt Dispatcher

The fast interrupt dispatcher for the DSP56852 is equivalent to the normal interrupt dispatcher, which is already very fast. Therefore, code developed for a DSP that uses a fast interrupt dispatcher will still work with the DSP56852.

Note that the hardware fast interrupt offered by the DSP56852 is different from the SDK's fast interrupt dispatcher. You may use the hardware's fast interrupt with the SDK, but the SDK provides no special software support for the hardware fast interrupt mechanism.

4.1.3 Normal Interrupt Dispatcher

The normal interrupt dispatcher saves all registers (n, a0, a1, a2, b0, b1, b2, c2, d2, x0, y0, y1, r0, r1, r2, r3, r4, omr, la, la2, m01, lc, lc2); saves the hardware stack; sets the modifier register (m01) to -1 (linear arithmetic); sets convergent rounding and no saturation mode in the OMR; then jumps to the installed ISR.

The overhead required when using the SDK normal interrupt dispatcher for the DSP56852 is shown in [Table 4-1](#).

4.1.4 SDK ISR Overhead

The following tables provide the number of instructions and clock cycles that the super fast, fast, and normal interrupt dispatchers require when they are executing from internal program memory. For the values below, the clock cycles for the initial JSR instruction to the ISR and the RTI instruction are not included because they are the same for each type of ISR and they are not actually part of the SDK ISR Overhead. They are due to the hardware overhead in dealing with interrupts. The initial JSR instruction takes four clock cycles and the RTI takes eight clock cycles.

Table 4-1. SDK ISR Overhead for DSP56852 (Running From Internal Memory)

Program Code - Internal Memory	Interrupt Dispatcher Instructions		Interrupt Dispatcher Clock Cycles	
	Super Fast	Fast/Normal	Super Fast	Fast/Normal
SDK ISR Overhead from the point where the DSP allows the interrupt to the point where interrupts of higher priority are unmasked	0	0	0	0
SDK ISR Overhead from the point where the DSP allows the interrupt to the start of User ISR	0	36	0	61
SDK ISR Overhead from the end of User ISR to the point where the Interrupt Dispatcher returns to the User Code	0	30	0	46

4.1.5 SDK ISR Stack Requirements

Table 4-2 provides the stack size requirements for the super fast, fast, and normal interrupt dispatchers. Note that these numbers include the two words required for the initial ISR from the interrupt vector table.

Calculating the overall amount of software stack consumed by ISRs depends on the number of nested ISRs; the type of ISR (fast/normal, or super fast); and the size of the ISRs. Once you have this information, you must account for the maximum software stack consumed by your application, which can be very complicated. To make the process of calculating the software stack easier, the SDK offers a stack check capability to determine how much stack the application (including ISRs) uses. Please see **Section 5.2.2** of the **SDK Programmer's Guide** for more information.

Table 4-2. SDK ISR Stack Requirements for DSP56852

Interrupt Dispatcher	Software Stack Size Consumed by SDK (words per interrupt level)
Super Fast	2
Fast/Normal	50

4.2 Nested Interrupts

Nested interrupts are automatically supported by DSP56852 core.

4.3 Interrupt Priorities

By default, as defined in *config.h*, `GPR_INT_PRIORITY_XX` for each interrupt is set to 0, which disables the interrupt. The `GPR_INT_PRIORITY_XX` for each interrupt to be used should be set between 1 and 3 in *appconfig.h* (3 is the highest priority). If the `GPR_INT_PRIORITY_XX` is not set between 1 and 3, then the vector in the interrupt vector table for the specific interrupt will contain the address of *configUnhandledInterruptISR*.

Note that `GPR_INT_PRIORITY_XX` is defined automatically to be 1 for interrupts used by device drivers that are included in *appconfig.h*. If you want the interrupts used to implement the device drivers to be nested, this definition can be changed in the *appconfig.h*.

4.4 Device Drivers

The following SDK device drivers used by the DSP56852 automatically set `GPR_INT_PRIORITY_xx` to 1 for the interrupts the drivers require.

If the user determines he does not need one of the interrupts for a driver, he can define the corresponding `GPR_INT_PRIORITY_XX` label to be 0.

4.4.1 Button Driver

When enabled, the Button driver uses interrupt numbers 17 and 18, so the following interrupt priorities are automatically set to 1:

- `GPR_INT_PRIORITY_17`
- `GPR_INT_PRIORITY_18`

4.4.2 Codec Driver

The Codec driver is implemented using the GPIO and SSI drivers; see [Section 4.4.3](#) and [Section 4.4.9](#) for details.

4.4.3 GPIO Driver

This driver does not use interrupts.

4.4.4 LED Driver

The LED driver is implemented using the GPIO driver; see [Section 4.4.3](#) for details.

4.4.5 Quad Timer Driver

When enabled, the Quad Timer driver uses interrupt numbers 52 to 63, automatically setting the following interrupt priorities to 1:

- GPR_INT_PRIORITY_52
- GPR_INT_PRIORITY_53
- GPR_INT_PRIORITY_54
- GPR_INT_PRIORITY_55
- GPR_INT_PRIORITY_56
- GPR_INT_PRIORITY_57
- GPR_INT_PRIORITY_58
- GPR_INT_PRIORITY_59
- GPR_INT_PRIORITY_60
- GPR_INT_PRIORITY_61
- GPR_INT_PRIORITY_62
- GPR_INT_PRIORITY_63

4.4.6 SCI Driver

The SCI driver, when enabled, uses interrupt numbers 42 to 46, so the following interrupt priorities are automatically set to 1:

- GPR_INT_PRIORITY_42
- GPR_INT_PRIORITY_43
- GPR_INT_PRIORITY_44
- GPR_INT_PRIORITY_45
- GPR_INT_PRIORITY_46

4.4.7 Serial Data Flash Driver

The Serial Data Flash driver is implemented using the GPIO and SPI drivers; see [Section 4.4.3](#) and [Section 4.4.8](#) for details.

4.4.8 SPI Driver

This driver does not use interrupts.

4.4.9 SSI Driver

The SSI driver, when enabled, uses interrupt numbers 28, 29, 31, and 32, automatically setting the following interrupt priorities to 1:

- GPR_INT_PRIORITY_28
- GPR_INT_PRIORITY_29
- GPR_INT_PRIORITY_31
- GPR_INT_PRIORITY_32

4.4.10 Timer Driver

The Timer driver is implemented using the Quad Timer driver, detailed in [Section 4.4.5](#).



Chapter 5

On-Chip Drivers

One strength of the SDK is that it provides a high degree of architectural and hardware independence for application code. This portability is due to the SDK's modular design, which, in this case, isolates all chip-specific functionality into a set of libraries that are part of a Board Support Package (BSP). This BSP can be found at <...>SDK\src\dsp56852evm\nos\bsp of the SDK.

While there are two major components, Interfaces and Drivers, this chapter focuses on the DSP56852 on chip device drivers only. Specific EVM hardware peripheral drivers are addressed in [Chapter 6](#).

5.1 Computer Operating Property (COP) Driver

The COP Driver supports a mechanism for initializing COP registers, the services to reset the COP timer and to check system status. Current restrictions of the COP Driver functionality are related to the hardware implementation, which disables the COP module when the OnCE debug interface is active. Due to this hardware limitation, the COP module does not generate reset in the debug mode, even though the COP timeout has expired.

5.1.1 Initialization

The SDK file *appconfig.h* provides the mechanism to statically configure the COP driver. The *appconfig.h* file allows the programmer to set C preprocessor variables which configure the COP driver default operation. Typically, these C preprocessor variables conditionally compile and link information into the application for use by the COP driver during initial operation.

To include the COP Driver, add the following to *appconfig.h*:

```
#define INCLUDE_COP
```

To have the COP enabled during initialization of the code, define a timeout value (in usec) in *appconfig.h*:

```
#define COP_TIMEOUT 1000000L
```

The COP control register can also be statically defined by the user. The defined constant, *COP_CONTROL_REG*, can be defined in the *appconfig.h* file by ORing any combination of the following defined constants:

- *COP_CLK_SRC_IS_IPBUS* - When defined, IPBus Clock is source. Default is oscillator (OSCCLK).
- *COP_RUN_IN_STOP* - When defined, COP counter will run in Stop mode. Default is COP counter will not run.
- *COP_RUN_IN_WAIT* - When defined, COP counter will run in Wait mode. Default is COP counter will not run.
- *COP_ENABLE* - When defined, COP is enabled. Default is disabled.
- *COP_WRITE_PROTECT* - When defined, the COP Registers are read only. Default is COP Registers are read/write.

The default definition of *COP_CONTROL_REG* as defined in *config.h* is:

```
#define COP_CONTROL_REG COP_ENABLE
```

To change this to have the registers be read only, define the following in *appconfig.h*:

```
#define COP_CONTROL_REG (COP_ENABLE | COP_WRITE_PROTECT)
```

If you do not define *COP_CONTROL_REG* to include *COP_ENABLE*, the COP will not be enabled. If you do not define a *COP_TIMEOUT*, the COP will not be enabled.

By adding the above-defined constants in *appconfig.h*, the COP will be enabled at initialization and the timeout will be 1000000usec, or 1 second.

Note that the maximum timeout is 2100000usec (this assumes the source clock is 4MHz).



To prevent the COP from timing out, call *copReload()* prior to the timeout. This will reset the COP, preventing it from timing out and resetting the DSP.

The COP peripheral is disabled by design when the processor is in Debug mode. Therefore, operation may only be verified when running the processor in stand-alone mode.

Please refer to the **DSP56F852 User's Manual** for a detailed description of the COP peripheral.



5.1.2 API Definition

The following pages specify the Application Programming Interface (API).

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

The *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.

5.1.2.1 *copInitialize()*

Call(s):

```
void copInitialize(UWord16 CtrlReg, UWord16 TReg);
```

Arguments:

Table 5-1. COP Driver Arguments - *copInitialize()*

<i>CtrlReg</i>	<i>in</i>	This argument will be written directly to the COP Control Register. The following macro, or its combinations, can be used: COP_RUN_IN_STOP - COP module will be active in the STOP mode COP_RUN_IN_WAIT - COP module will be active in the WAIT mode COP_ENABLE - Enables COP timeout processing COP_WRITE_PROTECT - Disables writing to the COP Control Register
<i>TReg</i>	<i>in</i>	This argument will be written directly to the COP Timeout Register; the maximum allowed value is 0xFFFF and minimum value is 0

Description: The *copInitialize()* reloads the COP timer and initializes COP Control and Timeout registers with the specified values.

Returns: None

Code Example 5-1. *copInitialize*

```
#include "cop.h"

/* Initialize COP module with the maximum allowed timeout value */

copInitialize(COP_ENABLE|COP_WRITE_PROTECT,0xFFFF);
```

5.1.2.2 *copReload()*

Call(s):

```
void copReload(void);
```

Arguments: None

Description: The *copReload()* writes a sequence to the COP peripheral to avoid having the COP Timer expire. The magical sequence is writing a 0x5555 to the COP Counter Register, followed by a 0xAAAA. The second write (0xAAAA) does not have to be the next instruction after the first write (0x5555); however, for this function, it is written that way.

This function must be called by the user application prior to the COP peripheral timing out. Place a *copReload()* call into the critical part of code that is executed periodically. If the code is not executed during the time specified by the COP_TIMEOUT value, the DSP chip will be restarted. Avoid placing the *copReload()* call into the interrupt service routine, because the interrupt can be going well, but the main code can be looping at the same time.

Returns: None

Code Example: See [Code Example 5-3](#).

5.1.2.3 *copForceReset()*

Call(s):

```
void copForceReset(void);
```

Arguments: None

Description: The *copForceReset()* call forces COP timeout and causes the processor to reset. This function may be used to debug the application's COP exception handling.

Note that if the COP control register has the COP_WRITE_PROTECT bit set, then this function will not write to the COP.

Returns: None**Code Example:** None

5.1.2.4 copIsReset()**Call(s):**

```
boolean copIsReset(void);
```

Arguments: None

Description: The *copIsReset()* function checks a bit in the Software Control Register 1 to determine if a COP Reset has occurred.

When a Power-On reset occurs, the Software Control Register 1 is cleared to 0. If the COP Driver is included, then a bit in this register (COP_RESET_PWR) is checked to see if it is 0 (Power-On Reset) or 1 (other reset). If the bit is 0, then the COP initialization code sets the bit, indicating a Power-On Reset has occurred. If a COP reset occurs, the COP_RESET_PWR bit in the Software Control Register 1 is still set. The COP initialization code then sets another bit (COP_RESET_COP) in the Software Control Register 1, indicating that a COP reset has occurred.

Returns: The *copIsReset()* returns "true" if COP reset occurred before and "false" in other cases (Power-On Reset).

Special Issues: This function utilizes the SIM module Software Control Register 1. Use of this register for any other purpose may cause incorrect responses.

Code Examples:

Code Example 5-2 shows how to overwrite the default COP driver configuration in the *appconfig.h* file. This example configures the COP timeout to be 1 second.

Code Example 5-2. Overwriting Default COP Driver Settings in *appconfig.h*

```
#define INCLUDE_COP

#define COP_TIMEOUT          1000000L
```

Code Example 5-3 shows how to use the COP Driver API.

NOTE: This sample will not work when the OnCE debug interface is active. Please refer to the **DSP56852 Evaluation Module Hardware User's Manual** to learn how to disable the OnCE port.


Code Example 5-3. COP Driver API usage

```

/*****
 *
 * Motorola Inc.
 * (c) Copyright 2002 Motorola, Inc.
 * ALL RIGHTS RESERVED.
 *
 *****/
 *
 * File Name:
 *   cop.c
 *
 * Description:
 *   This file contains demo code that exercises the Computer Operating
 *   Properly (COP) peripheral and driver.
 *
 *   This code must be executed without the Debugger attached. When the
 *   Debugger is attached, the COP peripheral is disabled by design. In
 *   order to execute this demo, you must first use the Serial Data Flash
 *   Programmer to load the S-Record of this application into the serial
 *   data flash of the EVM. If you haven't already done so, you will need
 *   to also load the S-Record of the Second Stage SPI Bootloader into
 *   the serial data flash as well.
 *
 *   Once you have programmed the serial data flash, you will need to
 *   remove power from the board, make sure the Boot MODE switch is
 *   set for Mode 1 (boot from SPI), and then re-apply power to the
 *   board.
 *
 *   The flashing RED LED indicates that the user application is running.
 *   The YELLOW LED indicates that a COP or HW reset has occurred.
 *   The toggling GREEN LED indicates that multiple COP or HW resets are
 *   occurring.
 *
 *   In order to prevent COP resets from occurring, you can press the
 *   IRQA button. The code associated with this button will call the
 *   copReload() function which will reset the COP Timer preventing it
 *   from timing out.
 *
 *****/

#include "port.h"
#include "io.h"
#include "fcntl.h"
#include "bsp.h"
#include "led.h"
#include "cop.h"
#include "button.h"

#define DELAY_COUNT 0x080000

void ButtonAFunc (void *pCallbackArg);
button_sCallback ButtonACallbackSpec = {ButtonAFunc, NULL};

```



```

/*****
*
* Module:
*   ButtonAFunc()
*
* Description:
*   When Button A is pressed, this function is called. This function
*   calls copReload() which resets the COP Timer and prevents the COP
*   from timing out.
*
* Returns:
*   None
*
* Global Data:
*   None
*
* Arguments:
*   pCallbackArg - Not Used
*
* Range Issues:
*
* Special Issues:
*   The button driver must be opened first
*
*****/

void ButtonAFunc (void *pCallbackArg)
{
    copReload();
}

/*****
*
* Module:
*   main()
*
* Description:
*   This function is the main function of the COP demo.
*
* Returns:
*   None
*
* Global Data:
*   None
*
* Arguments:
*   None
*
* Range Issues:
*
* Special Issues:
*
*****/

```



```
void main(void)
{
    UWord32 DelayCnt = 0;
    types_tHandle LedFD;
    types_tHandle ButtonA;

    /* Open LED driver */

    LedFD = open(BSP_DEVICE_NAME_LED_0, 0);

    /* Open the Button driver */

    ButtonA = open(BSP_DEVICE_NAME_BUTTON_A, 0, &ButtonACallbackSpec);

    if (copIsReset())
    {
        ioctl(LedFD, LED_ON, LED_YELLOW2); /* COP RESET */

        /* TOGGLE GREEN2 LED FOR MULTIPLE COP RESETS */

        if (periphMemRead(&ArchIO.SIM.SoftwareControlReg2))
        {
            ioctl(LedFD, LED_ON, LED_GREEN2);
            periphMemWrite(0,&ArchIO.SIM.SoftwareControlReg2);
        }
        else
        {
            ioctl(LedFD, LED_OFF, LED_GREEN2);
            periphMemWrite(1,&ArchIO.SIM.SoftwareControlReg2);
        }
    }
    else
    {
        ioctl(LedFD, LED_OFF, LED_YELLOW2); /* POWER-ON RESET */
    }

    do
    {
        /* TOGGLE RED LED TO INDICATE USER APPLICATION IS RUNNING */

        DelayCnt++;
        if (DelayCnt > DELAY_COUNT)
        {
            ioctl(LedFD, LED_TOGGLE, LED_RED2);
            DelayCnt = 0;
        }
    }
    while (1);
}
```

5.1.3 COP Driver Tests

Currently, there are no automated COP Driver tests in the SDK.

5.1.4 COP Driver Demos

5.1.4.1 COP Demo

The COP demo shows that the COP Driver and Peripheral are operating correctly.

This demo must be executed without the Debugger attached. When the Debugger is attached, the COP peripheral is disabled by design. In order to execute this demo, you must first use the Serial Data Flash Programmer to load the S-Record of this application into the serial data flash of the EVM. If you haven't already done so, you must also load the S-Record of the Second Stage SPI Bootloader into the serial data flash as well.

Once you have programmed the serial data flash, remove power from the board, make sure the Boot MODE switch is set for Mode 1 (boot from SPI), then reapply power to the board.

Once the demo is running, the RED, YELLOW, and GREEN LEDs will indicate the state of the demo. The flashing RED LED indicates that the user application is running. The YELLOW LED indicates that a COP or Hardware reset has occurred. The toggling GREEN LED indicates that multiple COP or Hardware resets are occurring.

In order to prevent COP resets from occurring, press the IRQA button. The code associated with this button will call the *copReload()* function, resetting the COP Timer and preventing it from timing out.

5.1.4.1.1 Set-up for COP Demo

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**, except:

- Set the Boot MODE (S5) dip-switch to mode 1 (bootstrap from SPI)
- Switch 1 should be Off
- Switches 2 and 3 should be On

The demo set-up requires a serial cable and parallel cable connecting the PC to the DSP56852EVM board.

5.1.4.1.1.1 Procedure for COP Demo

- Execute HyperTerminal on the PC
 - Configure HyperTerminal for:

Baud Rate:	115200
Data Bits:	8
Parity:	None
Stop Bit:	1
Flow Control:	Xon/Xoff

- Using CodeWarrior, open:
`...\nos\applications\bootloaders\serial_flash_prog\serial_flash_prog.mcp`
- Build and download the project; when the debugger reaches *main()*, it will stop
- Start HyperTerminal
- Select *Run* in the debugger to continue executing the Serial Data Flash Programmer
- The terminal will display a menu:
(c) 2002 Motorola Inc. 5685x Serial Data Flash Programmer. Version 1.0

Select :

1. Read 2nd Stage Bootloader S-Record and Store in Data Flash
 2. Read Application S-Record and Store in Data Flash
 3. Exit
- >

- Press 1
- The terminal will display:
Waiting for 2nd Stage Bootloader S-Record.
- Select HyperTerminal menu *Transfer\Send Text File*
- Select the file:
`...\nos\applications\bootloaders\2nd_stage_bootloader\Debug\InternalRAM.elf.S`
- Wait for next HyperTerminal message:
Loaded 0x01c9 Program and 0x0000 Data words.

(c) 2002 Motorola Inc. 5685x Serial Data Flash Programmer. Version 1.0

Select :

1. Read 2nd Stage Bootloader S-Record and Store in Data Flash
 2. Read Application S-Record and Store in Data Flash
 3. Exit
- >

- The Second Stage SPI Bootloader is now programmed into the SPI Data Flash
- To program the user application into SPI Data Flash, press 2
- The terminal will display:
Waiting for Application S-Record.
- Select HyperTerminal menu *Transfer\Send Text File*
- Select the COP demo S-Record
 - For this example, use:
`...\nos\applications\bsp\cop\Debug\InternalRAM.elf.S`
 - If this file does not exist, open the *cop.mcp* project
 - Change the project settings to generate an S-Record
 - Rebuild the *cop.mcp* project
- Wait for the next HyperTerminal message:
Loaded 0x0854 Program and 0x0074 Data words.

(c) 2002 Motorola Inc. 5685x Serial Data Flash Programmer. Version 1.0

Select :

1. Read 2nd Stage Bootloader S-Record and Store in Data Flash
 2. Read Application S-Record and Store in Data Flash
 3. Exit
- >

- The COP Demo is now programmed into the SPI Data Flash
- To exit the Serial Data Flash Programmer, press 3
- Kill the debugger session for the Serial Data Flash Programmer
- Disconnect the Parallel Port Cable from the DSP56852EVM
- Remove power from the DSP56852EVM
- Reapply power to the DSP56852EVM
- The LEDs should be blinking
- Press the IRQA Button to prevent the COP from resetting

5.2 GPIO Driver

5.2.1 Introduction

The General Purpose I/O (GPIO) interface manipulates external signals routed through general purpose pins. Typically, each pin may be may be programmed as an input, output, or level-sensitive interrupt input. However, peripherals may share control of these general purpose I/O pins; you may not use a pin as both a peripheral and as a general purpose I/O pin. Therefore, please consult the **DSP56852 16-Bit Digital Signal Processor User's Manual** to determine which pins are assigned to peripherals that you will use, and which pins may be available for general purpose use.

For DSP56852:

The DSP56852 has no dedicated GPIO and has 11 multiplexed GPIOs on Ports A, C, E.

5.2.2 Initialization

The GPIO driver currently has no default configurations that can be overwritten through a project's *appconfig.h* file.

5.2.3 API Definition

This section defines the Application Programming Interface (API) by identifying all public interface functions and data structures.

Code to create and initialize the GPIO driver may be automatically included in an SDK Embedded Project by inserting the following line in the *appconfig.h* file:

```
#define INCLUDE_GPIO
```

The following information may be found in the public header file *gpio.h*.

Public Interface Function(s):

```

/*****
*      types_tHandle open(const char *pName, int OFlags, NULL);
*      int      close(types_tHandle FileDesc);
*      UWord16 ioctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams);
***/
*      types_tHandle gpioOpen(const char *pName, int OFlags, NULL);
*      int      gpioClose(types_tHandle FileDesc);
*      UWord16 gpioIoctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams,
*                          const char* gpioDevice);
*****/

```

Public Data Structure(s):

None

Members:

None



5.2.4 GPIO Device-Independent I/O API Specifications

The following pages specify the device-independent Application Programming Interface (API).

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

5.2.4.1 *open*

Call(s):

```
types_tHandle open(const char *pName, int OFlags, NULL);
```

Arguments:
Table 5-2. GPIO Driver Arguments - *open*

<i>pName</i>	<i>in</i>	The GPIO device name; see Table 5-3
<i>OFlag</i>	<i>in</i>	Open Mode Flags which are ignored
<i>FileDesc</i>	<i>out</i>	GPIO device descriptor returned by <i>open</i> call and used in <i>ioctl</i> function calls

Table 5-3. GPIO Ports Available for the DSP56852 Chip

Chip	Can Use
DSP56852	BSP_DEVICE_NAME_GPIO_A BSP_DEVICE_NAME_GPIO_C BSP_DEVICE_NAME_GPIO_E

Description: The *open* call opens the GPIO driver for operations and returns a file descriptor used by *ioctl* function calls. The GPIO driver must be opened for *ioctl* calls.

Returns: The *open* call returns a type *types_tHandle*. When the *open* call is successful, this *types_tHandle* is a file descriptor for the GPIO driver and is used by the GPIO driver's *close* and *ioctl* functions. A "-1" is returned when the *open* call is not successful.

Code Example: See [Code Example 5-4](#).



5.2.4.2 *close*

Call(s):

```
int close (types_tHandle FileDesc);
```

Arguments:

Table 5-4. GPIO Driver Arguments - *close*

<i>FileDesc</i>	<i>in</i>	GPIO device descriptor returned by <i>open</i> call and used in <i>ioctl</i> function calls

Description: The *close* call closes the GPIO driver.

Returns: The *close* call returns a type *int*; this *int* is always zero.

Code Example: See [Code Example 5-4](#).

5.2.4.3 *ioctl*

Call(s):

```
UWord16 ioctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams);
```

Arguments:

Table 5-5. GPIO Driver Arguments - *ioctl*

<i>FileDesc</i>	<i>in</i>	GPIO device descriptor returned by <i>open</i> call and used in <i>ioctl</i> function calls
<i>Cmd</i>	<i>in</i>	Commands found in <i>gpio.h</i> which are used to modify GPIO pins
<i>pParams</i>	<i>in</i>	<i>gpioPin(Port,Pin)</i> macro defined in <i>gpio.h</i> . This macro can be ORed with itself as long as the port is the same; used to pass on a particular pin(s) on a port in which to perform one of the commands in Table 5-6 . The <i>gpioPin</i> macro defined in Code Example 5-4 is used to obtain a mask for that particular pin and port.

Table 5-6. GPIO *ioctl* Commands

Cmd	<i>pParams</i>	Return	Result
GPIO_SETAS_GPIO	<i>gpioPin(Port,Pin)</i> macro	None	When peripheral is disabled, DDR determines direction of data flow in PER register
GPIO_SETAS_PERIPHERAL	<i>gpioPin(Port,Pin)</i> macro	None	A peripheral masters the GPIO pin, in PER register
GPIO_SETAS_INPUT	<i>gpioPin(Port,Pin)</i> macro	None	Sets a GPIO pin as an input, in DDR register
GPIO_SETAS_OUTPUT	<i>gpioPin(Port,Pin)</i> macro	None	Sets a GPIO pin as an output, in DDR register
GPIO_DISABLE_PULLUP	<i>gpioPin(Port,Pin)</i> macro	None	Disables pull-up, in PUR register
GPIO_ENABLE_PULLUP	<i>gpioPin(Port,Pin)</i> macro	None	Enables pull-up, in PUR register
GPIO_SET	<i>gpioPin(Port,Pin)</i> macro	None	Sets a GPIO signal
GPIO_CLEAR	<i>gpioPin(Port,Pin)</i> macro	None	Clears a GPIO signal
GPIO_TOGGLE	<i>gpioPin(Port,Pin)</i> macro	None	Toggles a GPIO signal
GPIO_READ	<i>gpioPin(Port,Pin)</i> macro	0 or 1	Reads the value of an input pin Returns 0 or 1 for the value

Description: The *ioctl* call changes the state of GPIO pins.

Returns: The *ioctl* call returns a type *UWord16*, which is always zero.



Code Example: For the DSP56852, [Code Example 5-4](#) toggles Port A, pin 0, between a logic high and logic low.

Code Example 5-4. GPIO Driver Usage

```

/*****
#include "io.h"
#include "fcntl.h"
#include "bsp.h"

#include "gpio.h"

types_tHandle PortA;

PortA = open(BSP_DEVICE_NAME_GPIO_A, 0, NULL);

/* disable peripheral as the master of pin 0 and 1 on port A */
ioctl(PortA, GPIO_SETAS_GPIO, (gpioPin(A,0) | gpioPin(A,1));

/* set pin 0 and 1 on port A as an output pin, */
ioctl(PortA, GPIO_SETAS_OUTPUT, (gpioPin(A,0) | gpioPin(A,1));

/* toggle pin 0 on port A */
ioctl(PortA, GPIO_TOGGLE, gpioPin(A,0));

close(PortA);
*****/

```



5.2.5 GPIO Low-Level I/O API Specifications

The following pages specify the low-level Application Programming Interface (API).

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



5.2.5.1 gpioOpen

Call(s):

```
types_tHandle gpioOpen(const char *pName, int OFlags, NULL);
```

Table 5-7. GPIO Driver Arguments - gpioOpen

<i>pName</i>	<i>in</i>	The GPIO device name; see Table 5-3
<i>OFlag</i>	<i>in</i>	Open Mode Flags which are ignored

Description: Opens a particular port for operations; argument *pName* is the particular port’s name. The port must be opened before configuring the port with *gpioIoctl* calls.

Returns: If *gpioOpen* is successful, a port file descriptor is returned. If *gpioOpen* failed, a “-1” is returned.



5.2.5.2 *gpioClose*

Call(s):

```
int gpioClose (types_tHandle FileDesc);
```

Table 5-8. GPIO Driver Arguments - *close*

<i>FileDesc</i>	<i>in</i>	GPIO device descriptor returned by <i>gpioOpen</i> call and used in <i>gpioctl</i> function calls

Description: The *close* call closes the GPIO device.

Returns: The *close* call returns a type *int*. This *int* is always zero.

5.2.5.3 *gpioIoctl*

Call(s):

```
UWord16 gpioIoctl(types_tHandle FileDesc, UWord16 Cmd, UWord16 Params, const
char*
                    gpioDevice);
```

Table 5-9. GPIO Driver Arguments

<i>FileDesc</i>	<i>in</i>	GPIO device descriptor returned by <i>open</i> call and used in <i>ioctl</i> function calls
<i>Cmd</i>	<i>in</i>	Commands found in <i>gpio.h</i> which are used to modify GPIO pins; see Table 5-10
<i>Params</i>	<i>in</i>	<i>gpioPin(Port,Pin)</i> macro defined in <i>gpio.h</i> . This macro can be ORed with itself as long as the Port is the same; used to pass on a particular pin(s) on a port in which to perform one of the above commands. The <i>gpioPin</i> macro defined in Code Example 5-5 is used to obtain a mask for that particular pin and port.
<i>gpioDevice</i>	<i>in</i>	The GPIO device name; see Table 5-3

Table 5-10. *gpioIoctl* Commands

Cmd	<i>pParams</i>	Return	Result
GPIO_SETAS_GPIO	<i>gpioPin(Port,Pin)</i> macro	None	When peripheral is disabled, DDR determines direction of data flow in PER register
GPIO_SETAS_PERIPHERAL	<i>gpioPin(Port,Pin)</i> macro	None	A peripheral masters the GPIO pin, in PER register
GPIO_SETAS_INPUT	<i>gpioPin(Port,Pin)</i> macro	None	Sets a GPIO pin as an input, in DDR register
GPIO_SETAS_OUTPUT	<i>gpioPin(Port,Pin)</i> macro	None	Sets a GPIO pin as an output, in DDR register
GPIO_DISABLE_PULLUP	<i>gpioPin(Port,Pin)</i> macro	None	Disables pull-up, in PUR register
GPIO_ENABLE_PULLUP	<i>gpioPin(Port,Pin)</i> macro	None	Enables pull-up, in PUR register
GPIO_SET	<i>gpioPin(Port,Pin)</i> macro	None	Sets a GPIO signal
GPIO_CLEAR	<i>gpioPin(Port,Pin)</i> macro	None	Clears a GPIO signal
GPIO_TOGGLE	<i>gpioPin(Port,Pin)</i> macro	None	Toggles a GPIO signal
GPIO_READ	<i>gpioPin(Port,Pin)</i> macro	0 or 1	Reads the value of an input pin Returns 0 or 1 for the value

Description: The *gpioIoctl* modifies a GPIO port configuration or sets a GPIO pin.

Returns: The *gpioIoctl* call returns an *int* value.

Code Example: See [Code Example 5-5](#).

Code Example 5-5. GPIO Driver Usage

```

/*****
#include "bsp.h"
#include "gpio.h"

void main(void)
{
    types_tHandle PortA;
    int state;

    // open a gpio port A
    PortA = gpioOpen(BSP_DEVICE_NAME_GPIO_A,0);

    // disable peripheral as the master of pins 0 and 1 on port A */
    gpioIoctl (PortA, GPIO_SETAS_GPIO, (gpioPin(A,0) | gpioPin(A,1)),
               BSP_DEVICE_NAME_GPIO_A);

    // set pins 0 and 1 on port A as an output pin, */
    gpioIoctl(PortA, GPIO_SETAS_OUTPUT, (gpioPin(A,0) | gpioPin(A,1)),
               BSP_DEVICE_NAME_GPIO_A);

    // read the state of port A pin 1
    state = gpioIoctl (PortA, GPIO_READ, gpioPin(A,1), BSP_DEVICE_NAME_GPIO_A);

    gpioClose(PortA);
}
*****/

```

5.2.6 GPIO Driver Application

There are no applications that specifically test the GPIO Driver. However, many applications and drivers use the GPIO driver, including the LED and Codec drivers and the Codec, Timer, and Quad Timer applications.

5.3 Programmable Chip Select (PCS) Driver

The DSP56852 has four programmable chip selects, PCS0 - PCS3, which may be defined by the user to map hardware peripherals into the address space of the DSP. The PCS driver provides a means to initialize these chip selects.

5.3.1 Initialization

The SDK start-up code can automatically initialize chip selects and EMI (BCR). Add the following line to the *appconfig.h* file associated with the project created in CodeWarrior with Embedded SDK stationery:

```
#define INCLUDE_PCS
```

The PCS driver will be called to initialize the chip selects and EMI (BCR) based on the default chip select parameters found in *config.h*. These defaults may be redefined in the *appconfig.h* file to support the customization of target hardware. The defaults in the *config.h* file are valid for using the external Program and Data RAM on the DSP56852 EVM. Chip Select 0 is mapped to both bytes of Program space address range \$00000-\$1FFFF; Chip Select 1 is mapped to the lower byte of Data space address range \$00000-\$1FFFF; and Chip Select 2 is mapped to the upper byte of Data space address range \$00000-\$1FFFF. Chip Select 3 is disabled. The EMI (BCR) is defaulted to the hardware default.

```
#define PCS0_BASE_ADDR 0x0000
#define PCS0_BLKSIZE PCS_BLKSIZE_128K
#define PCS0_SPEN PCS_PROGRAM
#define PCS0_RWEN PCS_READ_WRITE_ENABLE
#define PCS0_BYTE_EN PCS_BOTH_BYTES_ENABLE
#define PCS0_RWS PCS_2_WAIT_STATES
#define PCS0_WWS PCS_2_WAIT_STATES
#define PCS0_WSS PCS_0_WAIT_STATES
#define PCS0_WSH PCS_0_WAIT_STATES
#define PCS0_RWSS PCS_0_WAIT_STATES
#define PCS0_RWSH PCS_0_WAIT_STATES
#define PCS0_MDAR PCS_0_WAIT_STATES

#define PCS1_BASE_ADDR 0x0000
#define PCS1_BLKSIZE PCS_BLKSIZE_128K
#define PCS1_SPEN PCS_DATA
#define PCS1_RWEN PCS_READ_WRITE_ENABLE
#define PCS1_BYTE_EN PCS_LOWER_BYTE_ENABLE
#define PCS1_RWS PCS_2_WAIT_STATES
#define PCS1_WWS PCS_2_WAIT_STATES
#define PCS1_WSS PCS_0_WAIT_STATES
#define PCS1_WSH PCS_0_WAIT_STATES
#define PCS1_RWSS PCS_0_WAIT_STATES
#define PCS1_RWSH PCS_0_WAIT_STATES
#define PCS1_MDAR PCS_0_WAIT_STATES

#define PCS2_BASE_ADDR 0x0000
#define PCS2_BLKSIZE PCS_BLKSIZE_128K
#define PCS2_SPEN PCS_DATA
#define PCS2_RWEN PCS_READ_WRITE_ENABLE
```



```
#define PCS2_BYTE_EN PCS_UPPER_BYTE_ENABLE
#define PCS2_RWS PCS_2_WAIT_STATES
#define PCS2_WWS PCS_2_WAIT_STATES
#define PCS2_WSS PCS_0_WAIT_STATES
#define PCS2_WSH PCS_0_WAIT_STATES
#define PCS2_RSS PCS_0_WAIT_STATES
#define PCS2_RSH PCS_0_WAIT_STATES
#define PCS2_MDAR PCS_0_WAIT_STATES

#define PCS3_BASE_ADDR 0x0000
#define PCS3_BLKSIZE PCS_BLKSIZE_32K
#define PCS3_SPEN PCS_DISABLE
#define PCS3_RWEN PCS_READ_WRITE_DISABLE
#define PCS3_BYTE_EN PCS_NO_BYTE_ENABLE
#define PCS3_RWS PCS_23_WAIT_STATES
#define PCS3_WWS PCS_23_WAIT_STATES
#define PCS3_WSS PCS_0_WAIT_STATES
#define PCS3_WSH PCS_0_WAIT_STATES
#define PCS3_RSS PCS_0_WAIT_STATES
#define PCS3_RSH PCS_0_WAIT_STATES
#define PCS3_MDAR PCS_3_WAIT_STATES

#define PCS_DRV PCS_DRV_DISABLED
#define PCS_BMDAR PCS_0_WAIT_STATES
#define PCS_BWWS PCS_18_WAIT_STATES
#define PCS_BRWS PCS_28_WAIT_STATES
```



5.3.2 API Definition

The following pages specify the Application Programming Interface (API).

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

5.3.2.1 *pcsInit()*

Call(s):

```
void pcsInit(ChipSelect, BaseAddress, BlockSize, SpaceEnable, RWEnable,
             ByteEnable, ReadWaitStates, WriteWaitStates,
             WriteWaitStatesSetupDelay, WriteWaitStatesHoldDelay,
             ReadWaitStatesSetupDelay, ReadWaitStatesHoldDelay,
             MinDelayAfterRead);
```

Arguments:

Table 5-11. PCS Argument Values

Argument	Possible Value
<i>ChipSelect</i>	Identifies the Chip Select to be initialized PCS_CS0 PCS_CS1 PCS_CS2 PCS_CS3
<i>BaseAddress</i>	Start address, which must be a multiple of the selected block size. For example, if a 4K block size is selected, then the start address can only be \$0000, \$1000, \$2000, ...
<i>BlockSize</i>	One of the predefined block sizes; sets end address PCS_BLKSIZE_4K PCS_BLKSIZE_8K PCS_BLKSIZE_16K PCS_BLKSIZE_32K PCS_BLKSIZE_64K PCS_BLKSIZE_128K PCS_BLKSIZE_256K PCS_BLKSIZE_512K PCS_BLKSIZE_1M PCS_BLKSIZE_2M PCS_BLKSIZE_4M PCS_BLKSIZE_8M PCS_BLKSIZE_16M
<i>SpaceEnable</i>	Selects the type of memory (program, data, or both) used for asserting the appropriate strobe signal PCS_DISABLE PCS_DATA PCS_PROGRAM Note: To specify both data and program: (PCS_DATA PCS_PROGRAM)
<i>RWEnable</i>	Identifies the type of access required in order to assert the Chip Select PCS_READ_WRITE_DISABLE PCS_WRITE_ONLY PCS_READ_ONLY PCS_READ_WRITE_ENABLE



Table 5-11. PCS Argument Values (Continued)

Argument	Possible Value
<i>ByteEnable</i>	Selects which Byte(s) to access when accessing the device controlled by the Chip Select PCS_NO_BYTE_ENABLE PCS_LOWER_BYTE_ENABLE PCS_UPPER_BYTE_ENABLE PCS_BOTH_BYTES_ENABLE
<i>ReadWaitStates</i>	Specifies the number of additional system clocks to delay for a read access to the selected memory PCS_0_WAIT_STATES PCS_1_WAIT_STATE PCS_2_WAIT_STATES ... PCS_(3-29)_WAIT_STATES PCS_30_WAIT_STATES
<i>WriteWaitStates</i>	Specifies the number of additional system clocks to delay for a write access to the selected memory PCS_0_WAIT_STATES PCS_1_WAIT_STATE PCS_2_WAIT_STATES ... PCS_(3-29)_WAIT_STATES PCS_30_WAIT_STATES
<i>WriteWaitStatesSetupDelay</i>	Specifies the number of system clocks to delay prior to asserting write signal PCS_0_WAIT_STATES PCS_1_WAIT_STATE PCS_2_WAIT_STATES ... PCS_(3-29)_WAIT_STATES PCS_30_WAIT_STATES
<i>WriteWaitStatesHoldDelay</i>	Specifies the number of system clocks to delay holding the address, data, and Chip Select signals after deasserting write signal PCS_0_WAIT_STATES PCS_1_WAIT_STATE PCS_2_WAIT_STATES ... PCS_(3-29)_WAIT_STATES PCS_30_WAIT_STATES
<i>ReadWaitStatesSetupDelay</i>	Specifies the number of system clocks to delay prior to asserting read signal PCS_0_WAIT_STATES PCS_1_WAIT_STATE PCS_2_WAIT_STATES ... PCS_(3-29)_WAIT_STATES PCS_30_WAIT_STATES
<i>ReadWaitStatesHoldDelay</i>	Specifies the number of system clocks to delay holding the address, data, and Chip Select signals after deasserting read signal PCS_0_WAIT_STATES PCS_1_WAIT_STATE PCS_2_WAIT_STATES ... PCS_(3-29)_WAIT_STATES PCS_30_WAIT_STATES

Table 5-11. PCS Argument Values (Continued)

Argument	Possible Value
<i>MinDelayAfterRead</i>	Specifies the number of system clocks to delay between reading from memory in a Chip Select controlled space and reading from another device PCS_0_WAIT_STATES PCS_1_WAIT_STATE PCS_2_WAIT_STATES ... PCS_(3-6)_WAIT_STATES PCS_7_WAIT_STATES

Description: This macro will initialize the Chip Select.

Returns: None

Code Examples: Chip Select 3 can be set up to assert for address range \$6000 - \$6FFF only for data space accesses; two wait states for read and write, read and write both enabled; both bytes enabled; 0 wait states delay for hold and setup times; and 0 wait states for minimal delay after read. A one-time configuration through the SDK start-up code requires that the code in [Code Example 5-6](#) be added to the *appconfig.h* file.

Code Example 5-6. *appconfig.h* file/Chip Select 3

```

appconfig.h:
    #define INCLUDE_PCS

    #define PCS3_BASE_ADDR 0x6000
    #define PCS3_BLKSIZE PCS_BLKSIZE_4K
    #define PCS3_SPEN PCS_DATA
    #define PCS3_RWEN PCS_READ_WRITE_ENABLE
    #define PCS3_BYTE_EN PCS_BOTH_BYTES_ENABLE
    #define PCS3_RWS PCS_2_WAIT_STATES
    #define PCS3_WWS PCS_2_WAIT_STATES
    #define PCS3_WSS PCS_0_WAIT_STATES
    #define PCS3_WSH PCS_0_WAIT_STATES
    #define PCS3_RWSS PCS_0_WAIT_STATES
    #define PCS3_RWSH PCS_0_WAIT_STATES
    #define PCS3_MDAR PCS_0_WAIT_STATES
    
```

If desired, the function can be called from the application to change the chip select. The code in [Code Example 5-7](#) will set up Chip Select 3 to assert for address range \$7000 to \$7FFF for data and program space accesses; both read and write enabled; 11 wait states for read and write; both bytes enabled; 0 wait states delay for hold and setup times; and 0 wait states for minimal delay after read.

**Code Example 5-7. main.c file/Chip Select 3**

```
main.c:
#include "pcs.h"

void main(void)
{
    pcsInit(PCS_CS3, 0x7000, PCS_BLKSIZE_4K, PCS_DATA|PCS_PROGRAM,
            PCS_READ_WRITE_ENABLE, PCS_BOTH_BYTES_ENABLE,
            PCS_11_WAIT_STATES, PCS_11_WAIT_STATES,
            PCS_0_WAIT_STATES, PCS_0_WAIT_STATES,
            PCS_0_WAIT_STATES, PCS_0_WAIT_STATES,
            PCS_0_WAIT_STATES);
}
```

5.3.2.2 *pcsInitEMI()*

Call(s):

```
void pcsInitEMI(DriveBit, BaseMinDelayAfterRead, BaseWriteWaitStates,
                BaseReadWaitStates);
```

Arguments:

Table 5-12. PCS Argument Values

Argument	Possible value
<i>DriveBit</i>	Enables or Disables the Drive Bit. Enabling drives the external memory port pins when not in use. Disabling tri-states the memory port pins when not in use. PCS_DRV_DISABLED PCS_DRV_ENABLED
<i>BaseMinDelayAfterRead</i>	Specifies the number of system clocks to delay between reading from memory that is not in a Chip Select-controlled space. PCS_0_WAIT_STATES PCS_1_WAIT_STATE PCS_2_WAIT_STATES ... PCS_(3-6)_WAIT_STATES PCS_7_WAIT_STATES
<i>BaseWriteWaitStates</i>	Specifies the number of additional system clocks to delay for a write access to the selected memory when the memory address does not fall within a Chip Select-controlled range. PCS_0_WAIT_STATES PCS_1_WAIT_STATE PCS_2_WAIT_STATES ... PCS_(3-29)_WAIT_STATES PCS_30_WAIT_STATES
<i>BaseReadWaitStates</i>	Specifies the number of additional system clocks to delay for a read access to the selected memory when the memory address does not fall within a Chip Select-controlled range PCS_0_WAIT_STATES PCS_1_WAIT_STATE PCS_2_WAIT_STATES ... PCS_(3-29)_WAIT_STATES PCS_30_WAIT_STATES

Description: This macro will initialize the EMI for those external memory addresses that do not fall within a Chip Select range.

Returns: None

Code Examples: The following example enables the Drive Bit and sets the base wait states to 0. A one-time configuration through the SDK start-up code requires that the code in [Code Example 5-8](#) be added to the *appconfig.h* file.

Code Example 5-8. *appconfig.h* file/EMI

```
appconfig.h:
#define INCLUDE_PCS

#define PCS_DRV PCS_DRV_ENABLED
#define PCS_BMDAR PCS_0_WAIT_STATES
#define PCS_BWWS PCS_0_WAIT_STATES
#define PCS_BRWS PCS_0_WAIT_STATES
```

If desired, the function can be called from the application to change the EMI (BCR). The code in [Code Example 5-9](#) will enable the Drive Bit and set the base wait states to 0.

Code Example 5-9. *main.c* file/Chip Select 3

```
main.c:
#include "pcs.h"

void main(void)
{
    pcsInitEMI(PCS_DRV_ENABLED, PCS_0_WAIT_STATES, PCS_0_WAIT_STATES,
               PCS_0_WAIT_STATES);
}
```

5.4 Quad Timer Driver

5.4.1 Introduction

The DSP56852 processor has one Quad Timer module, Quad Timer A. Within this Timer module is a set of four timers that may be configured and used individually. Please refer to **Chapter 13** of the **DSP56852 16-Bit Digital Signal Processor User's Manual** for information on the available configuration settings for the Quad Timer.

The Quad Timer driver provides a software interface that simplifies the configuration and use of this peripheral. This driver is also used by the SDK Timer Services described in the **SDK Programmer's Guide** and therefore requires some configuration and management of timer resources especially when both are required by the application.

5.4.2 Initialization

The Quad Timer driver has a default configuration that is defined in *config.h*. These default configuration settings can be redefined in the *appconfig.h* for a specific project.

A different set of defaults are defined based on the user's selections in the *appconfig.h* file. It may be configured for user definition by defining `INCLUDE_QUAD_TIMER` and configured for timer services by defining `INCLUDE_TIMER`. See the **SDK Programmer's Guide** for definition and specification of the Timer Services interface.

If the Quad Timer is being used in the absence of the Timer Services, by default, all four timers are configured for direct use with the Quad Timer driver, as shown in **Code Example 5-10**.

Code Example 5-10. Timer Configuration

```
#if \
    !(defined(INCLUDE_USER_TIMER_A_0) || defined(INCLUDE_USER_TIMER_A_1) || \
      defined(INCLUDE_USER_TIMER_A_2) || defined(INCLUDE_USER_TIMER_A_3))
/*
    Default set of timers configured for user definition. This occurs if the
    user has not specified any timers with the following definition in the
    appconfig.h file:
    #define INCLUDE_QUAD_TIMER
    #undef INCLUDE_TIMER
*/
    #if !defined(INCLUDE_TIMER)
        #define INCLUDE_USER_TIMER_A_0 0
        #define INCLUDE_USER_TIMER_A_1 0
        #define INCLUDE_USER_TIMER_A_2 0
        #define INCLUDE_USER_TIMER_A_3 0
    #endif
#endif

#endif
```

The defaults may be overridden if the user specifies the exact timers required by the application. This will minimize the code size of the driver when all timers are not required. A list of available timers is found in [Table 5-13](#).

Table 5-13. Quad Timers

Timer	State
INCLUDE_USER_TIMER_A_0	Defaults as a Quad Timer when INCLUDE_QUAD_TIMER is defined and INCLUDE_TIMER is not defined
INCLUDE_USER_TIMER_A_1	Defaults as a Quad Timer when INCLUDE_QUAD_TIMER is defined and INCLUDE_TIMER is not defined
INCLUDE_USER_TIMER_A_2	Defaults as a Quad Timer when INCLUDE_QUAD_TIMER is defined and INCLUDE_TIMER is not defined
INCLUDE_USER_TIMER_A_3	Defaults as a Quad Timer when INCLUDE_QUAD_TIMER is defined and INCLUDE_TIMER is not defined

As an example, if timers A0 and A2 are required, they can be specified by adding the following text to the *appconfig.h* file in the project created with SDK Embedded Stationery.

```
#define INCLUDE_QUAD_TIMER

#define INCLUDE_USER_TIMER_A_0 0 /* Timer A0 driver */
#define INCLUDE_USER_TIMER_A_2 0 /* Timer A2 driver */
```

NOTE: If definitions are declared as a “0”, this timer will be used by the Quad Timer driver. If definitions are declared as a “1”, this timer will be used by the Timer Services driver. An example follows:

```
#define INCLUDE_USER_TIMER_A_0 0 /* Quad Timer driver */
#define INCLUDE_USER_TIMER_A_3 1 /* Timer Services driver */
```

5.4.3 API Definition

This section defines the Application Programming Interface (API) by identifying all public interface functions and data structures.

Code to create and initialize the Quad Timer driver may be automatically included in an SDK Embedded Project by inserting the following line in the *appconfig.h* file:

```
#define INCLUDE_QUAD_TIMER
```

The following information may be found in the public header file *quadtimer.h*:

Public Interface Function(s):

```
/* ***** */
* types_tHandle open(const char *pName, int OFlags, qt_sState * pParams);
* int close(types_tHandle FileDesc);
* Uword16 ioctl(types_tHandle FileDesc, Uword16 Cmd, void * pParams);
/* ***** */
```



```
* types_tHandle qtOpen(const char *pName, int OFlags, qt_sState * pParams);
* int qtClose(types_tHandle FileDesc);
* UWord16 qtIoctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams,
* void * bspDeviceName);
*****/
```

Public Data Structure(s):

```
typedef struct
{
    qt_eMode Mode :4;

    qt_eInputSource InputSource :4;
    qt_ePolarity InputPolarity :1;
    qt_eSecondaryInputSource SecondaryInputSource :2;

    qt_eCountFrequency CountFrequency :1;
    qt_eCountLength CountLength :1;
    qt_eCountDirection CountDirection :1;

    qt_eOutputMode OutputMode :3;
    qt_ePolarity OutputPolarity :1;
    bool OutputDisabled :1;

    bool Master :1;
    bool OutputOnMaster :1;
    bool CoChannelInitialize :1;
    bool AssertWhenForced :1;

    qt_eCaptureMode CaptureMode :2;

    UInt16 CompareValue1;
    UInt16 CompareValue2;
    UInt16 InitialLoadValue;

    qt_sCallback CallbackOnCompare;
    qt_sCallback CallbackOnOverflow;
    qt_sCallback CallbackOnInputEdge;

}qt_sState;
```

Table 5-14. Quad Timer Data Structure Members

Parameter	Type	Description	Quad Timer Register Reference
Mode	qt_eMode	Quad Timer count mode Possible values are: <i>qtCount</i> , <i>qtCountBothEdges</i> , <i>qtGatedCount</i> , <i>qtQuadratureCount</i> , <i>qtSignedCount</i> , <i>qtTriggeredCount</i> , <i>qtCascadeCount</i> , <i>qtOneShot</i> , <i>qtPulseOutput</i> , <i>qtFixedFreqPWM</i> , <i>qtVariableFreqPWM</i>	Counting options, count code of control register
InputSource	qt_eInputSource	Primary count source type Possible values are: <i>qtCounter0Input</i> , <i>qtCounter1Input</i> , <i>qtCounter2Input</i> , <i>qtCounter3Input</i> , <i>qtCounter0Output</i> , <i>qtCounter1Output</i> , <i>qtCounter2Output</i> , <i>qtCounter3Output</i> , <i>qtPrescalerDiv1</i> , <i>qtPrescalerDiv2</i> , <i>qtPrescalerDiv4</i> , <i>qtPrescalerDiv8</i> , <i>qtPrescalerDiv16</i> , <i>qtPrescalerDiv32</i> , <i>qtPrescalerDiv64</i> , <i>qtPrescalerDiv128</i>	Primary count source of control register
InputPolarity	qt_ePolarity	Input polarity Possible values are: <i>qtNormal</i> , <i>qtInverted</i>	Input polarity select of status and control register
SecondaryInputSource	qt_eSecondaryInputSource	Secondary count source type Possible values are: <i>qtSISCounter0Input</i> , <i>qtSISCounter1Input</i> , <i>qtSISCounter2Input</i> , <i>qtSISCounter3Input</i>	Secondary input source of control register
CountFrequency	qt_eCountFrequency	To select continuous or one shot counting mode Possible values are: <i>qtRepeatedly</i> , <i>qtOnce</i>	Count once of control register

Table 5-14. Quad Timer Data Structure Members (Continued)

Parameter	Type	Description	Quad Timer Register Reference
CountLength	qt_eCountLength	Re-initialize after a compare or do not re-initialize after a compare Possible values are: <i>qtPastCompare</i> , <i>qtUntilCompare</i>	Count length of control register
CountDirection	qt_eCountDirection	Normal or reverse direction Possible values are: <i>qtUp</i> , <i>qtDown</i>	Count direction of control register
OutputMode	qt_eOutputMode	Output mode Possible values are: <i>qtAssertWhileActive</i> , <i>qtAssertOnCompare</i> , <i>qtDeassertOnCompare</i> , <i>qtToggleOnCompare</i> , <i>qtToggleUsingAlternateCompare</i> , <i>qtDeassertOnSecondary</i> , <i>qtDeassertOnCounterRollover</i> , <i>qtAssertOnGatedClock</i>	Output mode of control register
OutputPolarity	qt_ePolarity	Output polarity Possible values are: <i>qtNormal</i> , <i>qtInverted</i>	Output polarity select of status and control register
OutputDisabled	bool	Enable ("True")/disable ("False") output	Output enable of status and control register
Master	bool	Master mode ("True")	Master mode of status and control register
OutputOnMaster	bool	Enable as a master to force the state of counter by OFLAG ("True")	Enable external OFLAG force
CoChannellInitialize	bool	Enable another counter/force the initialization ("True")	Co-channel initialization of status register
CaptureMode	qt_eCaptureMode	Counter capture mode: <i>qtDisabled</i> , <i>qtRisingEdge</i> , <i>qtFallingEdge</i> , <i>qtBothEdges</i>	Input capture mode of status and control register
CompareValue1	UInt16	Compare value 1	Compare register 1
CompareValue2	UInt16	Compare value 2	Compare register 2

Table 5-14. Quad Timer Data Structure Members (Continued)

Parameter	Type	Description	Quad Timer Register Reference
InitialLoadValue	UInt16	Initial load value	Counter register
CallbackOnCompare	qt_sCallback	User callback on compare	See Callback Functions
CallbackOnOverflow	qt_sCallback	User callback on overflow	See Callback Functions
CallbackOnInputEdge	qt_sCallback	User callback on input edge	See Callback Functions

Callback Functions:

The Quad driver's callback functions allow functions created by the user in his application to be called as a result of Quad Timer-specific interrupts. More information about these Quad Timer-specific interrupts can be found in **Chapter 13** of the **DSP56852 16-Bit Digital Signal Processor User's Manual**.

The three types of callback functions that the Quad Timer driver allows include:

- A callback on compare
- A callback on overflow
- A callback on an input edge

To utilize the callback functions, the functions must be passed into the *open* or *ioctl* functions via the *qt_sState* data structure. An example is shown in [Code Example 5-11](#), where the user has defined a callback function, *CallbackOnCompare0*, with input argument 0.

Code Example 5-11. qt_sState Data Structure

```
const qt_sState quadParam0 =
{
    /* Mode = */ qtCount,
    /* InputSource = */ qtPrescalerDiv128,
    /* InputPolarity = */ qtNormal,
    /* SecondaryInputSource = */ 0,
    /* CountFrequency = */ qtOnce,
    /* CountLength = */ qtUntilCompare,
    /* CountDirection = */ qtDown,
    /* OutputMode = */ qtAssertWhileActive,
    /* OutputPolarity = */ qtNormal,
    /* OutputDisabled = */ 1,
    /* Master = */ 0,
    /* OutputOnMaster = */ 0,
    /* CoChannelInitialize = */ 0,
    /* AssertWhenForced = */ 0,
    /* CaptureMode = */ qtDisabled,
    /* CompareValue1 = */ 0x8000,
```



ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

```
/* CompareValue2 = */ 0,  
/* InitialLoadValue = */ 0xFFFF,  
/* CallbackOnCompare = */ {CallbackOnCompare0, 0},  
/* CallbackOnOverflow = */ {0, 0},  
/* CallbackOnInputEdge = */ {0, 0}  
};
```

Because of an order from the United States International Trade Commission, BGA-packaged product lines and part numbers indicated here currently are not available from Freescale for import or sale in the United States prior to September 2010: DSP56852VFE



5.4.4 Quad Timer Device-Independent I/O API Specifications

The following pages specify the device-independent Application Programming Interface (API).

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

5.4.4.1 *open*

Call(s):

```
types_tHandle open(const char *pName, int OFlags, qt_sState * pParams);
```

Arguments:

Table 5-15. Quad Driver Arguments - *open*

<i>pName</i>	<i>in</i>	The quad timer device name: BSP_DEVICE_NAME_QUAD_TIMER_A_0 BSP_DEVICE_NAME_QUAD_TIMER_A_1 BSP_DEVICE_NAME_QUAD_TIMER_A_2 BSP_DEVICE_NAME_QUAD_TIMER_A_3
<i>OFlag</i>	<i>in</i>	Open Mode Flags; use "0" because <i>read</i> and <i>write</i> are not supported
<i>pParams</i>	<i>in</i>	A pointer to the data structure used to configure the Quad Timer, or "NULL" if a subsequent <i>ioctl</i> call is made to QT_ENABLE to enable the Quad Timer
<i>FileDesc</i>	<i>out</i>	Quad Timer device descriptor returned by <i>open</i> call and used in <i>ioctl</i> function calls

Description: The *open* call opens the Quad Timer peripheral for operations and returns a file descriptor used by *ioctl* function calls. If *pParams* is a pointer to the *qt_sState* data structure, the open function will configure the timer and enable it to start counting. If *pParams* is NULL, the timer must be configured and enabled by a subsequent *ioctl* QT_ENABLE call.

Returns: The *open* call returns a type *types_tHandle*. When the *open* call is successful, this *types_tHandle* is a file descriptor for the Quad Timer driver. This file descriptor is used by the Quad Timer driver's *close* and *ioctl* functions. A "-1" is returned when the *open* call is not successful.

Code Example: See [Code Example 5-12](#).



5.4.4.2 *close*

Call(s):

```
int close (types_tHandle FileDesc);
```

Arguments:

Table 5-16. Quad Timer Driver Arguments - *close*

<i>FileDesc</i>	in	Quad Timer device descriptor returned by <i>open</i> call and used in <i>ioctl</i> function calls

Description: The *close* call halts operation of the timer peripheral specified in the *FileDesc*.

Returns: The *close* call returns a type *int*, which is always zero.

Code Example: See [Code Example 5-12](#).

5.4.4.3 *ioctl*

Call(s):

```
UWord16 ioctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams);
```

Table 5-17. Quad Timer Driver Arguments - *ioctl*

<i>FileDesc</i>	<i>in</i>	Quad Timer device descriptor returned by <i>open</i> call and used in <i>ioctl</i> function calls
<i>Cmd</i>	<i>in</i>	Commands found in <i>quadtimer.h</i> which are used to manage the Quad Timer driver
<i>pParams</i>	<i>in</i>	Pointer to <i>ioctl</i> command parameter; see Table 5-18

Table 5-18. Quad Timer *ioctl* / *qtioctl* Commands

Cmd	<i>pParams</i>	Return	Result
QT_ENABLE	qt_sState	None	Configures and Enables (starts) timer
QT_SET_COUNT_MODE	qt_eMode	None	Sets counter mode
QT_DISABLE	NULL	None	Disables (Stops) timer
QT_DISABLE_CALLBACK	qt_eCallback	None	Disables user callback
QT_ENABLE_CALLBACK	qt_eCallback	None	Enables user callback
QT_ENABLE_OUTPUT	NULL	None	Connects timer output signal to the external pin (OFLAG)
QT_DISABLE_OUTPUT	NULL	None	Disconnects timer output signal from external pin (OFLAG)
QT_FORCE_OUTPUT	bool	None	Forces output timer signal to the input value: True = force to Logic One False = force to Logic Zero
QT_ENABLE_CAPTURE_REG	NULL	None	Enables the detection of a transition on the package pin selected by the Secondary Count Source field of the Control register. This assumes that the Capture Mode bits of the Status and Control Registers have been enabled, as well as the Input Edge Flag Interrupt Enable bit.
QT_GET_STATUS	NULL	UWord16	Returns Status and Control register value
QT_WRITE_COMPARE_VALUE1	UWord16	None	Writes value to Compare register 1
QT_WRITE_COMPARE_VALUE2	UWord16	None	Writes value to Compare register 2
QT_WRITE_INITIAL_LOAD_VALUE	UWord16	None	Writes value to Initial Load Value register
QT_READ_CAPTURE_REG	NULL	UWord16	Returns Capture register value

Table 5-18. Quad Timer *ioctl* / *qtioctl* Commands (Continued)

Cmd	pParams	Return	Result
QT_READ_HOLD_REG	NULL	UWord16	Returns Hold register value
QT_READ_COUNTER_REG	NULL	UWord16	Returns Counter register value
QT_WRITE_COUNTER_REG	UWord16	NULL	Writes value to Counter register
QT_GET_INPUT_CLK_FREQ	UWord32 * (addr of output value)	None	Returns Quad Timer Input Clock Frequency in the input pointer
QT_SET_INPUT_CAPTURE_MODE	qt_eCapture Mode	None	Sets Input Capture Mode

Description: The *ioctl* call changes device modes. This interface references the peripheral dynamically through the *FileDesc* parameter.

Returns: The *ioctl* call returns a type *UWord16*. This *UWord16* is dependent on the command passed in as an input parameter. **Table 5-18** lists these returns based on the specific command.

Code Example: **Code Example 5-12** sets up Quad Timer A Channel Zero to turn the Green LED on and off.

Code Example 5-12. Quad Timer Driver Usage

```

/*****
#include "port.h"
#include "io.h"
#include "fcntl.h"
#include "bsp.h"

#include "quadtimer.h"
#include "led.h"

static void CallbackOnCompare0(qt_eCallbackType CallbackType, void* pParam);

const qt_sState quadParam0 = {
    /* Mode = */          qtCount,
    /* InputSource = */    qtPrescalerDiv128,
    /* InputPolarity = */  qtNormal,
    /* SecondaryInputSource = */ 0,
    /* CountFrequency = */ qtOnce,
    /* CountLength = */    qtUntilCompare,
    /* CountDirection = */ qtDown,
    /* OutputMode = */     qtAssertWhileActive,
    /* OutputPolarity = */ qtNormal,
    /* OutputDisabled = */ 1,
    /* Master = */         0,
    /* OutputOnMaster = */ 0,
    /* CoChannelInitialize = */ 0,
    /* AssertWhenForced = */ 0,

```



```

/* CaptureMode = */          qtDisabled,
/* CompareValue1 = */        0x8000,
/* CompareValue2 = */        0,
/* InitialLoadValue = */     0xFFFF,
/* CallbackOnCompare = */    {CallbackOnCompare0, 0},
/* CallbackOnOverflow = */   {0, 0},
/* CallbackOnInputEdge = */  {0, 0}
};

Word16      Counter;
bool        ServicedInterrupt = true;

/*****/
main()
{
    types_tHandle LedFD;
    types_tHandle TimerCompare0;

    /* open LED driver */
    LedFD = open(BSP_DEVICE_NAME_LED_0, 0);

    /* turn on "Green" LED */
    ioctl(LedFD, LED_OFF, LED_GREEN);

    /* initialize global counter variable */
    Counter = 0;

    /* open timer */
    TimerCompare0 = open(BSP_DEVICE_NAME_QUAD_TIMER_A_0, 0, NULL);

    /* disable timer */
    ioctl(TimerCompare0, QT_DISABLE, (void*)&quadParam0);

    /* initialize flag to true */
    ServicedInterrupt = true;

    /* infinite loop */
    while(1)
    {
        if(ServicedInterrupt == true)
        {
            switch(Counter % 2)
            {
                case 0:
                    ioctl(TimerCompare0, QT_ENABLE,
                        (void*)&quadParam0);

                    ioctl(LedFD, LED_ON, LED_GREEN);
                    break;

                case 1:
                    ioctl(TimerCompare0, QT_ENABLE,
                        (void*)&quadParam0);
                    ioctl(LedFD, LED_OFF, LED_GREEN);
            }
        }
    }
}

```



```

        break;
    }

    ServicedInterrupt = false;
}
}
/* close LED driver */
close(LedFD);

/* close timer */
close(TimerCompare0);
}

void CallbackOnCompare0(qt_eCallbackType CallbackType, void* pParam)
{
    /* increment count */
    Counter += 1;

    /* set flag to true */
    ServicedInterrupt = true;
}

/*****/

```



5.4.5 Quad Timer Low-Level I/O API Specifications

The following pages specify the low-level Application Programming Interface (API).

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

5.4.5.1 *qtOpen*

Call(s):

```
types_tHandle qtOpen(const char *pName, int OFlags, qt_sState * pParams);
```

Table 5-19. Quad Timer Driver Arguments - *open*

<i>pName</i>	<i>in</i>	The Quad timer device name; see Table 5-7
<i>OFlag</i>	<i>in</i>	General parameter to configure the Quad Timer driver; however, this parameter is not currently used
<i>pParams</i>	<i>in</i>	A pointer to the data structure used to configure the Quad Timer, or NULL if a subsequent <i>qtIoctl</i> call is made to QT_ENABLE to enable the Quad Timer

Description: The *open* call opens the Quad Timer peripheral for operations and returns a file descriptor used by *ioctl* function calls. If *pParams* is a pointer to the *qt_sState* data structure, the *open* function will configure the timer and enable it to start counting. If *pParams* is NULL, the timer must be configured and enabled by a subsequent *qtIoctl* QT_ENABLE call.

Returns: If *qtOpen* is successful, a file descriptor is returned. This file descriptor must be passed to other Quad Timer driver functions. If *qtOpen* fails, a “-1” value is returned.

Code Example: See [Code Example 5-13](#).



5.4.5.2 qtClose

Call(s):

```
int qtClose (types_tHandle FileDesc);
```

Table 5-20. Quad Timer Driver Arguments - close

FileDesc	in	Quad Timer device descriptor returned by qtOpen call and used in qtIoctl function calls

Description: The qtClose call halts operation of the timer peripheral specified in the FileDesc.

Returns: The qtClose call returns a type int, which is always zero.

Code Example: See Code Example 5-13.

5.4.5.3 *qtioctl*

```
UWord16 qtioctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams, void *
bspDeviceName);
```

Table 5-21. Quad Timer Driver Arguments - *ioctl*

<i>FileDesc</i>	<i>in</i>	Quad Timer device descriptor returned by <i>open</i> call and used in <i>ioctl</i> function calls
<i>Cmd</i>	<i>in</i>	Commands found in <i>quadtimer.h</i> which are used to manage the Quad Timer driver. See Section 5.4.4.3 for list of <i>qtioctl</i> Commands.
<i>pParams</i>	<i>in</i>	Pointer to <i>qtioctl</i> command parameter; see Table 5-18
<i>bspDeviceName</i>	<i>void*</i>	BSP device name, which was used in <i>open</i> statement

Description: The *qtioctl* call changes device modes. This interface references the peripheral statically through the *bspDeviceName* parameter.

Returns: The *qtioctl* call returns a type *UWord16*, which is dependent on the command passed in as an input parameter. [Table 5-18](#) lists these returns based on the specific command.

Code Example: [Code Example 5-13](#) shows an example of the use of the low level Quad Timer driver.

Code Example 5-13. Low-Level Quad Timer Driver Usage

```

/*****
#include "bsp.h"                /* Board support package driver */
#include "quadtimer.h"          /* Quad timer support */

void main(void)
{
    types_tHandle qtFD;

    /* Open the QT driver */
    qtFD = qtOpen(BSP_DEVICE_NAME_QUAD_TIMER_D_0, 0, NULL);

    /* Disable the QT */
    qtioctl(qtFD, QT_DISABLE, NULL, BSP_DEVICE_NAME_QUAD_TIMER_D_0);

    /* Close the QT driver */
    qtClose(qtFD);
}
*****/

```

5.4.6 Quad Timer Driver Tests

Currently, there are no stand-alone Quad Timer driver tests in the SDK.

5.4.7 Quad Timer Driver Demos

5.4.7.1 Quad Timer Flashing LEDs Demo

The Quad Timer flashing LEDs demonstrate use of three channels of the Quad Timer. The first timer goes off on an overflow, which starts the second timer. The second timer goes off on Compare 1, which starts the third timer. The third timer goes off on Compare 2, which starts the first timer. Each LED indicator reflects the status of the corresponding timer, and as a result, the LEDs on the running application are also lighted continuously.

5.4.7.1.1 Set-up for Quad Timer Flashing LEDs Demo

EVM Jumper Settings:

Use default settings as shown in the **DSP56852 Evaluation Module User's Manual**.

5.4.7.1.2 Procedure for Quad Timer Flashing LEDs Demo

- Using CodeWarrior, open:
...\\nos\\applications\\quadtimer\\quadtimer.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- At this point, three LEDs will turn on and off sequentially (red LED on, red LED off, green LED on, green LED off, yellow LED on, yellow LED off ...)

5.5 SCI Driver

5.5.1 Introduction

The DSP56852 processor has one SCI device, SCI0. This SCI device allows full-duplex, asynchronous serial communication between the DSP, a host computer, or other devices.

Note that the SCI is multiplexed with the GPIO that controls the codec on the DSP56852EVM. Care must be taken if the user desires to use both the SCI and codec. For example, the codec should be opened first, which will initialize it and then the SCI may be opened. If the codec must be reconfigured, the SCI will have to be reconfigured afterward.

5.5.2 Initialization

The SDK file *appconfig.h* provides the mechanism to statically configure the SCI driver. The *appconfig.h* file allows the programmer to set C preprocessor variables which configure the SCI

driver default operation. Typically, these C preprocessor variables conditionally compile and link information into the application for use by the SCI driver during initial operation.

To include the SCI Driver in the application, insert the following line into the *appconfig.h* file:

```
#define INCLUDE_SCI
```

Defining the preprocessor variable, INCLUDE_SCI, directs the SDK to include the SCI driver in the executable image. If the preprocessor variable INCLUDE_SCI is not defined, then the SCI driver will not be included in the executable.

The SCI_GET_SBR is a macro defined in *config.h*. The macro calculates the required baud rate based on PLL_MUL and BSP_OSCILLATOR_FREQ to a corresponding SCI Baud Rate Register value.

You may also define additional preprocessor variables in the *appconfig.h* file to statically configure and initialize the SCI driver. The list of these variables is provided in [Table 5-22](#).

Table 5-22. SCI Configuration Items for *appconfig.h*

SCI #define	Explanation and Default Value
SCI0_LOOP_MODE	Loopback mode of SCI peripheral Valid values are: SCI_CNTL_LOOPBACK_DISABLED (default) SCI_CNTL_LOOPBACK_ENABLED SCI_CNTL_LOOPBACK_SINGLE_WIRE_ENABLED
SCI0_DATA_MODE	Selects the SCI Data Mode Valid values are: SCI_8_DATA_BITS_NO_PARITY (default) SCI_7_DATA_BITS_WITH_ODD_PARITY SCI_7_DATA_BITS_WITH_EVEN_PARITY SCI_8_DATA_BITS_WITH_ODD_PARITY SCI_8_DATA_BITS_WITH_EVEN_PARITY SCI_9_DATA_BITS_NO_PARITY
SCI0_POLARITY	Selects the polarity of the SCI data Valid values are: SCI_DONT_INVERT_ANY_BITS (default) SCI_INVERT_ALL_BITS
SCI0_CONTROL_REG	Allows the user to define the SCI0 Control Register The default values is: (SCI0_DATA_MODE SCI0_POLARITY)
SCI0_BAUD_RATE	Selects the baud rate of the SCI peripheral 9600 (default)
SCI0_SEND_BUFFER_LENGTH	Fifo buffer size for write operation for SCI0 Default value is 8
SCI0_RECEIVE_BUFFER_LENGTH	Fifo buffer size for read operation for SCI0 Default value is 8

Table 5-22. SCI Configuration Items for *appconfig.h*

SCI #define	Explanation and Default Value
SCI0_TX_CALLBACK_FUNCTION	A user function for the SCI driver to call when the TX buffer has fallen below SCI0_WRITE_WATERMARK samples. The user function must be of type <i>tpSciDataCallback</i>: <pre>typedef void (*tpSciDataCallback)(void);</pre> The default is NULL
SCI0_RX_CALLBACK_FUNCTION	A user function for the SCI driver to call when it has received at least SCI0_READ_WATERMARK samples in its internal RX buffer. The user function must be of type <i>tpSciDataCallback</i> <pre>typedef void (*tpSciDataCallback)(void);</pre> The default is NULL
SCI0_ERROR_CALLBACK_FUNCTION	A user function for the SCI driver to call when it has detected an exception condition. The user function must be of type <i>tpSciErrorCallback</i> <pre>typedef void (*tpSciErrorCallback)(UWord16 Exception);</pre> The parameters to the callback function are: <i>Exception</i> - The actual exception that occurred 0x0800 - SCI_EXCEPTION_OVERRUN_ERROR 0x0400 - SCI_EXCEPTION_NOISE_ERROR 0x0200 - SCI_EXCEPTION_FRAME_ERROR 0x0100 - SCI_EXCEPTION_PARITY_ERROR 0x0008 - SCI_EXCEPTION_BUFFER_OVERFLOW 0x0004 - SCI_EXCEPTION_ADDRESS_MARK 0x0002 - SCI_EXCEPTION_BREAK_SYMBOL The default is NULL
SCI0_WRITE_WATERMARK	The minimum number of bytes that the SCI must maintain in the transmit FIFO buffer. If the number of samples falls below SCI0_WRITE_WATERMARK, the driver calls the user function specified by SCI0_TX_CALLBACK_FUNCTION. The value specified must be less than SCI0_SEND_BUFFER_LENGTH. Increasing the SCI0_WRITE_WATERMARK may increase the efficiency of the application, because more data can be transferred between the driver and the application in each callback. However, such an increase may adversely affect latency in the serial channel, because more data is buffered internally in the driver before passing it on to the application. The default is 0
SCI0_READ_WATERMARK	The number of bytes that the SCI must receive before calling the user function specified by SCI0_RX_CALLBACK_FUNCTION. The value specified must be less than SCI0_RECEIVE_BUFFER_LENGTH. Increasing the SCI0_READ_WATERMARK may increase the efficiency of the application, because more data can be transferred between the driver and the application in each callback. However, such an increase may adversely affect latency in the serial channel, because more data is buffered internally in the driver before passing it on to the application. The default is 0

5.5.3 API Definition

This section defines the Application Programming Interface (API) by identifying all public interface functions and data structures.

To automatically include code to create and initialize the SCI driver in an SDK Embedded Project, insert the following line in the *appconfig.h* file:

```
#define INCLUDE_SCI
```

The following information may be found in the public header file *sci.h*:

Public Interface Function(s):

```

/*****
*      types_tHandle open(const char *pName, int OFlags);
*      ssize_t write(types_tHandle FileDesc, const void * pBuffer, size_t NBytes);
*      ssize_t read(types_tHandle FileDesc, void * pBuffer, size_t NBytes);
*      int      close(types_tHandle FileDesc);
*      UWord16 ioctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams);
*****/
/*****
*      types_tHandle sciOpen(const char *pName, int OFlags);
*      ssize_t sciwrite(types_tHandle FileDesc, const void * pBuffer, size_t NBytes);
*      ssize_t sciRead(types_tHandle FileDesc, void * pBuffer, size_t NBytes);
*      int      sciClose(types_tHandle FileDesc);
*      UWord16 sciIoctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams,
*                      const char * pName);
*****/

```

Public Data Structure(s):

None

5.5.4 SCI Device Independent I/O API Specifications

The following pages specify the device-independent Application Programming Interface (API).

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

5.5.4.1 *open*

Call(s):

```
types_tHandle open(const char *pName, int OFlags);
```

Arguments:

Table 5-23. SCI Driver Arguments - *open*

<i>pName</i>	<i>in</i>	The SCI device name; see Table 5-24
<i>OFlag</i>	<i>in</i>	Open Mode Flags; specify O_RDWR Use O_NONBLOCK flags for Non-Blocking mode
<i>FileDesc</i>	<i>out</i>	SCI device descriptor returned by <i>open</i> call and used in <i>read/write</i> function calls

Table 5-24. SCI Devices for DSP56852 Chip

Chip	Can Use
DSP56852	BSP_DEVICE_NAME_SCI_0

Description: The *open* call enables the SCI peripheral for operations and returns a file descriptor used by the SCI driver's *read*, *write*, *close*, and *ioctl* function calls. The SCI device needs to be opened for *read/write* calls.

Returns: The *open* call returns a type *types_tHandle*. When the *open* call is successful, this *types_tHandle* is a file descriptor for the SCI device and is used by the SCI's *read*, *write*, and *ioctl* functions. A "-1" is returned when the *open* call is not successful.

Code Example: See [Code Example 5-14](#).

5.5.4.2 *write*

Call(s):

```
ssize_t write(types_tHandle FileDesc, const void * pBuffer, size_t NBytes);
```

Arguments:

Table 5-25. SCI Driver Arguments - *write*

<i>FileDesc</i>	<i>in</i>	SCI device descriptor returned by <i>open</i> call
<i>pBuffer</i>	<i>in</i>	Pointer to user buffer. The write function type casts <i>pBuffer</i> to be a (char *).
<i>NBytes</i>	<i>in</i>	Number of bytes to write

Description: The *write* call writes data from the user buffer to the SCI device.

Returns: The *write* call returns a type *ssize_t*, which is the number of transferred bytes.

Code Example: See [Code Example 5-14](#).

5.5.4.3 *read*

Call(s):

```
ssize_t read(types_tHandle FileDesc, void * pBuffer, size_t NBytes);
```

Arguments:

Table 5-26. SCI Driver Arguments - *read*

<i>FileDesc</i>	<i>in</i>	SCI device descriptor returned by <i>open</i> call
<i>pBuffer</i>	<i>in</i>	Pointer to user buffer. The read function type casts <i>pBuffer</i> to be a (char *).
<i>Size</i>	<i>in</i>	Number of bytes to read

Description: The *read* call reads data from the SCI device into the user buffer.

Returns: The *read* call returns a type *ssize_t*, which is the number of received bytes.

Code Example: None

5.5.4.4 *close*

Call(s):

```
int close (types_tHandle FileDesc);
```

Arguments:

Table 5-27. SCI Driver Arguments - *close*

<i>FileDesc</i>	<i>in</i>	SCI device descriptor returned by <i>open</i> call

Description: The *close* call disables the SCI peripheral.

Returns: The *close* call returns a type *int*. This *int* is always zero.

Code Example: See [Code Example 5-14](#).

5.5.4.5 *ioctl*

Call(s):

```
UWord16 ioctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams);
```

Arguments:

Table 5-28. SCI Driver Arguments - *ioctl*

<i>FileDesc</i>	<i>in</i>	SCI device descriptor returned by <i>open</i> call
<i>Cmd</i>	<i>in</i>	Commands found in <i>sci.h</i> which are used to manage the SCI driver
<i>pParams</i>	<i>in</i>	Pointer to <i>ioctl</i> command parameter

Table 5-29. SCI *ioctl* Commands

Cmd	<i>pParams</i>	Return	Result
SCI_DEVICE_RESET	NULL	None	Sets new SCI configuration and resets device; cancels all non-blocking operations
SCI_DEVICE_OFF	NULL	None	Disables device
SCI_DEVICE_ON	NULL	None	Enables device
SCI_CALLBACK_RX	void (*sciDataCallback) (void)	None	Installs Receive callback function; this function is called by the SCI Driver after receiving the number of bytes defined by SCI_SET_READ_LENGTH
SCI_CALLBACK_TX	void (*sciDataCallback) (void)	None	Installs Transmit Completed callback; this function is called when the SCI Driver has no more data to transmit
SCI_CALLBACK_EXCEPTION	void (*sciErrorCallback) (UWord16 Exception)	None	Installs Error callback; this function is called when an exception is detected by the SCI Driver
SCI_SET_READ_LENGTH	UWord16 length	None	Sets the number of words/bytes for Read callback; after this number of words is received, the Receive callback is called
SCI_GET_STATUS	NULL	Status	Gets device status
SCI_GET_EXCEPTION	NULL	Exception	Gets device exception status
SCI_CMD_SEND_BREAK	NULL	None	Sends a break symbol via SCI
SCI_CMD_WAIT	NULL	None	Puts the SCI device in wait state
SCI_CMD_WAKEUP	NULL	None	Wakes up the device from wait mode
SCI_CMD_READ_CLEAR	NULL	None	Clears the Read buffer in non-blocking mode
SCI_CMD_WRITE_CANCEL	NULL	None	Cancels data transmission after Write operation in non-blocking mode

Table 5-29. SCI *ioctl* Commands (Continued)

Cmd	pParams	Return	Result
SCI_CHANGE_BAUD_RATE	UWord16 BaudRate	None	Changes the Baud Rate
SCI_CHANGE_DATA_MODE	UWord16 DataMode	None	Changes the Data Mode SCI_7_DATA_BITS_WITH_ODD_PARITY SCI_7_DATA_BITS_WITH_EVEN_PARITY SCI_8_DATA_BITS_NO_PARITY SCI_8_DATA_BITS_WITH_ODD_PARITY SCI_8_DATA_BITS_WITH_EVEN_PARITY SCI_9_DATA_BITS_NO_PARITY
SCI_ENABLE_LOOPBACK	NULL	None	Enables Loopback mode
SCI_DISABLE_LOOPBACK	NULL	None	Disables Loopback mode
SCI_ENABLE_LOOPBACK_SINGLE_WIRE	NULL	None	Enables Single Wire Loopback mode
SCI_DISABLE_LOOPBACK_SINGLE_WIRE	NULL	None	Disables Single Wire Loopback mode

Description: The *ioctl* call changes SCI device modes or gives SCI driver status.

Returns: The *ioctl* call returns a type *UWord16*. This *UWord16* is dependent on the command passed in as an input parameter. [Table 5-29](#) lists these returns based on the specific command.

Code Example: [Code Example 5-14](#) sets up the SCI to write data out the serial port.

Code Example 5-14. SCI Driver Usage

```

/*****
#include "port.h"
#include "io.h"
#include "fcntl.h"
#include "bsp.h"

#include "sci.h"

void main(void)
{
    types_tHandle SciDevice;
    char Data;

    /* open sci driver */
    SciDevice = open(BSP_DEVICE_NAME_SCI_0, O_RDWR);

    Data = 0x0030;

    /* Send one byte via SCI0 */
    write(SciDevice, (void *)&Data, sizeof(char));

```



```
/* close sci driver */
close(SciDevice);
}
/*****/
```

5.5.5 SCI Low-Level I/O Specifications

The following pages specify the low-level Application Programming Interface (API).

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

5.5.5.1 *sciOpen*

Call(s):

```
types_tHandle sciOpen(const char *pName, int OFlags);
```

Arguments:

Table 5-30. SCI Driver Arguments - *sciOpen*

<i>pName</i>	<i>in</i>	The SCI device name; see Table 5-31
<i>OFlag</i>	<i>in</i>	Open Mode Flags; specify O_RDWR Use O_NONBLOCK flags for Non-Blocking mode
<i>FileDesc</i>	<i>out</i>	SCI device descriptor returned by <i>sciOpen</i> call and used in <i>sciRead/sciWrite</i> function calls

Table 5-31. SCI Devices for DSP56852 Chip

Chip	Can Use
DSP56852	BSP_DEVICE_NAME_SCI_0

Description: The *sciOpen* call enables the SCI peripheral for operations and returns a file descriptor used by the SCI driver's *sciRead*, *sciWrite*, *sciClose*, and *sciIoctl* function calls. The SCI device must be opened for *read/write* calls.

Returns: The *sciOpen* call returns a type *types_tHandle*. When the *sciOpen* call is successful, this *types_tHandle* is a file descriptor for the SCI device and is used by the SCI *sciRead*, *sciWrite*, and *sciIoctl* functions. A "-1" is returned when the *sciOpen* call is not successful.

Code Example: See [Code Example 5-15](#).

5.5.5.2 *sciWrite*

Call(s):

```
ssize_t sciWrite(types_tHandle FileDesc, const void * pBuffer, size_t NBytes);
```

Arguments:

Table 5-32. SCI Driver Arguments - *sciWrite*

<i>FileDesc</i>	<i>in</i>	SCI device descriptor returned by <i>sciOpen</i> call
<i>pBuffer</i>	<i>in</i>	Pointer to user buffer. The write function type casts <i>pBuffer</i> to be a (char *).
<i>NBytes</i>	<i>in</i>	Number of bytes to write

Description: The *sciWrite* call writes data from the user buffer to the SCI device.

Returns: The *sciWrite* call returns a type *ssize_t*, which is the number of transferred bytes.

Code Example: See [Code Example 5-15](#).



5.5.5.3 *sciRead*

Call(s):

```
ssize_t sciRead(types_tHandle FileDesc, void * pBuffer, size_t NBytes);
```

Arguments:

Table 5-33. SCI Driver Arguments - *sciRead*

<i>FileDesc</i>	<i>in</i>	SCI device descriptor returned by <i>sciOpen</i> call
<i>pBuffer</i>	<i>in</i>	Pointer to user buffer. The read function type casts <i>pBuffer</i> to be a (char *).
<i>Size</i>	<i>in</i>	Number of bytes to read

Description: The *sciRead* call reads data from SCI device into the user buffer.

Returns: The *sciRead* call returns a type *ssize_t*, which is the number of received bytes.

Code Example: None

5.5.5.4 *sciClose*

Call(s):

```
int sciClose (types_tHandle FileDesc);
```

Arguments:

Table 5-34. SCI Driver Arguments - *sciClose*

<i>FileDesc</i>	<i>in</i>	SCI device descriptor returned by <i>sciOpen</i> call

Description: The *sciClose* call disables the SCI peripheral.

Returns: The *sciClose* call returns a type *int*. This *int* is always zero.

Code Example: See [Code Example 5-15](#).

5.5.5.5 *scioctl*

Call(s):

```
UWord16 sciIoctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams, const char *pName);
```

Arguments:

Table 5-35. SCI Driver Arguments - *scioctl*

<i>FileDesc</i>	in	SCI device descriptor returned by <i>sciOpen</i> call
<i>Cmd</i>	in	Commands found in <i>sci.h</i> which are used to manage the SCI driver
<i>pParams</i>	in	Pointer to <i>scioctl</i> command parameter
<i>pName</i>	in	the SCI device name from <i>bsp.h</i> ; typically, BSP_DEVICE_NAME_SCI_0

Table 5-36. SCI *scioctl* Commands

Cmd	<i>pParams</i>	Return	Result
SCI_DEVICE_RESET	NULL	None	Sets new SCI configuration and resets device; cancels all non-blocking operations
SCI_DEVICE_OFF	NULL	None	Disables device
SCI_DEVICE_ON	NULL	None	Enables device
SCI_CALLBACK_RX	void (*sciDataCallback) (void)	None	Installs Receive callback function; this function is called by the SCI Driver after receiving the number of bytes defined by SCI_SET_READ_LENGTH
SCI_CALLBACK_TX	void (*sciDataCallback) (void)	None	Installs Transmit completed callback; this function is called when the SCI Driver has no more data to transmit
SCI_CALLBACK_EXCEPTION	void (*sciErrorCallback) (UWord16 Exception)	None	Installs Error callback; this function is called when an exception is detected by the SCI Driver
SCI_SET_READ_LENGTH	UWord16 length	None	Sets the number of words/bytes for Read callback; after this number of words is received, the Receive callback is called
SCI_GET_STATUS	NULL	Status	Gets device status
SCI_GET_EXCEPTION	NULL	Exception	Gets device exception status
SCI_CMD_SEND_BREAK	NULL	None	Sends a break symbol via SCI
SCI_CMD_WAIT	NULL	None	Puts an SCI device in wait state
SCI_CMD_WAKEUP	NULL	None	Wakes up the device from wait mode
SCI_CMD_READ_CLEAR	NULL	None	Clears Read buffer in non-blocking mode

Table 5-36. SCI *scioctl* Commands (Continued)

Cmd	pParams	Return	Result
SCI_CMD_WRITE_CANCEL	NULL	None	Cancels data transmission after Write operation in non-blocking mode
SCI_CHANGE_BAUD_RATE	UWord16 BaudRate	None	Changes the Baud Rate
SCI_CHANGE_DATA_MODE	UWord16 DataMode	None	Changes the Data Mode SCI_7_DATA_BITS_WITH_ODD_PARITY SCI_7_DATA_BITS_WITH_EVEN_PARITY SCI_8_DATA_BITS_NO_PARITY SCI_8_DATA_BITS_WITH_ODD_PARITY SCI_8_DATA_BITS_WITH_EVEN_PARITY SCI_9_DATA_BITS_NO_PARITY
SCI_ENABLE_LOOPBACK	NULL	None	Enables Loopback mode
SCI_DISABLE_LOOPBACK	NULL	None	Disables Loopback mode
SCI_ENABLE_LOOPBACK_SINGLE_WIRE	NULL	None	Enables Single Wire Loopback mode
SCI_DISABLE_LOOPBACK_SINGLE_WIRE	NULL	None	Disables Single Wire Loopback mode

Description: The *scioctl* call changes SCI device modes or gives SCI driver status.

Returns: The *scioctl* call returns a type *UWord16*. This *UWord16* is dependent on the command passed in as an input parameter. [Table 5-36](#) lists these returns based on the specific command.

Code Example: [Code Example 5-15](#) sets up the SCI to write data out the serial port.

Code Example 5-15. SCI Driver Usage

```

/*****
#include "port.h"
#include "fcntl.h"
#include "bsp.h"

#include "sci.h"

void main(void)
{
    types_tHandle SciDevice;
    char          Data;

    /* open sci driver */
    SciDevice = sciOpen(BSP_DEVICE_NAME_SCI_0, 0_RDWR);

    Data = 0x0030;

    /* Send one byte via SCI0 */
    sciWrite(SciDevice, (void *)&Data, sizeof(char));

```



```
/* close sci driver */
sciClose(SciDevice);
}
/*****/
```

5.5.6 SCI Driver Tests

The SCI Driver Tests verify correct operation of the SCI driver provided by the SDK.

5.5.6.1 SCI Loopback Test

The SCI loopback test, which runs on the DSP56852, configures the SCI port to loop data back internally and uses the SCI driver to write and read samples from the SCI port.

5.5.6.1.1 Set-up for SCI Loopback Test

EVM Jumper Settings:

Use default settings as shown in the **DSP56852 Evaluation Module User's Manual**.

5.5.6.1.2 Procedure for SCI Loopback Test

- Using CodeWarrior, open:
...\\nos\\bsp\\test\\sci\\testsci.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- The test will display the results of the test on CodeWarrior's console window

5.5.7 SCI Driver Demos

5.5.7.1 SCI Serial Port Demo

The SCI Serial Port Demo interacts with a desktop PC via a serial connection. The SCI driver for this application is configured by the PC application to run at a baud rate of 28800.

5.5.7.1.1 Set-up for SCI Serial Port Demo

EVM Jumper Settings:

Use default settings as shown in the **DSP56852 Evaluation Module User's Manual**.

5.5.7.1.2 Procedure for SCI Serial Port Demo

- Using CodeWarrior, open:
...\\nos\\applications\\sci\\sci.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the demo
- Once this debugger is running, the user can execute the *serial.exe* program on the PC to launch a serial GUI interface. This file can be found at:
..\\x86\\win32\\applications\\serial\\
 - With this GUI, the user can display any memory location on the DSP56852EVM board by selecting the data space and address range, then selecting *READ*
 - When *READ* is selected, a command is sent to the DSP56852 processor via the serial interface
 - After the application, running on the DSP56852, processes this command, a response message with the requested data is sent back to the PC
 - As soon as the GUI application running on the PC receives the response, the data for the requested memory location will be displayed

5.5.7.2 SCI Echo Demo

The SCI Echo Demo interacts with a desktop PC via a serial connection. The SCI driver for this application is configured by the PC application to run at a baud rate of 9600 bps.

5.5.7.2.1 Set-up for SCI Echo Demo

EVM Jumper Settings:

Use default settings as shown in the **DSP56852 Evaluation Module User's Manual**.

- Connect the serial port of the PC (COM 1) to the serial port on the EVM board
- Execute Hyperterminal on the PC
- Configure Hyperterminal for:

Baud Rate:	9600
Data Bits:	8
Parity:	None
Stop Bit:	1

5.5.7.2.2 Procedure for SCI Echo Demo

- Using CodeWarrior, open:
 ...\\nos\\applications\\sci_echo\\sci_echo.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the demo
- Once this debugger is running, the user can type characters using Hyperterminal and the SCI Echo demo will echo the characters to the Hyperterminal's screen
- To exit this demo, type 'X'

5.6 SPI Driver

5.6.1 Introduction

The DSP56852 processor has an SPI device which allows full duplex, synchronous serial communication between the DSP and peripheral devices. This section describes the Serial Peripheral Interface driver software providing the lowest-level software layer interfacing hardware to software.

NOTE: On the DSP56852, the SPI is multiplexed with the SSI; thus, both cannot be used at the same time.

5.6.2 Initialization

The SPI device driver currently has no default configurations that can be overwritten through a project's *appconfig.h* file.

5.6.3 API Definition

This section defines the Application Programming Interface (API) by identifying all public interface functions and data structures.

Insert the following line in the *appconfig.h* file to automatically include the code to create and initialize the SPI driver in an SDK Embedded Project:

```
#define INCLUDE_SPI
```

Defining the preprocessor statement, `INCLUDE_SPI`, forces the SDK to include the SPI driver in the project. If the `INCLUDE_SPI` preprocessor statement is not defined, then the SPI driver will not be included in the executable.

You may also define additional preprocessor statements in the *appconfig.h* file to statically configure and initialize the SPI driver. A list of these variables is provided in [Table 5-37](#).

Table 5-37. SPI Configuration Items for *appconfig.h*

SPI #define	Explanation and Default Value
<code>SPI_NON_BLOCKED</code>	Blocked/non-blocked mode is selected statically Default = 0
<code>SPI_IS_MASTER</code>	A bool that sets the SPI device to a Master when set to 1 and to a Slave when set to 0.
<code>SPI_SLAVE_SELECT_CALLBACK</code>	A pointer to a function that enables the SS; as default, SPI0 will use the MPIO C pin 3 to assert the SS
<code>SPI_SLAVE_DESELECT_CALLBACK</code>	A pointer to a function that disables the SS; as default, SPI0 will use the MPIO C pin 3 to deassert the SS
<code>SPI_SEND_BUFFER_LENGTH</code>	FIFO transmission buffer length for non-blocked mode Default = 32
<code>SPI_RECEIVE_BUFFER_LENGTH</code>	FIFO reception buffer length for non -locked mode Default = 32
<code>SPI_BAUD_RATE_DIVISOR</code>	Divides SPI clock by value Default = 32
<code>SPI_TRANSMISSION_SIZE</code>	Sets the size of transmission from 2 to 16 bits. This element must be the desired length - 1. Example: If the data size is 8 bits, this element must be set to 7; a value of 0 is invalid Default = 0x07 (8 bit)

The following information may be found in the public header file *spi.h*:

Public Interface Function(s):

```

/*****
*      types_tHandle open(const char *pName, int OFlags);
*      ssize_t write(types_tHandle FileDesc, const void * pBuffer, size_t NBytes);
*      ssize_t read(types_tHandle FileDesc, void * pBuffer, size_t NBytes);
*      int      close(types_tHandle FileDesc);
*      UWord16 ioctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams);
*****/
/*****
*      types_tHandle spiOpen(const char *pName, int OFlags);
*      ssize_t spiBlockedWrite(types_tHandle FileDesc, const void * pBuffer,
*                               size_t NBytes);
*      ssize_t spiBlockedRead(types_tHandle FileDesc, void * pBuffer, size_t NBytes);
*      int      spiBlockedClose(types_tHandle FileDesc);
*      ssize_t spiNonBlockedWrite(types_tHandle FileDesc, const void * pBuffer,
*                                  size_t NBytes);
*      ssize_t spiNonBlockedRead(types_tHandle FileDesc, void * pBuffer,
*                                  size_t NBytes);
*      int      spiNonBlockedClose(types_tHandle FileDesc);
*      UWord16 spiIoctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams);
*****/

```

5.6.4 SPI Device-Independent I/O API Specifications

The following pages specify the device-independent Application Programming Interface (API).

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

5.6.4.1 *open*

Call(s):

```
types_tHandle open(const char *pName, int OFlags);
```

Arguments:

Table 5-38. SPI Driver Arguments - *open*

<i>pName</i>	<i>in</i>	The SPI device name; use BSP_DEVICE_NAME_SPI_0
<i>OFlag</i>	<i>in</i>	Open Mode Flags which are ignored
<i>FileDesc</i>	<i>out</i>	SPI device descriptor returned by the <i>open</i> call and used in <i>read/write/ioctl</i> function calls

Description: The *open* call opens the SPI peripheral for operations and returns a file descriptor used by the SPI driver's *read*, *write*, *close*, and *ioctl* function calls. The SPI device needs to be opened for *read/write* calls.

Returns: The *open* call returns a type *types_tHandle*. When the *open* call is successful, this *types_tHandle* is a file descriptor for the SPI device used by the SPI driver's *read*, *write*, *close*, and *ioctl* functions. A "-1" is returned when the *open* call is not successful.

Code Example: See [Code Example 5-17](#).

5.6.4.2 **write**

Call(s):

```
ssize_t write(types_tHandle FileDesc, const void * pBuffer, size_t NBytes);
```

Arguments:

Table 5-39. SPI Driver Arguments - *write*

<i>FileDesc</i>	<i>in</i>	SPI device descriptor returned by the <i>open</i> call and used in <i>read/write/ioctl</i> function calls
<i>pBuffer</i>	<i>in</i>	Pointer to user buffer
<i>NBytes</i>	<i>in</i>	Size of data to read/write in bytes

Description: The *write* call writes data from the user buffer to the SPI device.

Returns: The *write* call returns a type *ssize_t*, which is the number of transferred bytes. These bytes are defined as the transmission size of the input bytes.

Code Example: See [Code Example 5-17](#).

5.6.4.3 *read*

Call(s):

```
ssize_t read(types_tHandle FileDesc, void * pBuffer, size_t NBytes);
```

Arguments:

Table 5-40. SPI Driver Arguments - *read*

<i>FileDesc</i>	<i>in</i>	SPI device descriptor returned by <i>open</i> call and used in <i>read/write/ioctl</i> function calls
<i>pBuffer</i>	<i>in</i>	Pointer to user buffer
<i>NBytes</i>	<i>in</i>	Size of data to read/write in bytes

Description: The *read* call reads data from SPI device into the user buffer.

Returns: The *read* call returns a type *ssize_t*. When the SPI device is in Master mode, the function will return only one word (2 bytes). This word is the last word read from the SPI device, due to the last write to the SPI device. When the SPI device is in Slave mode, the function will return *NBytes* bytes.

NOTE: Since the SPI peripheral is a full-duplex device, a write must be performed before each read while using the SPI device in Master mode.

Code Example: None

5.6.4.4 *close*

Call(s):

```
int close (types_tHandle FileDesc);
```

Arguments:

Table 5-41. SPI Driver Arguments - *close*

<i>FileDesc</i>	<i>in</i>	SPI device descriptor returned by <i>open</i> call and used in <i>read/writel/ioctl</i> function calls

Description: The *close* call closes the SPI peripheral and releases the file descriptor handle.

Returns: The *close* call returns a type *int*, which is always zero.

Code Example: [Code Example 5-17](#) sets up the SPI as a master device and writes data out of the SPI peripheral.

Code Example 5-16. SPI Driver Settings in *appconfig.h*

```
#define INCLUDE_SPI

#define SPI_NON_BLOCKED          0 /* blocked mode */
#define SPI_IS_MASTER            1 /* SPI is master */
#define SPI_TRANSMISSION_SIZE    0xF /* 16 bit */
#define SPI_BAUD_RATE_DIVISOR    SPI_BAUDRATE_DIVIDER_32
#define SPI_SLAVE_SELECT_CALLBACK /* Use default PortB GPIO Pin 3 to toggle SS */
#define SPI_SLAVE_DESELECT_CALLBACK /* Use default PortB GPIO Pin 3 to toggle SS */
```

Code Example 5-17. SPI Driver Usage

```
/******
#include "port.h"
#include "io.h"
#include "fcntl.h"
#include "bsp.h"
#include "spi.h"

types_tHandle SerialMaster;
UWord16      Data = 0x1234;

/* open the spi driver */
SerialMaster = open(BSP_DEVICE_NAME_SPI_0, O_BLOCK);

/* write "Data" out of the spi device */
write(SerialMaster, &Data, sizeof(UWord16));

/* close spi driver */
close(SerialMaster);

******/
```

5.6.4.5 *ioctl*

Call(s):

```
UWord16 ioctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams);
```

Arguments:

Table 5-42. SPI Driver Arguments - *ioctl*

<i>FileDesc</i>	<i>in</i>	SPI device descriptor returned by <i>open</i> call
<i>Cmd</i>	<i>in</i>	Commands found in <i>spi.h</i> which are used to modify the SPI status and control registers
<i>pParams</i>	<i>in</i>	Used to pass in data transmission size with SPI_TRANSMISSION_DATA_SIZE <i>ioctl</i> function call

Table 5-43. SPI *ioctl* Commands

Cmd	<i>pParams</i>	Return	Result
SPI_DATA_SHIFT_MSB_FIRST	None	None	MSB transmitted first
SPI_DATA_SHIFT_LSB_FIRST	None	None	LSB transmitted first
SPI_ERROR_INTERRUPT_ENABLE	None	None	MODF and OVRF can generate DSP interrupt requests
SPI_ERROR_INTERRUPT_DISABLE	None	None	MODF and OVRF cannot generate DSP interrupt requests
SPI_MODE_FAULT_ENABLE	None	None	Allows MODF to be set
SPI_MODE_FAULT_DISABLE	None	None	The level of the SS pin does not affect the operation of an enabled SPI configured as a Master
SPI_CLEAR_MODE_FAULT	None	None	Clears Mode Fault Bit after it has been set
SPI_BAUDRATE_DIVIDER_2	None	None	Divides clock by 2
SPI_BAUDRATE_DIVIDER_8	None	None	Divides clock by 8
SPI_BAUDRATE_DIVIDER_16	None	None	Divides clock by 16
SPI_BAUDRATE_DIVIDER_32	None	None	Divides clock by 32
SPI_RX_INTERRUPT_ENABLE	None	None	SPRF DSP interrupt requests enabled
SPI_RX_INTERRUPT_DISABLE	None	None	SPRF DSP interrupt requests disabled
SPI_MODE_MASTER	None	None	Puts SPI in Master mode
SPI_MODE_SLAVE	None	None	Puts SPI in Slave mode
SPI_CLK_POL_RISING_EDGE	None	None	Rising edge of SCLK starts transmission
SPI_CLK_POL_FALLING_EDGE	None	None	Falling edge of SCLK starts transmission

Table 5-43. SPI *ioctl* Commands (Continued)

Cmd	<i>pParams</i>	Return	Result
SPI_CLOCK_PHASE_SET	None	None	SS pin of the slave SPI module does not have to be set to a Logic One between full length data transmissions
SPI_CLOCK_PHASE_NOTSET	None	None	SS pin of the slave device module must be set to a Logic One between full length data transmissions
SPI_ENABLE	None	None	SPI module enabled
SPI_DISABLE	None	None	SPI module disabled
SPI_TX_INTERRUPT_ENABLE	None	None	SPTE interrupt requests enabled
SPI_TX_INTERRUPT_DISABLED	None	None	SPTE interrupt requests disabled
SPI_TRANSMISSION_DATA_SIZE	2 - 16	None	The data transmission size in bits. (<i>pParams</i> must be one less bit than the desired length. For instance, if the data size is 8 bits, then <i>pParams</i> should be set to 7. A value of 0 for <i>pParams</i> is invalid.)

Description: The *ioctl* call changes SPI device modes.

Returns: The *ioctl* call returns a type *UWord16*, which is always zero.

Code Example: [Code Example 5-18](#) sets up the SPI as a master device and makes the LSB transmit first.

Code Example 5-18. SPI Driver Usage

```

/*****
#include "port.h"
#include "io.h"
#include "fcntl.h"
#include "bsp.h"
#include "spi.h"

types_tHandle SerialMaster;

/* open spi driver */
SerialMaster = open(BSP_DEVICE_NAME_SPI_0, O_BLOCK);

/* this will make the LSB tx first */
ioctl(SerialMaster, SPI_DATA_SHIFT_LSB_FIRST, NULL);

/* close spi driver */
close(SerialMaster);
*****/
    
```



5.6.5 SPI Low-Level I/O API Specification

The following pages specify the low-level Application Programming Interface (API).

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

5.6.5.1 *spiOpen*

Call(s):

```
types_tHandle spiOpen(const char *pName, int OFlags);
```

Arguments:

Table 5-44. SPI Driver Arguments - *spiOpen*

<i>pName</i>	<i>in</i>	The SPI device name; use BSP_DEVICE_NAME_SPI_0
<i>OFlag</i>	<i>in</i>	Open Mode Flags which are ignored
<i>FileDesc</i>	<i>out</i>	SPI device descriptor returned by the <i>spiOpen</i> call and used in <i>spiRead/spiWrite/spiIoctl</i> function calls

Description: Opens a particular SPI device. Argument *pName* is the name of the particular device.

Returns: If *spiOpen* is successful, a file descriptor is returned. This file descriptor must be passed to other SPI driver functions. If *spiOpen* fails, a “-1” value is returned.

Code Example: See [Code Example 5-17](#).

5.6.5.2 *spiBlockedWrite, spiNonBlockedWrite*

Call(s):

```
ssize_t spiBlockedWrite(types_tHandle FileDesc, const void * pBuffer, size_t
NBytes);
```

Arguments:

Table 5-45. SPI Driver Arguments - *spiWrite*

<i>FileDesc</i>	<i>in</i>	SPI device descriptor returned by the <i>spiOpen</i> call
<i>pBuffer</i>	<i>in</i>	Pointer to user buffer
<i>NBytes</i>	<i>in</i>	Size of data to read/write in bytes

Description: The *spiWrite* call writes data from the user buffer to the SPI device.

Returns: The *spiWrite* call returns a type *ssize_t*, which is the number of transferred bytes. These bytes are defined as the transmission size of the input data.

Code Example: See [Code Example 5-17](#).

5.6.5.3 *spiBlockedRead, spiNonBlockedRead*

Call(s):

```
ssize_t spiNonBlockedRead(types_tHandle FileDesc, void * pBuffer, size_t
NBytes);
```

Arguments:
Table 5-46. SPI Driver Arguments - *spiRead*

<i>FileDesc</i>	<i>in</i>	SPI device descriptor returned by <i>spiOpen</i> call
<i>pBuffer</i>	<i>in</i>	Pointer to user buffer
<i>NBytes</i>	<i>in</i>	Size of data to read/write in bytes

Description: The *spiRead* call reads data from the SPI device into the user buffer.

Returns: The *spiRead* call returns a type *ssize_t*. When the SPI device is in Master mode, the function will return only one word (2 bytes). This word is the last word read from the SPI device, due to the last write to the SPI device. When the SPI device is in Slave mode, the function will return *NBytes* bytes.

NOTE: Since the SPI peripheral is a full-duplex device, a write must be performed before each read while using the SPI device in Master mode.

Code Example: None

5.6.5.4 *spiBlockedClose, spiNonBlockedClose*

Call(s):

```
int spiBlockedClose (types_tHandle FileDesc);
```

Arguments:

Table 5-47. SPI Driver Arguments - *spiClose*

<i>FileDesc</i>	<i>in</i>	SPI device descriptor returned by <i>spiOpen</i> call

Description: The *spiClose* call closes the SPI peripheral and releases the file descriptor handle.

Returns: The *spiClose* call returns a type *int*, which is always zero.

Code Example: [Code Example 5-17](#) sets up the SPI as a master device and writes data out of the SPI peripheral.

Code Example 5-19. SPI Driver Usage with *ioctl* call

```

/*****
#include "port.h"
#include "io.h"
#include "fcntl.h"
#include "bsp.h"
#include "spi.h"

types_tHandle SerialMaster;
UWord16      Data = 0x1234;

/* open the spi driver */
SerialMaster = spiOpen(BSP_DEVICE_NAME_SPI_0, 0_BLOCK);

/* write "Data" out of the spi device */
spiWrite(SerialMaster, &Data, sizeof(UWord16));

/* close spi driver */
spiClose(SerialMaster);
*****/

```

5.6.5.5 *spioctl*

Call(s):

```
UWord16 spiIoctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams, void *
bspDeviceName);
```

Arguments:
Table 5-48. SPI Driver Arguments - *spioctl*

<i>FileDesc</i>	<i>in</i>	SPI device descriptor returned by <i>open</i> call
<i>Cmd</i>	<i>in</i>	Commands found in <i>spi.h</i> which are used to modify the SPI status and control registers
<i>pParams</i>	<i>in</i>	Used to pass in data transmission size with the SPI_TRANSMISSION_DATA_SIZE <i>ioctl</i> function call
<i>bspDeviceName</i>	<i>in</i>	BSP device name, which was used in <i>open</i> statement

Table 5-49. SPI *spioctl* Commands

Cmd	<i>pParams</i>	Return	Result
SPI_DATA_SHIFT_MSB_FIRST	None	None	MSB transmitted first
SPI_DATA_SHIFT_LSB_FIRST	None	None	LSB transmitted first
SPI_ERROR_INTERRUPT_ENABLE	None	None	MODF and OVRF can generate DSP interrupt requests
SPI_ERROR_INTERRUPT_DISABLE	None	None	MODF and OVRF cannot generate DSP interrupt requests
SPI_MODE_FAULT_ENABLE	None	None	Allows MODF to be set
SPI_MODE_FAULT_DISABLE	None	None	The level of the SS pin does not affect the operation of an enabled SPI configured as a Master
SPI_CLEAR_MODE_FAULT	None	None	Clears Mode Fault Bit after it has been set
SPI_BAUDRATE_DIVIDER_2	None	None	Divides clock by 2
SPI_BAUDRATE_DIVIDER_8	None	None	Divides clock by 8
SPI_BAUDRATE_DIVIDER_16	None	None	Divides clock by 16
SPI_BAUDRATE_DIVIDER_32	None	None	Divides clock by 32
SPI_RX_INTERRUPT_ENABLE	None	None	SPRF DSP interrupt requests enabled
SPI_RX_INTERRUPT_DISABLE	None	None	SPRF DSP interrupt requests disabled
SPI_MODE_MASTER	None	None	Puts SPI in Master mode
SPI_MODE_SLAVE	None	None	Puts SPI in Slave mode
SPI_CLK_POL_RISING_EDGE	None	None	Rising edge of SCLK starts transmission

Table 5-49. SPI *spioctl* Commands (Continued)

Cmd	<i>pParams</i>	Return	Result
SPI_CLK_POL_FALLING_EDGE	None	None	Falling edge of SCLK starts transmission
SPI_CLOCK_PHASE_SET	None	None	SS pin of the slave SPI module does not have to be set to a Logic One between full length data transmissions
SPI_CLOCK_PHASE_NOTSET	None	None	SS pin of the slave device module must be set to a Logic One between full length data transmissions
SPI_ENABLE	None	None	SPI module enabled
SPI_DISABLE	None	None	SPI module disabled
SPI_TX_INTERRUPT_ENABLE	None	None	SPTE interrupt requests enabled
SPI_TX_INTERRUPT_DISABLED	None	None	SPTE interrupt requests disabled
SPI_TRANSMISSION_DATA_SIZE	2 - 16	None	The data transmission size in bits. (<i>pParams</i> must be one less bit than the desired length. For instance, if the data size is 8 bits, then <i>pParams</i> should be set to 7. A value of 0 for <i>pParams</i> is invalid.)

Description: The *spioctl* call changes SPI device modes.

Returns: The *spioctl* call returns a type *UWord16*, which is always zero.

Code Example: [Code Example 5-20](#) sets up the SPI as a master device and makes the LSB transmit first.

Code Example 5-20. SPI Driver Usage

```

/*****
#include "port.h"
#include "fcntl.h"
#include "bsp.h"
#include "spi.h"

types_tHandle SerialMaster;

/* open spi driver */
SerialMaster = spiOpen(BSP_DEVICE_NAME_SPI_0, 0_BLOCK);

/* this will make the LSB tx first */
spioctl(SerialMaster, SPI_DATA_SHIFT_LSB_FIRST, NULL);

/* close spi driver */
spiClose(SerialMaster);
*****/

```

5.6.6 SPI Driver Applications

The Serial Data Flash application uses the SPI driver. Please see [Section 6.8.6](#) for more information.

5.6.7 SPI Driver Tests

The SPI Driver Tests verify correct operation of the SPI driver provided by the SDK.

5.6.7.1 SPI Test

The SPI test configures the SPI port to work with the flash chip on the EVM board in non-blocked mode. It reads the flash status register and some flash data.

5.6.7.1.1 Set-up for SPI Test

EVM Jumper Settings:

Use default settings as shown in the **DSP56852 Evaluation Module User's Manual**.

5.6.7.1.2 Procedure for SPI Test

- Using CodeWarrior, open:
 `...\nos\bsp\test\spitest\spitest.mcp`
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- The test will display the results of the test on CodeWarrior's console window

5.7 SSI Driver

5.7.1 Introduction

The DSP56852 processor has an SSI device which allows full duplex, synchronous serial communication between the DSP and peripheral devices. This section describes the SSI driver software providing the lowest-level software layer interfacing hardware to software.

NOTE: On the DSP56852, the SSI is multiplexed with the SPI; thus, both cannot be used at the same time.

5.7.2 Initialization

The SDK file *appconfig.h* provides the mechanism to statically configure the SSI driver. The *appconfig.h* file allows the programmer to set C preprocessor variables which configure the SSI driver default operation. Typically, these C preprocessor variables conditionally compile and link information into the application for use by the SSI driver during initial operation.

To include the SSI Driver in the application, insert the following line into the *appconfig.h* file:

```
#define INCLUDE_SSI
```

Defining the preprocessor variable, `INCLUDE_SSI`, directs the SDK to include the SSI driver in the executable image. If the preprocessor variable `INCLUDE_SSI` is not defined, then the SSI driver will not be included in the executable.

You may also define additional preprocessor variables in the *appconfig.h* file to statically configure and initialize the SSI driver. The list of these variables is provided in [Table 5-50](#).

The SSI's default configuration is defined in *config.h* and is shown in [Table 5-50](#). Most of the SSI #defines in [Table 5-50](#) corresponds directly to a bit or group of bits in the SSI control registers. Please refer to the **DSP56852 16-Bit Digital Signal Processor User's Manual** for a detailed description of these bits.

Table 5-50. SSI Configuration Items for *appconfig.h*

SSI #define	Explanation and Default Value
SSI_FIFO_SIZE	Default size (number of 16-bit samples) for both the RX FIFO and TX FIFO buffers used internally by the driver. For more information, refer to <code>SSI_RX_FIFO_SIZE</code> and <code>SSI_TX_FIFO_SIZE</code> Default =32
SSI_RX_FIFO_SIZE	Size (number of 16-bit samples) of the RX FIFO buffer used internally by the driver for the RX channel. The length of the RX FIFO buffer determines the maximum number of SSI input samples that can be buffered within the driver. Default =SSI_FIFO_SIZE
SSI_TX_FIFO_SIZE	Size (number of 16-bit samples) of the TX FIFO buffer used internally by the driver for the TX channel. The length of the TX FIFO buffer determines the maximum number of SSI output samples that can be buffered within the driver. Default = SSI_FIFO_SIZE
SSI_RX_CALLBACK_LEVEL	The number of 16-bit samples that the SSI must receive before calling the user function specified by <code>SSI_RX_CALLBACK</code>. The value specified must be less than <code>SSI_RX_FIFO_SIZE</code>. Increasing the <code>SSI_RX_CALLBACK_LEVEL</code> may increase the efficiency of the application, because more data can be transferred between the driver and the application in each callback. However, such an increase may adversely affect latency in the audio channel because more data is buffered internally in the driver before passing it on to the application. Default = 8

Table 5-50. SSI Configuration Items for *appconfig.h*

SSI #define	Explanation and Default Value
SSI_RX_CALLBACK	A user function for the SSI driver to call when it has received at least SSI_RX_CALLBACK_LEVEL samples in its internal RX buffer. The user function must be of type <i>ssi_tCallback</i>: <pre>typedef void (*ssi_tCallback) (void *pCallbackArg, int MaxNBytes);</pre> The parameters to the callback function are: <i>pCallbackArg</i> - The original value specified by the user for SSI_RX_CALLBACK_ARG <i>MaxNBytes</i> - The number of 8-bit bytes currently in the driver's RX buffers; the user application may then <i>read</i> up to <i>MaxNBytes</i> bytes. Note that the maximum number of 16-bit samples is <i>MaxNBytes</i>/2 Default = NULL
SSI_RX_CALLBACK_ARG	A value that the user would like passed to the SSI_RX_CALLBACK function. This value can be anything desired by the user. It is typically defined as a pointer to a device-specific data structure or an index into an application table. This argument is most useful for sharing code between multiple instances of a device, channel, or data structure. The driver does nothing with this value other than pass it back to the user application as a parameter when calling the SSI_RX_CALLBACK function. Default =NULL
SSI_TX_CALLBACK_LEVEL	The minimum number of 16-bit samples that the SSI must maintain in the transmit FIFO buffer. If the number of samples falls below SSI_TX_CALLBACK_LEVEL, the driver calls the user function specified by SSI_TX_CALLBACK. The value specified must be less than SSI_TX_FIFO_SIZE. Increasing the SSI_TX_CALLBACK_LEVEL may increase the efficiency of the application, because more data can be transferred between the driver and the application in each callback. However, such an increase may adversely affect latency in the audio channel because more data is buffered internally in the driver before passing it on to the application. Default = 8

Table 5-50. SSI Configuration Items for *appconfig.h*

SSI #define	Explanation and Default Value
SSI_TX_CALLBACK	A user function for the SSI driver to call when the TX buffer has fallen below SSI_TX_CALLBACK_LEVEL samples. The user function must be of type <i>ssi_tCallback</i>: <pre>typedef void (*ssi_tCallback) (void *pCallbackArg, int MaxNBytes);</pre> The parameters to the callback function are: <i>pCallbackArg</i> - The original value specified by the user for SSI_TX_CALLBACK_ARG <i>MaxNBytes</i> - The maximum number of 8-bit bytes that can be written to the driver's TX buffers using the <i>write</i> function; note that the maximum number of 16-bit samples is MaxNBytes/2 Default = NULL
SSI_TX_CALLBACK_ARG	A value that the user would like passed to the SSI_TX_CALLBACK function. This value can be anything desired by the user. It is typically defined as a pointer to a device specific data structure or an index into an application table. This argument is most useful for sharing code between multiple instances of a device, channel, or data structure. The driver does nothing with this value, other than pass it back to the user application as a parameter when calling the SSI_TX_CALLBACK function. Default = NULL
SSI_NUM_TIME_SLOTS	The number of time slots to be used when configured for network mode. Default = 2
SSI_HW_FIFO_DEPTH	The SSI contains a Receive and Transmit hardware FIFO that can buffer up to 8 samples. This value defines how many samples the ISR will process. Default = 8
SSI_RX_FIFO_FULL_WM	When the RX FIFO contains this many samples, the SSI Rx ISR will be called, if enabled. Default = 8
SSI_TX_FIFO_EMPTY_WM	When the TX FIFO contains this many empty samples, the SSI Tx ISR will be called, if enabled. Default = 8

Table 5-50. SSI Configuration Items for *appconfig.h*

SSI #define	Explanation and Default Value
SSI_PRESCALER	Bit 15 of SSI_RXTX_CONTROL_INITIAL_STATE Valid values are: SSI_PRESCALER_1 SSI_PRESCALER_8 Default = SSI_PRESCALER_1
SSI_WORD_LENGTH	Bits 13 and 14 of SSI_RXTX_CONTROL_INITIAL_STATE Valid values are: SSI_WORD_LENGTH_8 SSI_WORD_LENGTH_10 SSI_WORD_LENGTH_12 SSI_WORD_LENGTH_16 Default = SSI_WORD_LENGTH_16
SSI_DC	Used by SSI_FRAME_RATE_DIVIDER Valid values are: 1 to 32 Default = SSI_NUM_TIME_SLOTS
SSI_PM	Used by SSI_FRAME_RATE_DIVIDER Valid values are: 1 to 256 Default = 8
SSI_FRAME_RATE_DIVIDER	Bits 8 to 12 of SSI_RXTX_CONTROL_INITIAL_STATE Default = ((SSI_DC-1) << 8)
SSI_PRESCALE_MODULUS	Bits 0 to 7 of SSI_RXTX_CONTROL_INITIAL_STATE Default = (SSI_PM-1)
SSI_RXTX_CONTROL_INITIAL_STATE	Initial state of the SSI Rx and Tx Control Registers (SSI_PRESCALER SSI_WORD_LENGTH SSI_FRAME_RATE_DIVIDER SSI_PRESCALE_MODULUS)
SSI_DIV4DIS	Bit 15 of SSI_CONTROL_STATUS_INITIAL_STATE Valid values are: SSI_DIVIDE_MCU_CLOCK_1 SSI_DIVIDE_MCU_CLOCK_4 Default = SSI_DIVIDE_MCU_CLOCK_1
SSI_RSHFD	Bit 14 of SSI_CONTROL_STATUS_INITIAL_STATE Valid values are: SSI_RX_SHIFT_DIR_MSB_FIRST SSI_RX_SHIFT_DIR_LSB_FIRST Default = SSI_RX_SHIFT_DIR_MSB_FIRST

Table 5-50. SSI Configuration Items for *appconfig.h*

SSI #define	Explanation and Default Value
SSI_RSCKP	Bit 13 of SSI_CONTROL_STATUS_INITIAL_STATE Valid values are: SSI_RX_CLK_POL_FALLING_EDGE SSI_RX_CLK_POL_RISING_EDGE Default = SSI_RX_CLK_POL_FALLING_EDGE
SSI_RFSI	Bit 10 of SSI_CONTROL_STATUS_INITIAL_STATE Valid values are: SSI_RX_FRAME_SYNC_ACTIVE_HI SSI_RX_FRAME_SYNC_ACTIVE_LOW Default = SSI_RX_FRAME_SYNC_ACTIVE_HI
SSI_RFSL	Bit 9 of SSI_CONTROL_STATUS_INITIAL_STATE Valid values are: SSI_RX_ONE_WORD_FRAME_SYNC SSI_RX_ONE_BIT_FRAME_SYNC Default = SSI_RX_ONE_BIT_FRAME_SYNC
SSI_REFS	Bit 8 of SSI_CONTROL_STATUS_INITIAL_STATE Valid values are: SSI_RX_FRAME_SYNC_EARLY SSI_RX_FRAME_SYNC_FIRST_BIT Default = SSI_RX_FRAME_SYNC_EARLY
SSI_CONTROL_STATUS_INITIAL_STATE	Initial state of the SSI Control/Status Register (SSI_DIV4DIS SSI_RSHFD SSI_RSCKP SSI_RFSI SSI_RFSL SSI_REFS)
SSI_RIE	Bit 15 of SSI_CONTROL2_INITIAL_STATE Valid values are: SSI_RX_INTERRUPT_DISABLE SSI_RX_INTERRUPT_ENABLE Default = SSI_RX_INTERRUPT_ENABLE
SSI_RE	Bit 13 of SSI_CONTROL2_INITIAL_STATE Valid values are: SSI_RX_DISABLE SSI_RX_ENABLE Default = SSI_RX_DISABLE
SSI_RFEN	Bit 11 of SSI_CONTROL2_INITIAL_STATE Valid values are: SSI_RX_FIFO_DISABLE SSI_RX_FIFO_ENABLE Default = SSI_RX_FIFO_ENABLE

Table 5-50. SSI Configuration Items for *appconfig.h*

SSI #define	Explanation and Default Value
SSI_RXDIR	Bit 9 of SSI_CONTROL2_INITIAL_STATE Valid values are: SSI_RX_CLK_INTERNAL SSI_RX_CLK_EXTERNAL Default = SSI_RX_CLK_EXTERNAL
SSI_TIE	Bit 14 of SSI_CONTROL2_INITIAL_STATE Valid values are: SSI_TX_INTERRUPT_DISABLE SSI_TX_INTERRUPT_ENABLE <i>appconfig.h</i>
SSI_TE	Bit 12 of SSI_CONTROL2_INITIAL_STAT Valid values are: SSI_TX_DISABLE SSI_TX_ENABLE Default = SSI_TX_DISABLE
SSI_TFEN	Bit 10 of SSI_CONTROL2_INITIAL_STATE Valid values are: SSI_TX_FIFO_DISABLE SSI_TX_FIFO_ENABLE Default = SSI_TX_FIFO_ENABLE
SSI_TXDIR	Bit 8 of SSI_CONTROL2_INITIAL_STATE Valid values are: SSI_TX_CLK_INTERNAL SSI_TX_CLK_EXTERNAL Default = SSI_TX_CLK_EXTERNAL
SSI_SYN	Bit 7 of SSI_CONTROL2_INITIAL_STATE Valid values are: SSI_SYNC_MODE_DISABLED SSI_SYNC_MODE_ENABLED Default = SSI_SYNC_MODE_ENABLED
SSI_TSHFD	Bit 6 of SSI_CONTROL2_INITIAL_STATE Valid values are: SSI_TX_SHIFT_DIR_LSB_FIRST SSI_TX_SHIFT_DIR_MSB_FIRST Default = SSI_TX_SHIFT_DIR_MSB_FIRST
SSI_TSCKP	Bit 5 of SSI_CONTROL2_INITIAL_STATE Valid values are: SSI_TX_CLK_POL_FALLING_EDGE SSI_TX_CLK_POL_RISING_EDGE Default = SSI_TX_CLK_POL_RISING_EDGE

Table 5-50. SSI Configuration Items for *appconfig.h*

SSI #define	Explanation and Default Value
SSI_SSIEN	Bit 4 of SSI_CONTROL2_INITIAL_STATE Valid values are: SSI_DISABLE SSI_ENABLE Default = SSI_ENABLE
SSI_NET	Bit 3 of SSI_CONTROL2_INITIAL_STATE Valid values are: SSI_NET_MODE_DISABLED SSI_NET_MODE_ENABLED Default = SSI_NET_MODE_ENABLED
SSI_TFSI	Bit 2 of SSI_CONTROL2_INITIAL_STATE Valid values are: SSI_TX_FRAME_SYNC_ACTIVE_LOW SSI_TX_FRAME_SYNC_ACTIVE_HI Default = SSI_TX_FRAME_SYNC_ACTIVE_HI
SSI_TFSL	Bit 1 of SSI_CONTROL2_INITIAL_STATE Valid values are: SSI_TX_ONE_BIT_FRAME_SYNC SSI_TX_ONE_WORD_FRAME_SYNC Default = SSI_TX_ONE_BIT_FRAME_SYNC
SSI_TEFS	Bit 0 of SSI_CONTROL2_INITIAL_STATE Valid values are: SSI_TX_FRAME_SYNC_EARLY SSI_TX_FRAME_SYNC_FIRST_BIT Default = SSI_TX_FRAME_SYNC_EARLY
SSI_CONTROL2_INITIAL_STATE	Initial state of the SSI Control 2 Register (SSI_RIE SSI_RE SSI_RFEN SSI_RXDIR SSI_TIE SSI_TE SSI_TFEN SSI_TXDIR SSI_SYN SSI_TSHFD SSI_TSCKP SSI_SSIEN SSI_NET SSI_TFSI SSI_TFSL SSI_TEFS)
SSI_RFWM	Actual bits in the FIFO Control Status Register defining the Rx FIFO Full Watermark Default = SSI_RX_FIFO_FULL_WM<<4
SSI_TFWM	Actual bits in the FIFO Control Status Register defining the Tx FIFO Empty Watermark. Default = SSI_TX_FIFO_EMPTY_WM
SSI_FIFO_CNTL_STAT_INITIAL_STATE	Initial state of the SSI FIFO Control/Status Register (SSI_RFWM SSI_TFWM)

Table 5-50. SSI Configuration Items for *appconfig.h*

SSI #define	Explanation and Default Value
SSI_RFDIR	Bit 5 of SSI_OPTION_REGISTER_INITIAL_STATE Valid values are: SSI_RX_FRAME_SYNC_INTERNAL SSI_RX_FRAME_SYNC_EXTERNAL Default = SSI_RX_FRAME_SYNC_EXTERNAL
SSI_TFDIR	Bit 4 of SSI_OPTION_REGISTER_INITIAL_STATE Valid values are: SSI_TX_FRAME_SYNC_INTERNAL SSI_TX_FRAME_SYNC_EXTERNAL Default = SSI_TX_FRAME_SYNC_EXTERNAL
SSI_INIT	Bit 3 of SSI_OPTION_REGISTER_INITIAL_STATE Valid values are: SSI_INIT_RESET_STATE_OFF SSI_INIT_RESET_STATE_ON Default = SSI_INIT_RESET_STATE_OFF
SSI_SYNRST	Bit 0 of SSI_OPTION_REGISTER_INITIAL_STATE Valid values are: SSI_FRAME_SYNC_RESET_OFF SSI_FRAME_SYNC_RESET_ON Default = SSI_FRAME_SYNC_RESET_OFF
SSI_OPTION_REGISTER_INITIAL_STATE	Initial state of the SSI Option Register (SSI_RFDIR SSI_TFDIR SSI_INIT SSI_SYNRST)

5.7.3 API Definition

This section defines the Application Programming Interface (API) by identifying all public interface functions and data structures.

To automatically include code to create and initialize the SSI driver in an SDK Embedded Project, insert the following line in the *appconfig.h* file:

```
#define INCLUDE_SSI
```

The following information may be found in the public header file *ssi.h*:

Public Interface Function(s):

```

/*****
*      types_tHandle open(const char *pName, int OFlags);
*      ssize_t write(types_tHandle FileDesc, const void * pBuffer, size_t NBytes);
*      ssize_t read(types_tHandle FileDesc, void * pBuffer, size_t NBytes);
*      int      close(types_tHandle FileDesc);
*      UWord16 ioctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams);
*****/

```



```

/*****
*      types_tHandle ssiOpen(const char *pName, int OFlags);
*      ssize_t ssiWrite(types_tHandle FileDesc, const void * pBuffer, size_t NBytes);
*      ssize_t ssiRead(types_tHandle FileDesc, void * pBuffer, size_t NBytes);
*      int ssiClose(types_tHandle FileDesc);
*      UWord16 ssiIoctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams,
*                      const char * pName);
*****/

```

Public Data Structure(s):

None

5.7.4 SSI Device Independent I/O API Specifications

The following pages specify the device-independent Application Programming Interface (API).

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

5.7.4.1 *open*

Call(s):

```
types_tHandle open(const char *pName, int OFlags);
```

Arguments:

Table 5-51. SSI Driver Arguments - *open*

<i>pName</i>	<i>in</i>	The SSI device name; see Table 5-52
<i>OFlag</i>	<i>in</i>	Open Mode Flags; specify O_RDWR Use O_NONBLOCK flags for Non-Blocking mode
<i>FileDesc</i>	<i>out</i>	SSI device descriptor returned by <i>open</i> call and used in <i>read/write</i> function calls

Table 5-52. SSI Devices for DSP56852 Chip

Chip	Can Use
DSP56852	BSP_DEVICE_NAME_SSI_0

Description: The *open* call enables the SSI peripheral for operations and returns a file descriptor used by the SSI driver's *read*, *write*, *close*, and *ioctl* function calls. The SSI device must be opened for *read/write* calls.

Returns: The *open* call returns a type *types_tHandle*. When the *open* call is successful, this *types_tHandle* is a file descriptor for the SSI device and is used by the SSI's *read*, *write*, and *ioctl* functions. A "-1" is returned when the *open* call is not successful.

Code Example: See [Code Example 5-21](#).

5.7.4.2 *write*

Call(s):

```
ssize_t write(types_tHandle FileDesc, const void * pBuffer, size_t NBytes);
```

Arguments:

Table 5-53. SSI Driver Arguments - *write*

<i>FileDesc</i>	<i>in</i>	SSI device descriptor returned by <i>open</i> call
<i>pBuffer</i>	<i>in</i>	Pointer to user buffer; the write function type casts <i>pBuffer</i> to be a (<i>UWord16 *</i>).
<i>NBytes</i>	<i>in</i>	Number of bytes to write

Description: The *write* call writes data from the user buffer to the SSI device.

Returns: The *write* call returns a type *ssize_t*, which is the number of transferred bytes.

Code Example: See [Code Example 5-21](#).

5.7.4.3 *read*

Call(s):

```
ssize_t read(types_tHandle FileDesc, void * pBuffer, size_t NBytes);
```

Arguments:

Table 5-54. SSI Driver Arguments - *read*

<i>FileDesc</i>	<i>in</i>	SSI device descriptor returned by <i>open</i> call
<i>pBuffer</i>	<i>in</i>	Pointer to user buffer. The read function type casts <i>pBuffer</i> to be a (<i>UWord16 *</i>).
<i>Size</i>	<i>in</i>	Number of bytes to read

Description: The *read* call reads data from the SSI device into the user buffer.

Returns: The *read* call returns a type *ssize_t*, which is the number of received bytes.

Code Example: None

5.7.4.4 *close*

Call(s):

```
int close (types_tHandle FileDesc);
```

Arguments:

Table 5-55. SSI Driver Arguments - *close*

<i>FileDesc</i>	<i>in</i>	SSI device descriptor returned by <i>open</i> call

Description: The *close* call disables the SSI peripheral.

Returns: The *close* call returns a type *int*. This *int* is always zero.

Code Example: See [Code Example 5-21](#).

5.7.4.5 *ioctl*

Call(s):

```
UWord16 ioctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams);
```

Arguments:

Table 5-56. SSI Driver Arguments - *ioctl*

<i>FileDesc</i>	<i>in</i>	SSI device descriptor returned by <i>open</i> call
<i>Cmd</i>	<i>in</i>	Commands found in <i>ssi.h</i> which are used to manage the SSI driver
<i>pParams</i>	<i>in</i>	Pointer to <i>ioctl</i> command parameter

Table 5-57. SSI *ioctl* Commands

Cmd	<i>pParams</i>	Return	Result
SSI_DEVICE_RESET	NULL	None	Sets new SSI configuration and resets device; cancels all non-blocking operations
SSI_DEVICE_ENABLE	NULL	None	Disables device
SSI_DEVICE_DISABLE	NULL	None	Enables device
SSI_DEVICE_SET_RX_CALL BACK_LEVEL	UWord16 length	None	Sets the number of words/bytes for Read callback; after this number of words is received, the Receive callback is called
SSI_DEVICE_SET_TX_CALL BACK_LEVEL	UWord16 length	None	Sets the number of words/bytes for Write callback.; when this number of words is left in the Transmit FIFO, the Write callback is called

Description: The *ioctl* call changes SSI device modes or reports SSI driver status.

Returns: The *ioctl* call returns a type *UWord16*. This *UWord16* is dependent on the command passed in as an input parameter. **Table 5-57** lists these returns based on the specific command.

Code Example: **Code Example 5-21** sets up the SSI to write data out the serial port.

Code Example 5-21. SSI Driver Usage

```

/*****
#include "port.h"
#include "io.h"
#include "fcntl.h"
#include "bsp.h"

#include "ssi.h"

```



```
void main(void)
{
    types_tHandle SsiDevice;
    UWord16      Data;

    /* open ssi driver */
    SsiDevice = open(BSP_DEVICE_NAME_SSI_0, O_RDWR);

    Data = 0x0030;

    /* Send one sample via SSI */
    write(SsiDevice, (void *)&Data, sizeof(UWord16));

    /* close ssi driver */
    close(SsiDevice);
}
/*****/
```



5.7.5 SSI Low-Level I/O Specifications

The following pages specify the low-level Application Programming Interface (API).

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

5.7.5.1 *ssiOpen*

Call(s):

```
types_tHandle ssiOpen(const char *pName, int OFlags);
```

Arguments:

Table 5-58. SSI Driver Arguments - *ssiOpen*

<i>pName</i>	<i>in</i>	The SSI device name; see Table 5-59
<i>OFlag</i>	<i>in</i>	Open Mode Flags; specify O_RDWR Use O_NONBLOCK flags for non-blocking mode
<i>FileDesc</i>	<i>out</i>	SSI device descriptor returned by <i>ssiOpen</i> call and used in <i>ssiRead</i> / <i>ssiWrite</i> function calls

Table 5-59. SSI Devices for DSP56852 Chip

Chip	Can Use
DSP56852	BSP_DEVICE_NAME_SSI_0

Description: The *ssiOpen* call enables the SSI peripheral for operations and returns a file descriptor used by the SSI driver's *ssiRead*, *ssiWrite*, *ssiClose*, and *ssiIoctl* function calls. The SSI device must be opened for *read/write* calls.

Returns: The *ssiOpen* call returns a type *types_tHandle*. When the *ssiOpen* call is successful, this *types_tHandle* is a file descriptor for the SSI device and is used by the SSI *ssiRead*, *ssiWrite*, and *ssiIoctl* functions. A "-1" is returned when the *ssiOpen* call is not successful.

Code Example: See [Code Example 5-22](#).

5.7.5.2 *ssiWrite*

Call(s):

```
ssize_t ssiWrite(types_tHandle FileDesc, const void * pBuffer, size_t NBytes);
```

Arguments:

Table 5-60. SSI Driver Arguments - *ssiWrite*

<i>FileDesc</i>	<i>in</i>	SSI device descriptor returned by <i>ssiOpen</i> call
<i>pBuffer</i>	<i>in</i>	Pointer to user buffer; the write function type casts <i>pBuffer</i> to be a (<i>UWord16 *</i>)
<i>NBytes</i>	<i>in</i>	Number of bytes to write

Description: The *ssiWrite* call writes data from the user buffer to the SSI device.

Returns: The *ssiWrite* call returns a type *ssize_t*, which is the number of transferred bytes.

Code Example: See [Code Example 5-22](#).

5.7.5.3 ssiRead

Call(s):

```
ssize_t ssiRead(types_tHandle FileDesc, void * pBuffer, size_t NBytes);
```

Arguments:

Table 5-61. SSI Driver Arguments - ssiRead

<i>FileDesc</i>	<i>in</i>	SSI device descriptor returned by <i>ssiOpen</i> call
<i>pBuffer</i>	<i>in</i>	Pointer to user buffer. The read function type casts <i>pBuffer</i> to be a (<i>UWord16 *</i>)
<i>Size</i>	<i>in</i>	Number of bytes to read

Description: The *ssiRead* call reads data from SSI device into the user buffer.

Returns: The *ssiRead* call returns a type *ssize_t*, which is the number of received bytes.

Code Example: None



5.7.5.4 *ssiClose*

Call(s):

```
int ssiClose (types_tHandle FileDesc);
```

Arguments:

Table 5-62. SSI Driver Arguments - *ssiClose*

<i>FileDesc</i>	<i>in</i>	SSI device descriptor returned by <i>ssiOpen</i> call

Description: The *ssiClose* call disables the SSI peripheral.

Returns: The *ssiClose* call returns a type *int*. This *int* is always zero.

Code Example: See [Code Example 5-22](#).

5.7.5.5 *ssioctl*

Call(s):

```
UWord16 ssiIoctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams,
                  const char *pName);
```

Arguments:

Table 5-63. SSI Driver Arguments - *ssioctl*

<i>FileDesc</i>	<i>in</i>	SSI device descriptor returned by <i>ssiOpen</i> call
<i>Cmd</i>	<i>in</i>	Commands found in <i>ssi.h</i> which are used to manage the SSI driver
<i>pParams</i>	<i>in</i>	Pointer to <i>ssioctl</i> command parameter
<i>pName</i>	<i>in</i>	The SSI device name from <i>bsp.h</i> ; typically, BSP_DEVICE_NAME_SSI_0

Table 5-64. SSI *ssioctl* Commands

Cmd	<i>pParams</i>	Return	Result
SSI_DEVICE_RESET	NULL	None	Sets new SSI <i>new</i> configuration and resets device; cancels all non-blocking operations
SSI_DEVICE_ENABLE	NULL	None	Disables device
SSI_DEVICE_DISABLE	NULL	None	Enables device
SSI_DEVICE_SET_RX_CALL BACK_LEVEL	UWord16 length	None	Set the number of words/bytes for Read callback; after this number of words is received, the Receive callback is called
SSI_DEVICE_SET_TX_CALL BACK_LEVEL	UWord16 length	None	Sets the number of words/bytes for Write callback; when this number of words is left in the Transmit FIFO, the Write callback is called

Description: The *ssioctl* call changes SSI device modes or reports SSI driver status.

Returns: The *ssioctl* call returns a type *UWord16*. This *UWord16* is dependent on the command passed in as an input parameter. [Table 5-64](#) lists these returns based on the specific command.

Code Example: [Code Example 5-22](#) sets up the SSI to write data out the serial port.

Code Example 5-22. SSI Driver Usage

```

/*****
#include "port.h"
#include "fcntl.h"
#include "bsp.h"

#include "ssi.h"

```



```

void main(void)
{
    types_tHandle SsiDevice;
    UWord16      Data;

    /* open ssi driver */
    SsiDevice = ssiOpen(BSP_DEVICE_NAME_SSI_0, 0_RDWR);

    Data = 0x0030;

    /* Send one byte via SSI */
    ssiWrite(SsiDevice, (void *)&Data, sizeof(UWord16));

    /* close ssi driver */
    ssiClose(SsiDevice);
}
/*****/

```

5.7.6 SSI Driver Tests and Demos

Please refer to the Codec Driver application in [Section 6.2](#). The codec application uses the Codec driver, which is implemented using the SSI driver.

5.8 Timer Driver

5.8.1 Introduction

The Timer driver interface is implemented through SDK timer services. The SDK includes a timer service whose purpose is to furnish a device-independent, standard method both for integrating timer drivers into the system and for calling these timer drivers from the user's application. Please refer to **Chapter 5.1.5 Timer Services of the SDK Programmer's Guide** for definitions of the Timer Device-Independent I/O Public Interface.

5.8.2 Timer Driver Tests

Currently, there are no stand-alone timer driver tests in the SDK.

5.8.3 Timer Driver Demos

5.8.3.1 Timer Driver Flashing LEDs Demo

This demo exercises Timer0, Timer1, and Timer2 peripherals located in the DSP56852 processor. It initializes Timer1 using clock ID `CLOCK_AUX1` to 250ms with a reload value of 250ms. Timer2 is initialized using clock ID `CLOCK_AUX2` to 125ms with a reload value of 125ms. When these timers expire, the application changes the on/off state of the yellow and red LEDs. The net effect of this is that the yellow LED will come on once every 0.5 seconds, and the red LED will come on once every 0.25 seconds.

For interaction with Timer0, the application uses the SDK timer services. The SDK timer services reserve Timer0 with clock ID `CLOCK_REALTIME`, and utilizes this timer for *nanosleep* functionality. This application calls the *nanosleep* interface with a timeout value of 0.5 seconds in a tight loop. The net effect of this call will be suspension of the *main* function execution for 0.5 seconds.

5.8.3.1.1 Set-up for Timer Driver Flashing LEDs Demo

EVM Jumper Settings:

Use default settings as shown in the **DSP56852 Evaluation Module User's Manual**.

5.8.3.1.2 Procedure for Timer Driver Flashing LEDs Demo

- Using CodeWarrior, open:
...\\nos\\applications\\timers\\timers.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the demo
- Once the application is running, you should see the green LED come on every 1 second, the yellow LED come on every 0.5 seconds, and the red LED come on every 0.25 seconds



Chapter 6

Off-Chip Drivers

One strength of the SDK is that it provides a high degree of architectural and hardware independence for application code. This portability is due to the SDK's modular design, which, in this case, isolates all EVM board-specific functionality into a set of libraries that are part of Board Support Package (BSP).

This BSP can be found at `\src\dsp56852evm\nos\bsp` of the SDK. This chapter is composed of two major components: Interfaces and Drivers. This document focuses on specific DSP56852EVM board device drivers only. Specific on-chip device drivers are addressed in [Chapter 5](#).

6.1 Button Driver

6.1.1 Introduction

The Button driver provides a software interface to the push buttons on the DSP56852EVM board.

6.1.2 Initialization

The Button driver currently has no default configurations that can be overwritten through a project's *appconfig.h* file.

To include the Button Driver in the application, insert the following line into the *appconfig.h* file:

```
#define INCLUDE_BUTTON
```

Defining the preprocessor variable, `INCLUDE_BUTTON`, directs the SDK to include the Button driver in the executable image. If the preprocessor variable `INCLUDE_BUTTON` is not defined, then the `BUTTON` driver will not be included in the executable.

6.1.3 API Definition

This section defines the Application Programming Interface (API) by identifying all public interface functions and data structures.

The Button interface can be used at two alternative levels: a common IO layer interface and a low-level Button driver interface. The common IO layer interface invokes the lower-level Button driver interface.

The IO layer API to the Button interface is standard and more portable than the low-level Button interface, but potentially less efficient. The low-level Button driver provides a non-standard interface that is potentially more efficient than the IO layer calls, but less portable.

Your application may use either the IO layer interface or the low-level Button driver interface to the Button driver, depending on your specific goals for efficiency and portability. However, you should not mix the Button driver APIs in your application. In particular, you **must not** use the File Descriptor returned by the IO layer *open* function with any low-level Button Driver function call, nor may you use the File Descriptor returned by the low-level driver *buttonOpen* call with any IO layer interface function call.

To utilize the Button Driver API, you must include *button.h* in your application. The following information may be found in the public header file *button.h*:

Public Interface Function(s):

```

/*****
 *
 * typedef void (*button_tCallback)(void *pCallbackArg);
 * typedef struct
 * {
 *     button_tCallback    pCallback;
 *     void                *pCallbackArg;
 * } button_sCallback;
 *
 * The IO layer interface defines functions as follows:
 *     types_tHandle open(const char *pName, int OFlags, button_sCallback * pCallbackSpec);
 *     int close(types_tHandle FileDesc);
 *
 * The low level Button driver interface defines functions as follows:
 *     types_tHandle buttonOpen(const char *pName, int OFlags,
 *                             button_sCallback * pCallbackSpec);
 *     int buttonClose(types_tHandle FileDesc);
 *
 * *****/

```

Public Data Structure(s):

None

Members:

None



6.1.4 Button Device-Independent I/O API Specifications

The following pages specify the device-independent Application Programming Interface (API).

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

6.1.4.1 *open*

Call(s):

```
types_tHandle open(const char *pName, int OFlags,
                  button_sCallback * pCallbackSpec);
```

Arguments:

Table 6-1. Button Driver Arguments - *open*

<i>pName</i>	<i>in</i>	The Button device name; see Table 6-2
<i>OFlag</i>	<i>in</i>	Open Mode Flags which are ignored
<i>pCallbackSpec</i>	<i>in</i>	The function to be called when a button is pushed
<i>FileDesc</i>	<i>out</i>	Button device descriptor returned by <i>open</i> call and used in <i>ioctl</i> function calls

Table 6-2. Availability of Button Devices for Specific Platforms

DSP56852EVM
BSP_DEVICE_NAME_BUTTON_A BSP_DEVICE_NAME_BUTTON_B

Description: The *open* call opens the Button driver for operations and returns a file descriptor used by other Button driver function calls.

The callback function, specified by *pCallbackSpec*, is called whenever the specified push button is pressed. The button is debounced to ensure that the callback function is called only once per button press.

Returns: The *open* call returns a type *types_tHandle*. When the *open* call is successful, this *types_tHandle* is a file descriptor for the Button driver and is used by the Button driver's *close* function. A "-1" is returned when the *open* call is not successful.

Code Example: See [Code Example 6-2](#).

6.1.4.2 *close*

Call(s):

```
int close (types_tHandle FileDesc);
```

Arguments:

Table 6-3. Button Driver Arguments - *close*

<i>FileDesc</i>	<i>in</i>	Button device descriptor returned by <i>open</i> call and used in driver function calls

Description: The *close* call closes the Button driver.

Returns: The *close* call returns a type *int*; this *int* is always zero.

Code Example: [Code Example 6-1](#) shows how a user function will be called when push button A is pressed on the EVM board.

Code Example 6-1. Button Driver Example

```

/*****
#include "io.h"
#include "fcntl.h"
#include "bsp.h"

#include "button.h"

/* button isr */
void ButtonISR (void *pCallbackArg)
{
... /* logic to execute when button is pushed */
}

static types_tHandle ButtonAFD;
button_sCallback ButtonSpec = {ButtonISR, NULL};

/* open button driver */
ButtonAFD = open(BSP_DEVICE_NAME_BUTTON_A, 0, &ButtonSpec);
...

/* close button driver */
close(ButtonFD);

*****/

```



6.1.5 Low-Level I/O Services API Specifications

The following pages specify the low-level Application Programming Interface (API).

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

6.1.5.1 *buttonOpen*

Call(s):

```
types_tHandle buttonOpen(const char *pName, int OFlags,
                        button_sCallback * pCallbackParam);
```

Table 6-4. Button Driver Arguments - *buttonOpen*

<i>pName</i>	<i>in</i>	The name of the particular device
<i>OFlag</i>	<i>in</i>	Open Mode Flags which are ignored
<i>pCallbackParam</i>	<i>in</i>	A pointer to a structure which specifies a function to call when the button is pressed

Description: Opens a particular BUTTON device. The argument *pName* is the particular device name.

The *pCallbackParam* is a pointer to a structure that specifies which user function to call when the button is pressed. This structure contains two values: the address of the function to be called and the value of the parameter to that function. The parameter value can be used in the function for such purposes as designating which button was pressed or to refer to an application-specific structure which must be modified by the callback function.

Returns: If *open* is successful, a file descriptor is returned. This file descriptor must be passed to other Button driver functions. If *open* failed, a “-1” value is returned.

Code Example: See [Code Example 6-2](#).

6.1.5.2 *buttonClose*

Call(s):

```
int buttonClose (types_tHandle FileDesc);
```

Arguments:

Table 6-5. Button Driver Arguments - *buttonClose*

<i>FileDesc</i>	<i>in</i>	Button device descriptor returned by <i>buttonOpen</i> call and used in driver function calls

Description: The *close* call closes the Button driver.

Returns: The *close* call returns a type *int*. This *int* is always zero.

Code Example: See [Code Example 6-2](#).

Code Example 6-2. Low Level Button Driver Example

```

/*****
#include "bsp.h"
#include "button.h"

/* Function prototype */
void ButtonFunc(void *pCallbackArg);

volatile int buttonAcounter;

/* ButtonFunc Function */
void ButtonFunc (void *pCallbackArg)
{
    volatile int * pCounter = (volatile int *)pCallbackArg;

    (*pCounter)++;
}

void main(void)
{
    types_tHandle buttonFD;

    button_sCallback ButtonACallbackSpec = {ButtonFunc, (void*) &buttonAcounter};

    /* Open button driver */
    buttonFD = buttonOpen(BSP_DEVICE_NAME_BUTTON_A, 0, (void*)
    &ButtonACallbackSpec);

    // .....

    /* Close button driver */
    buttonClose(buttonFD);
}
*****/

```

6.1.6 Button Driver Tests

The Button Driver Tests verify the correct operation of the Button Driver provided by the SDK.

6.1.6.1 Button Driver Test

This test verifies detection of Button A and Button B.

6.1.6.1.1 Set-up for Button Driver Test

EVM Jumper Settings:

Use default settings as shown in the **DSP56852 Evaluation Module User's Manual**.

6.1.6.1.2 Procedure for Button Driver Test

- Using CodeWarrior, open:
...\\nos\\bsp\\test\\button\\button.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- Once the debugger is running, follow the instructions in the CodeWarrior console window
 - The test prompts the user to press Button A
 - After detecting that Button A has been pressed, the test prompts the user to press Button B
 - When it detects that Button B has been pressed, it successfully ends the test

6.1.7 Button Driver Applications

There is not a specific application demonstrating just the Button driver.

6.2 Codec Driver

6.2.1 Introduction

This driver is designed for the Crystal CS4218 Stereo Codec hardware located on the DSP56852EVM board. The Crystal CS4218 codec is configurable for many modes. Please refer to the Crystal CS4218 user's manual for detailed information about the codec.

The DSP56852EVM configures the codec to be in Mode 4, which has separate data and control channels. The DSP56852EVM uses GPIO pins on Port C and E to interface to the codec control channel. The information used to configure the codec can be modified by the user via the SDK by defining the new codec configuration in the *appconfig.h* file. This will overwrite the default configuration in the *config.h* file and will be used to configure the codec when the codec device is opened.

The data channel transmits and receives 32 bits at a time. The first 16 bits are for the left channel and the second 16 bits are for the right channel. A single frame sync is generated for each 32-bit sample. The Codec driver configures the SSI for network mode to handle this case (one frame sync for multiple samples). The SSI has also been configured to use its FIFO to receive two 16-bit samples before causing an interrupt. By doing this, only a single interrupt is required to read both the left and right samples.

The Codec driver can be configured for either stereo mode or mono mode. In stereo mode, all samples are placed in the user's buffer and are interlaced. The first sample is from the left channel, the second sample is from the right channel, the third sample is from the left channel, and so on. In mono mode, the Codec driver receives samples only from the left channel and places the samples in the user's buffer; upon transmit, the driver uses a single sample from the user to drive both the left and right channels with the same signal.

The Codec driver also configures the SSI to synchronize the receive and transmit channels which means only one interrupt service routine (ISR) is needed. The Codec driver uses the SSI transmit interrupt to read and transmit samples to and from the codec. When an interrupt occurs, the ESSIO transmit FIFO is empty. The ISR transmits eight new samples and, since the receiver is synchronized to the transmitter, reads eight samples from the receiver. By combining the receive and transmit interrupts, less overhead is required when processing the interrupts.

The Codec driver provides the following features:

- Blocking and non-blocking modes
- Function callbacks to notify the user when data is received or when data is transmitted
- User specified data levels to packetize the amount of data available to/from callback functions, thus minimizing processing overhead
- Static configuration parameters to establish initial Codec settings for gain; attenuation; mode (stereo/mono); buffer size; and user callback functions
- Dynamic control functions to change channel gain and attenuation; enable/disable the codec; reset codec operation; and mute all sound

The following sections describe how to statically configure the Codec device driver via the SDK *appconfig.h* file, along with APIs for I/O and control functions.

6.2.2 Initialization

The SDK file *appconfig.h* provides the mechanism to statically configure the Codec driver. The *appconfig.h* file allows the programmer to set C preprocessor variables which configure the Codec driver default operation. Typically, these C preprocessor variables conditionally compile and link information into the application for use by the Codec driver during initial operation.

To include the Codec Driver in the application, insert the following line into the *appconfig.h* file:

```
#define INCLUDE_CODEC
```

Defining the preprocessor variable, `INCLUDE_CODEC`, directs the SDK to include the Codec driver in the executable image. If the preprocessor variable `INCLUDE_CODEC` is not defined, then the Codec driver will not be included in the executable.

You may also define additional preprocessor variables in the *appconfig.h* file to statically configure and initialize the Codec driver. The list of these variables is provided in [Table 6-6](#).

Table 6-6. Codec Configuration Items for *appconfig.h*

Codec #define	Explanation and Default Value
CODEC_MODE	Sets Codec operation to either stereo or mono mode. Valid values are: CODEC_STEREO (default) CODEC_MONO
CODEC_LEFT_D2A_ATTENUATION	Left Channel Output Attenuation in 1.5 dB Increments: 00000 = 0 dB attenuation (default) ... 11111 = 46.5 dB attenuation
CODEC_RIGHT_D2A_ATTENUATION	Right Channel Output Attenuation in 1.5 dB Increments: 00000 = 0 dB attenuation (default) ... 11111 = 46.5 dB attenuation
CODEC_MUTE	Initial state of the Codec Mute Valid values are CODEC_MUTE_DISABLED (default) CODEC_MUTE_ENABLED
CODEC_ISL	Selects the Codec input pin for the left channel Valid values are: CODEC_LEFT_INPUT_LINE_1 (default) CODEC_LEFT_INPUT_LINE_2

Table 6-6. Codec Configuration Items for *appconfig.h* (Continued)

Codec #define	Explanation and Default Value
CODEC_ISR	Selects the Codec input pin for the right channel Valid values are: CODEC_RIGHT_INPUT_LINE_1 (default) CODEC_RIGHT_INPUT_LINE_2
CODEC_LEFT_A2D_GAIN	Left Channel Input Gain in 1.5 dB Increments: 0000 = 0 dB gain 0001 = 1.5 dB gain 0010 = 3.0 dB gain (default) ... 1111 = 22.5 dB gain
CODEC_RIGHT_A2D_GAIN	Right Channel Input Gain in 1.5 dB Increments: 0000 = no gain (0 dB) 0001 = 1.5 dB gain 0010 = 3.0 dB gain (default) ... 1111 = 22.5 dB gain
CODEC_FIFO_SIZE	Default size (number of 16-bit samples) for both the RX FIFO and TX FIFO buffers used internally by the driver. For more information, refer to CODEC_RX_FIFO_SIZE and CODEC_TX_FIFO_SIZE Default = 32
CODEC_RX_FIFO_SIZE	Size (number of 16-bit samples) of the RX FIFO buffer used internally by the driver for the RX channel. The length of the RX FIFO buffer determines the maximum number of Codec input samples that can be buffered within the driver. Note that in stereo mode, the driver inserts two 16-bit samples at a time (one for each channel) into the RX FIFO buffer; in mono mode, the driver inserts only one sample at a time into the RX FIFO buffer. Default = CODEC_FIFO_SIZE
CODEC_TX_FIFO_SIZE	Size (number of 16-bit samples) of the TX FIFO buffer used internally by the driver for the TX channel. The length of the TX FIFO buffer determines the maximum number of Codec output samples that can be buffered within the driver. Note that in stereo mode, the driver extracts two 16-bit samples at a time (one for each channel) from the TX FIFO buffer; in mono mode, the driver extracts only one sample at a time from the TX FIFO buffer. Default = CODEC_FIFO_SIZE

Table 6-6. Codec Configuration Items for *appconfig.h* (Continued)

Codec #define	Explanation and Default Value
CODEC_RX_CALLBACK_LEVEL	The number of 16-bit samples that the Codec must receive before calling the user function specified by CODEC_RX_CALLBACK. The value specified must be less than CODEC_RX_FIFO_SIZE. Increasing the CODEC_RX_CALLBACK_LEVEL may increase the efficiency of the application, because more data can be transferred between the driver and the application in each callback. However, such an increase may adversely affect latency in the audio channel because more data is buffered internally in the driver before passing it on to the application. Default = 8
CODEC_RX_CALLBACK	A user function for the Codec driver to call when it has received at least CODEC_RX_CALLBACK_LEVEL samples in its internal RX buffer. The user function must be of type <i>codec_tCallback</i>: <pre>typedef void (*codec_tCallback) (void *pCallbackArg, int MaxNBytes);</pre> The parameters to the callback function are: <i>pCallbackArg</i> - The original value specified by the user for CODEC_RX_CALLBACK_ARG <i>MaxNBytes</i> - The number of 8-bit bytes currently in the driver's RX buffers; the user application may then <i>read</i> up to <i>MaxNBytes</i> bytes. Note that the maximum number of 16-bit samples is <i>MaxNBytes</i>/2 Default = NULL
CODEC_RX_CALLBACK_ARG	A value that the user would like passed to the CODEC_RX_CALLBACK function. This value can be anything desired by the user. It is typically defined as a pointer to a device specific data structure or an index into an application table. This argument is most useful for sharing code between multiple instances of a device, channel, or data structure. The driver does nothing with this value other than pass it back to the user application as a parameter when calling the CODEC_RX_CALLBACK function. Default = NULL

Table 6-6. Codec Configuration Items for *appconfig.h* (Continued)

Codec #define	Explanation and Default Value
CODEC_TX_CALLBACK_LEVEL	The minimum number of 16-bit samples that the Codec must maintain in the transmit FIFO buffer. If the number of samples falls below CODEC_TX_CALLBACK_LEVEL, the driver calls the user function specified by CODEC_TX_CALLBACK. The value specified must be less than CODEC_TX_FIFO_SIZE. Increasing the CODEC_TX_CALLBACK_LEVEL may increase the efficiency of the application, because more data can be transferred between the driver and the application in each callback. However, such an increase may adversely affect latency in the audio channel because more data is buffered internally in the driver before passing it on to the application. Default is 8
CODEC_TX_CALLBACK	A user function for the Codec driver to call when the TX buffer has fallen below CODEC_TX_CALLBACK_LEVEL samples. The user function must be of type <i>codec_tCallback</i>: <pre>typedef void (*codec_tCallback) (void *pCallbackArg, int MaxNBytes);</pre> The parameters to the callback function are: <i>pCallbackArg</i> - The original value specified by the user for CODEC_TX_CALLBACK_ARG <i>MaxNBytes</i> - The maximum number of 8-bit bytes that can be written to the driver's TX buffers using the <i>write</i> function; note that the maximum number of 16-bit samples is MaxNBytes/2 Default = NULL
CODEC_TX_CALLBACK_ARG	A value that the user would like passed to the CODEC_TX_CALLBACK function. This value can be anything desired by the user. It is typically defined as a pointer to a device specific data structure or an index into an application table. This argument is most useful for sharing code between multiple instances of a device, channel, or data structure. The driver does nothing with this value other than pass it back to the user application as a parameter when calling the CODEC_TX_CALLBACK function. Default = NULL

Table 6-6. Codec Configuration Items for *appconfig.h* (Continued)

Codec #define	Explanation and Default Value
CODEC_MASK	Bit 11 of CODEC_TX_CONTROL_WORD Valid values are: CODEC_INTERRUPT_MASKED (default) CODEC_INTERRUPT_UNMASKED Please see the Crystal CS4218 Stereo Codec hardware reference for more details
CODEC_DO1	Bit 10 of CODEC_TX_CONTROL_WORD Valid values are CODEC_DIGITAL_OUTPUT_1_LOW (default) CODEC_DIGITAL_OUTPUT_1_HIGH Please see the Crystal CS4218 Stereo Codec hardware reference for more details.

Code Example: [Code Example 6-3](#) configures and initializes the Codec driver.

Code Example 6-3. Sample *appconfig.h* code

```

#define INCLUDE_IO          /* include IO layer (optional) */
#define INCLUDE_CODEC       /* include Codec driver (required) */

#define CODEC_MODE          CODEC_MONO

extern void CodecRXISR (void *pCallbackArg, int MaxNum);
extern void CodecTXISR (void *pCallbackArg, int MaxNum);

#define CODEC_RX_CALLBACK    CodecRXISR
#define CODEC_RX_CALLBACK_LEVEL 16

#define CODEC_TX_CALLBACK    CodecTXISR
#define CODEC_TX_CALLBACK_LEVEL 4

```

6.2.3 API Definition

This section defines the Application Programming Interface (API) by identifying all public interface functions and data structures.

The codec interface can be used at two alternative levels: a common IO layer interface and a low-level Codec driver interface. The common IO layer interface invokes the lower-level Codec driver interface.

The IO layer API to the Codec interface is standard and more portable than the low-level codec interface, but potentially less efficient. The low-level Codec driver provides a non-standard interface that is potentially more efficient than the IO layer calls, but less portable.

Your application may use either the IO layer interface or the low-level Codec driver interface to the Codec driver, depending on your specific goals for efficiency and portability. However, you should not mix the Codec driver APIs in your application. In particular, you **must not** use the File Descriptor returned by the IO layer *open* function with any low-level Codec driver function call, nor may you use the File Descriptor returned by the low-level driver *codecOpen* call with any IO layer interface function call.

The following information may be found in the public header file *codec.h*. You must include *codec.h* in your application to utilize the Codec driver API.

Public Interface Function(s):

```

/*****
* The IO layer interface defines functions as follows:
*
*   types_tHandle open(const char *pName, int OFlags);
*   ssize_t read(types_tHandle FileDesc, void * pBuffer, size_t NBytes);
*   ssize_t write(types_tHandle FileDesc, const void * pBuffer, size_t NBytes);
*   UWord16 ioctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams);
*   int close(types_tHandle FileDesc);
*
* The low level Codec driver interface defines functions as follows:
*
*   types_tHandle codecOpen(const char *pName, int OFlags);
*   ssize_t codecRead(types_tHandle FileDesc, void * pBuffer, size_t NBytes);
*   ssize_t codecWrite(types_tHandle FileDesc, const void * pBuffer, size_t NBytes);
*   UWord16 codecIoctl(types_tHandle FileDesc, UWord16 Cmd, codec_sParams * pParams,
*                      const char *pName);
*   int codecClose (types_tHandle FileDesc);
*
*****/

```

Public Data Structure(s):

None

Members:

None

6.2.4 IO Layer API Specification

The following pages specify the Application Programming Interface (API) for the IO Layer Interface to the Codec driver.

To use the IO Layer Interface to the Codec driver you must define both the `INCLUDE_CODEC` variable and the `INCLUDE_IO` variable in the `appconfig.h` file of your SDK Embedded Project created in CodeWarrior:

```
#define INCLUDE_IO
#define INCLUDE_CODEC
```

The user's application code then uses the Codec driver API as follows:

- An *open* call is made to initialize the codec device
- *ioctl* calls are made to control the codec device
- *write* calls are made to write data to the codec (transmit)
- *read* calls are made to read data from the codec (receive)
- After all codec operations are completed, the Codec driver is closed via a *close* call.

The following sections describe each function of the IO Layer API for the Codec driver.

In the following specifications, function arguments for each routine are described as *in*, *out*, or *inout*. An *in* argument means that the parameter value is an input only to the function. An *out* argument means that the parameter value is an output only from the function. 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.

6.2.4.1 *open*

Call(s):

```
types_tHandle open(const char *pName, int OFlags);
```

Arguments:

Table 6-7. Codec Driver Arguments - *open*

<i>pName</i>	<i>in</i>	The codec device name; see the <i>bsp.h</i> file for target-specific values, such as: BSP_DEVICE_NAME_CODEC_0
<i>OFlag</i>	<i>in</i>	Open Mode Flags Valid values are: O_BLOCK (default) O_NONBLOCK
<i>FileDesc</i>	<i>out</i>	Codec device descriptor returned by <i>open</i> call and used in subsequent <i>read/write/close/ioctl</i> function calls

Description: The *open* call opens the Codec driver for operations and returns a file descriptor used by subsequent *read/write/close/ioctl* function calls. The codec device must be opened before any calls are made to other Codec Driver functions.

Return: The *open* call returns a type *types_tHandle*. When the *open* call is successful, this *types_tHandle* is a file descriptor (*FileDesc*) for the codec device and is used by the Codec driver's *read*, *write*, *close*, and *ioctl* functions. A "-1" is returned when the *open* call is not successful.

Code Example: See [Code Example 6-4](#).

6.2.4.2 *write*

Call(s):

```
ssize_t write(types_tHandle FileDesc, const void * pBuffer, size_t NBytes);
```

Arguments:

Table 6-8. Codec Driver Arguments - *write*

<i>FileDesc</i>	<i>in</i>	Codec device descriptor returned by the <i>open</i> call
<i>pBuffer</i>	<i>in</i>	Pointer to user buffer of at least <i>NBytes</i> bytes
<i>NBytes</i>	<i>in</i>	Number of 8-bit bytes to write to the Codec; note that the number of 16-bit samples is <i>NBytes</i> /2

Description: The *write* call writes the data from the user buffer (*pBuffer*) to the Codec driver's internal data structures. User data is 16-bit signed integers representing linear PCM values. Note that the input parameter *NBytes* specifies the size of the data to write as a number of 8-bit bytes.

Returns: The *write* call returns a value of type *ssize_t*. This return value is the actual number of 8-bit bytes transferred. Note that the number of 16-bit samples written is half the return value.

The actual number of data bytes written will be equal to *NBytes* if the driver is operating in blocking mode; see [Section 6.2.4.1](#). In blocking mode, the Codec driver will not return from the *write* function until *NBytes* bytes from the user's buffer are transferred to the driver's internal data structures.

However, the number of data bytes written may be less than or equal to *NBytes* if operating in non-blocking mode. In non-blocking mode, the Codec driver will return immediately from the *write* function after transferring as much of the user's data as possible into the driver's internal data structures.

The *write* function may be called from normal application code, from an ISR, or from the user's callback function (which is called in turn from the Codec driver's ISR). However, avoid calling *write* in blocking mode from an ISR or from a callback function with more data than the Codec driver can immediately transfer (i.e., more than the *MaxNBytes* parameter of the callback function); otherwise, deadlock may occur if Codec interrupts are disabled during the *write*.

Avoid using *write* in both normal application code and from an ISR or callback function. The asynchronous nature of the ISR and callback may lead to re-entrancy issues if you call *write* from normal application code as well.

Code Example: See [Code Example 6-4](#).

6.2.4.3 *read*

Call(s):

```
ssize_t read(types_tHandle FileDesc, void * pBuffer, size_t NBytes);
```

Arguments:

Table 6-9. Codec Driver Arguments - *read*

<i>FileDesc</i>	<i>in</i>	Codec device descriptor returned by the <i>open</i> call
<i>pBuffer</i>	<i>inout</i>	Pointer to a user buffer of at least <i>NBytes</i> bytes
<i>NBytes</i>	<i>in</i>	Number of 8-bit bytes to read; note that the number of 16-bit samples is <i>NBytes/2</i>

Description: The *read* call reads from the Codec driver's internal data structures into the user buffer (*pBuffer*). The data is 16-bit signed integers representing linear PCM values. Note that the input parameter *NBytes* specifies the size of the data to write as a number of 8-bit bytes.

Returns: The *read* call returns a value of type *ssize_t*. This return value is the actual number of 8-bit bytes transferred. Note that the number of 16-bit samples read is half the return value.

The actual number of data bytes read will be equal to *NBytes* if the driver is operating in blocking mode; see [Section 6.2.4.1](#). In blocking mode, the Codec driver will not return from the *read* function until *NBytes* bytes are transferred from the driver's internal data structures to the user's buffer (*pBuffer*).

However, the number of data bytes written may be less than or equal to *NBytes* if operating in non-blocking mode. In non-blocking mode, the Codec driver will return immediately from the *read* function after transferring as much data as possible from the driver's internal data structures to the user buffer (*pBuffer*).

The *read* function may be called from normal application code, from an ISR, or from the user's callback function, which is called in turn from the Codec driver's ISR. However, avoid calling *read* in blocking mode from an ISR or from a callback function with more data than the Codec driver can immediately transfer (i.e., more than the *MaxNBytes* parameter of the callback function); otherwise, deadlock may occur if codec interrupts are disabled during the *read*.

Avoid using *read* in both normal application code and from an ISR or callback function. The asynchronous nature of the ISR and callback may lead to re-entrancy issues if you call *read* from normal application code as well.

Code Example: See [Code Example 6-4](#).

6.2.4.4 *close*

Call(s):

```
int close (types_tHandle FileDesc);
```

Arguments:

Table 6-10. Codec Driver Arguments - *close*

<i>FileDesc</i>	<i>in</i>	Codec device descriptor returned by <i>open</i> call

Description: The *close* call closes the Codec driver and releases the file descriptor handle.

Returns: The *close* call returns a type *int*; this *int* is always zero.

Code Example: See [Code Example 6-4](#).

6.2.4.5 *ioctl*

Call(s):

```
UWord16 ioctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams);
```

Arguments:

Table 6-11. Codec Driver Arguments - *ioctl*

<i>FileDesc</i>	<i>in</i>	Codec device descriptor returned by <i>open</i> call
<i>Cmd</i>	<i>in</i>	Commands found in <i>codec.h</i> which are used to manage the Codec driver
<i>pParams</i>	<i>in</i>	<i>ioctl</i> command parameter

Table 6-12. Codec *ioctl* Commands

Cmd	<i>pParams</i>	Return	Result
CODEC_DEVICE_ENABLE	<i>NULL</i>	<i>void</i>	Enables the Codec device and starts data transfer
CODEC_DEVICE_DISABLE	<i>NULL</i>	<i>void</i>	Disables the Codec device and stops data transfer
CODEC_DEVICE_SET_RX_LEFT_GAIN	Left Channel Input Gain in 1.5 dB Increments: 0000 = 0 dB gain 0001 = 1.5 dB gain 0010 = 3.0 dB gain (default) ... 1111 = 22.5 dB gain To calculate the RX gain value using a portable function, use the macro CODEC_RX_GAIN_FROM_PERCENT (Description follows)	<i>void</i>	Sets the left channel input gain

Table 6-12. Codec *ioctl* Commands (Continued)

Cmd	pParams	Return	Result
CODEC_DEVICE_SET_RX_RIGHT_GAIN	Right Channel Input Gain in 1.5 dB Increments: 0000 = 0 dB gain 0001 = 1.5 dB gain 0010 = 3.0 dB gain (default) ... 1111 = 22.5 dB gain To calculate the RX gain value using a portable function, use the macro CODEC_RX_GAIN_FROM_PERCENT See Macros	<i>void</i>	Sets the right channel input gain
CODEC_DEVICE_SET_TX_LEFT_GAIN	Left Channel Output Gain (integer) However, the Crystal CS4218 Stereo Codec supports only attenuation in 1.5 dB Increments: 00000 = 0 dB ... 11111 = 46.5 dB To calculate the TX gain value using a portable function, use the macro CODEC_TX_GAIN_FROM_PERCENT See Macros	<i>void</i>	Sets the left channel output gain (attenuation)

Table 6-12. Codec *ioctl* Commands (Continued)

Cmd	pParams	Return	Result
CODEC_DEVICE_SET_TX_RIGHT_GAIN	Right Channel Output Gain (integer) However, the Crystal CS4218 Stereo Codec supports only attenuation in 1.5 dB Increments: 00000 = 0 dB ... 11111 = 46.5 dB To calculate the TX gain value using a portable function, use the macro CODEC_TX_GAIN_FROM_PERCENT See Macros	<i>void</i>	Sets the right channel output gain (attenuation)
CODEC_DEVICE_MUTE	<i>bool</i>	<i>void</i>	Mutes ("True") or un-mutes ("False") the codec
CODEC_DEVICE_RESET	<i>codec_sParams</i>	<i>void</i>	Resets the codec, discarding all data currently in internal buffers of the driver
CODEC_DEVICE_SET_RX_CALLBACK_LEVEL	<i>unsigned int</i>	<i>void</i>	Dynamically sets the RX Callback Level (see CODEC_RX_CALLBACK_LEVEL); must be less than or equal to CODEC_RX_FIFO_SIZE
CODEC_DEVICE_SET_TX_CALLBACK_LEVEL	<i>unsigned int</i>	<i>void</i>	Dynamically sets the TX Callback Level (see CODEC_TX_CALLBACK_LEVEL); must be less than or equal to CODEC_TX_FIFO_SIZE

Macros: The macro CODEC_RX_GAIN_FROM_PERCENT(x), where $0 \leq x \leq 100$, can be used as a portable method to calculate a target-specific RX gain value from the percentage "x". A value of 0% returns the maximum amount of attenuation available from the codec receiver; a value of 100% returns the maximum amount of gain available from the codec receiver.

The macro CODEC_TX_GAIN_FROM_PERCENT(x), where $0 \leq x \leq 100$, can be used as a portable method to calculate a target-specific TX gain value from the percentage "x". A value of 0% returns the maximum amount of denotation available from the codec transmitter; a value of 100% returns the maximum amount of gain available from the codec transmitter.

Description: The *ioctl* call provides various control functions for the codec. See the list of commands in [Table 6-12](#) for specific control functions.

Returns: The *ioctl* call returns a type *UWord16*. This *UWord16* is dependent on the command passed in as an input parameter. [Table 6-12](#) lists these returns based on the specific command.

Code Example: [Code Example 6-4](#) reads and writes eight codec samples (16 bytes) at a time, using the IO Layer API to the Codec driver in a non-blocking mode:

Code Example 6-4. Codec Driver IO Layer API Example

```

/*****
#include "codec.h"

UWord16 pSamples[16];

/*****
void main(void)
{
    types_tHandle Codec;
    UWord16 Bytes;
    UWord16 LoopCount = 0;
    UWord16 Size      = 16;

    Codec = open(BSP_DEVICE_NAME_CODEC_0, O_RDWR | O_NONBLOCK);

    ioctl (Codec, CODEC_DEVICE_MUTE, false);

    ioctl (Codec, CODEC_DEVICE_SET_RX_LEFT_GAIN, CODEC_RX_GAIN_FROM_PERCENT(50));
    ioctl (Codec, CODEC_DEVICE_SET_RX_RIGHT_GAIN, CODEC_RX_GAIN_FROM_PERCENT(50));

    ioctl (Codec, CODEC_DEVICE_SET_TX_LEFT_GAIN, CODEC_TX_GAIN_FROM_PERCENT(100));
    ioctl (Codec, CODEC_DEVICE_SET_TX_RIGHT_GAIN, CODEC_TX_GAIN_FROM_PERCENT(100));

    /*
        The following Enable command actually starts codec data transfer
    */
    ioctl (Codec, CODEC_DEVICE_ENABLE, NULL);
    while(1)
    {
        Bytes = 0;

        do {
            Bytes += read(Codec, (void *) (pSamples + Bytes), Size - Bytes);
        } while (Bytes < Size);

        write(Codec, (void *) pSamples, Size);
    }

    close(Codec);
}

```

6.2.5 Low-Level Codec Driver API

The following pages specify the Application Programming Interface (API) for the low-level interface to the Codec driver.

To use the low-level interface to the Codec driver, you must define the `INCLUDE_CODEC` variable in the *appconfig.h* file of your SDK Embedded Project created in CodeWarrior:

```
#define INCLUDE_CODEC
```

The user's application code then uses the Codec driver API as follows:

- A *codecOpen* call is made to initialize the codec device
- *codecIoctl* calls are made to control the codec device
- *codecWrite* calls are made to write data to the codec (transmit)
- *codecRead* calls are made to read data from the codec (receive)
- After all codec operations are completed, the Codec driver is closed via a *codecClose* call

The following sections describe each function of the low-level API for the Codec driver.

In the following specifications, function arguments for each routine are described as *in*, *out*, or *inout*. An *in* argument means that the parameter value is an input only to the function. An *out* argument means that the parameter value is an output only from the function. 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.

6.2.5.1 *codecOpen*

Call(s):

```
types_tHandle codecOpen(const char *pName, int OFlags);
```

Arguments:

Table 6-13. Codec Driver Arguments - *codecOpen*

<i>pName</i>	<i>in</i>	The Codec device name; see the <i>bsp.h</i> file for target specific values, such as: BSP_DEVICE_NAME_CODEC_0
<i>OFlag</i>	<i>in</i>	Open Mode Flags Valid values are: O_BLOCK (default) O_NONBLOCK
<i>FileDesc</i>	<i>out</i>	Codec device descriptor returned by <i>codecOpen</i> call and used in subsequent <i>codecRead/codecWrite/codecClose/codecIoctl</i> function calls

Description: The *codecOpen* call opens the Codec driver for operations and returns a file descriptor used by subsequent *codecRead/codecWrite/codecClose/codecIoctl* function calls. The codec device must be opened before any calls are made to other Codec driver functions.

Return: The *codecOpen* call returns a type *types_tHandle*. When the *codecOpen* call is successful, this *types_tHandle* is a file descriptor (*FileDesc*) for the codec device and is used by the Codec driver's *codecRead*, *codecWrite*, *codecClose*, and *codecIoctl* functions. A "-1" is returned when the *codecOpen* call is not successful.

Code Example: See [Code Example 6-5](#).

6.2.5.2 *codecWrite*

Call(s):

```
ssize_t codecWrite(types_tHandle FileDesc, const void * pBuffer, size_t
NBytes);
```

Arguments:

Table 6-14. Codec Driver Arguments - *codecWrite*

<i>FileDesc</i>	<i>in</i>	Codec device descriptor returned by the <i>codecOpen</i> call
<i>pBuffer</i>	<i>in</i>	Pointer to user buffer of at least <i>NBytes</i> bytes
<i>NBytes</i>	<i>in</i>	Number of 8-bit bytes to write to the Codec; note that the number of 16-bit samples is <i>NBytes</i> /2

Description: The *codecWrite* call writes the data from the user buffer (*pBuffer*) to the Codec driver's internal data structures. User data is 16-bit signed integers representing linear PCM values. Note that the input parameter *NBytes* specifies the size of the data to write as a number of 8-bit bytes.

Returns: The *codecWrite* call returns a value of type *ssize_t*. This return value is the actual number of 8-bit bytes transferred. Note that the number of 16-bit samples written is half the return value.

The actual number of data bytes written will be equal to *NBytes* if the driver is operating in blocking mode; see [Section 6.2.5.1](#). In blocking mode, the Codec Driver will not return from the *codecWrite* function until *NBytes* bytes from the user's buffer are transferred to the driver's internal data structures.

However, the number of data bytes written may be less than or equal to *NBytes* if operating in *non-blocking* mode. In non-blocking mode, the Codec driver will return immediately from the *write* function after transferring as much of the user's data as possible into the driver's internal data structures.

The *codecWrite* function may be called from normal application code, an ISR, or from the user's callback function which is called in turn from the Codec Driver's ISR. However, avoid calling *codecWrite* in blocking mode from an ISR or from a callback function with more data than the Codec driver can immediately transfer (i.e., more than the *MaxNBytes* parameter of the callback function); otherwise, deadlock may occur if codec interrupts are disabled during the *codecWrite*.

Avoid using *codecWrite* in both normal application code and from an ISR or callback function. The asynchronous nature of the ISR and callback may lead to re-entrancy issues if you call *codecWrite* from normal application code as well.

Code Example: See [Code Example 6-5](#).

6.2.5.3 *codecRead*

Call(s):

```
ssize_t codecRead(types_tHandle FileDesc, void * pBuffer, size_t NBytes);
```

Arguments:

Table 6-15. Codec Driver Arguments - *codecRead*

<i>FileDesc</i>	<i>in</i>	Codec device descriptor returned by the <i>codecOpen</i> call
<i>pBuffer</i>	<i>inout</i>	Pointer to a user buffer of at least <i>NBytes</i> bytes
<i>NBytes</i>	<i>in</i>	Number of 8-bit bytes to read; note that the number of 16-bit samples is <i>NBytes/2</i>

Description: The *codecRead* call reads from the Codec driver's internal data structures into the user buffer (*pBuffer*). The data is 16-bit signed integers representing linear PCM values. Note that the input parameter *NBytes* specifies the size of the data to write as a number of 8-bit bytes.

Returns: The *codecRead* call returns a value of type *ssize_t*. This return value is the actual number of 8-bit bytes transferred. Note that the number of 16-bit samples read is half the return value.

The actual number of data bytes read will be equal to *NBytes* if the Driver is operating in blocking mode; see [Section 6.2.5.1](#). In blocking mode, the Codec driver will not return from the *codecRead* function until *NBytes* bytes are transferred from the driver's internal data structures to the user's buffer (*pBuffer*).

However, the number of data bytes written may be less than or equal to *NBytes* if operating in non-blocking mode. In non-blocking mode, the Codec driver will return immediately from the *codecRead* function after transferring as much data as possible from the driver's internal data structures to the user buffer (*pBuffer*).

The *codecRead* function may be called from normal application code, from an ISR, or from the user's callback function (which is called in turn from the Codec Driver's ISR). However, avoid calling *codecRead* in blocking mode from an ISR or from a callback function with more data than the Codec driver can immediately transfer (i.e., more than the *MaxNBytes* parameter of the callback function); otherwise, deadlock may occur if codec interrupts are disabled during the *codecRead*.

Avoid using *codecRead* in both normal application code and from an ISR or callback function. The asynchronous nature of the ISR and callback may lead to re-entrancy issues if you call *codecRead* from normal application code as well.

Code Example: See [Code Example 6-5](#).

6.2.5.4 *codecClose*

Call(s):

```
int codecClose (types_tHandle FileDesc);
```

Arguments:

Table 6-16. Codec Driver Arguments - *codecClose*

<i>FileDesc</i>	<i>in</i>	Codec device descriptor returned by <i>codecOpen</i> call

Description: The *codecClose* call closes the Codec driver and releases the file descriptor handle.

Returns: The *codecClose* call returns a type *int*. This *int* is always zero.

Code Example: See [Code Example 6-5](#).

6.2.5.5 *codecloctl***Call(s):**

```
UWord16 codecloctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams, const
char *pName);
```

Arguments:**Table 6-17. Codec Driver Arguments - *codecloctl***

<i>FileDesc</i>	<i>in</i>	Codec device descriptor returned by the <i>codecOpen</i> call
<i>Cmd</i>	<i>in</i>	Commands found in <i>codec.h</i> which are used to control the Codec driver and Codec device
<i>pParams</i>	<i>in</i>	<i>codecloctl</i> command parameter
<i>pName</i>	<i>in</i>	The Codec device name; see the <i>bsp.h</i> file for target specific values, such as: BSP_DEVICE_NAME_CODEC_0

Table 6-18. Codec *codecloctl* Commands

Cmd	<i>pParams</i>	Return	Result
CODEC_DEVICE_ENABLE	NULL	<i>void</i>	Enables the codec device and starts data transfer
CODEC_DEVICE_DISABLE	NULL	<i>void</i>	Disables the codec device and stops data transfer
CODEC_DEVICE_SET_RX_LEFT_GAIN	Left Channel Input Gain in 1.5 dB Increments: 0000 = 0 dB gain 0001 = 1.5 dB gain 0010 = 3.0 dB gain (default) ... 1111 = 22.5 dB gain To calculate the RX gain value using a portable function, use the macro CODEC_RX_GAIN_FROM_PERCENT See Macros	<i>void</i>	Sets the left channel input gain

Table 6-18. Codec *codeclctl* Commands (Continued)

Cmd	pParams	Return	Result
CODEC_DEVICE_SET_RX_RIGHT_GAIN	Right Channel Input Gain in 1.5 dB Increments: 0000 = 0 dB gain 0001 = 1.5 dB gain 0010 = 3.0 dB gain (default) ... 1111 = 22.5 dB gain To calculate the RX gain value using a portable function, use the macro CODEC_RX_GAIN_FROM_PERCENT See Macros	<i>void</i>	Sets the right channel input gain
CODEC_DEVICE_SET_TX_LEFT_GAIN	Left Channel Output Gain (integer) However, the Crystal CS4218 Stereo Codec supports only attenuation in 1.5 dB Increments: 00000 = 0 dB ... 11111 = 46.5 dB To calculate the TX gain value using a portable function, use the macro CODEC_TX_GAIN_FROM_PERCENT See Macros	<i>void</i>	Sets the left channel output gain (attenuation)
CODEC_DEVICE_SET_TX_RIGHT_GAIN	Right Channel Output Gain (integer) However, the Crystal CS4218 Stereo Codec supports only attenuation in 1.5 dB Increments: 00000 = 0 dB ... 11111 = 46.5 dB To calculate the TX gain value using a portable function, use the macro CODEC_TX_GAIN_FROM_PERCENT See Macros	<i>void</i>	Sets the right channel output gain (attenuation)

Table 6-18. Codec *codectl* Commands (Continued)

Cmd	<i>pParams</i>	Return	Result
CODEC_DEVICE_MUTE	<i>bool</i>	<i>void</i>	Mutes (“True”) or un-mutes (“False”) the codec
CODEC_DEVICE_RESET	<i>codec_sParams</i>	<i>void</i>	Resets the codec, discarding all data currently in internal buffers of the driver
CODEC_DEVICE_SET_RX_CALLBACK_LEVEL	<i>unsigned int</i>	<i>void</i>	Dynamically sets the RX Callback Level (see CODEC_RX_CALLBACK_LEVEL); must be less than or equal to CODEC_RX_FIFO_SIZE
CODEC_DEVICE_SET_TX_CALLBACK_LEVEL	<i>unsigned int</i>	<i>void</i>	Dynamically sets the TX Callback Level (see CODEC_TX_CALLBACK_LEVEL); must be less than or equal to CODEC_TX_FIFO_SIZE

Macros: The macro CODEC_RX_GAIN_FROM_PERCENT(x), where $0 \leq x \leq 100$, can be used as a portable method to calculate a target-specific RX gain value from the percentage “x”. A value of 0% returns the maximum amount of attenuation available from the codec receiver; a value of 100% returns the maximum amount of gain available from the codec receiver.

The macro CODEC_TX_GAIN_FROM_PERCENT(x), where $0 \leq x \leq 100$, can be used as a portable method to calculate a target-specific TX gain value from the percentage “x”. A value of 0% returns the maximum amount of attenuation available from the codec transmitter; a value of 100% returns the maximum amount of gain available from the codec transmitter.

Description: The *codectl* call provides various control functions for the codec. See the list of commands above for specific control functions.

Returns: The *codectl* call returns a type *UWord16*. This *UWord16* is dependent on the command passed in as an input parameter. [Table 6-12](#) lists these returns based on the specific command.

Code Example: [Code Example 6-5](#) reads and writes eight codec samples (16 bytes) at a time, using the low-level layer API to the Codec driver in a non-blocking mode.

Code Example 6-5. Codec Driver Low Level API Example

```

/*****
#include "codec.h"

UWord16 pSamples[16];

/*****
void main(void)
{
    types_tHandle Codec;
    UWord16 Bytes;
    UWord16 LoopCount = 0;
    UWord16 Size      = 16;

    Codec = codecOpen(BSP_DEVICE_NAME_CODEC_0, O_RDWR | O_NONBLOCK);

    codecIoctl (Codec, CODEC_DEVICE_MUTE, false, BSP_DEVICE_NAME_CODEC_0);

    codecIoctl (Codec, CODEC_DEVICE_SET_RX_LEFT_GAIN,
                CODEC_RX_GAIN_FROM_PERCENT(50), BSP_DEVICE_NAME_CODEC_0);
    codecIoctl (Codec, CODEC_DEVICE_SET_RX_RIGHT_GAIN,
                CODEC_RX_GAIN_FROM_PERCENT(50), BSP_DEVICE_NAME_CODEC_0);

    codecIoctl (Codec, CODEC_DEVICE_SET_TX_LEFT_GAIN,
                CODEC_TX_GAIN_FROM_PERCENT(100), BSP_DEVICE_NAME_CODEC_0);
    codecIoctl (Codec, CODEC_DEVICE_SET_TX_RIGHT_GAIN,
                CODEC_TX_GAIN_FROM_PERCENT(100), BSP_DEVICE_NAME_CODEC_0);

    /*
        The following Enable command actually starts codec data transfer
    */
    codecIoctl (Codec, CODEC_DEVICE_ENABLE, NULL, BSP_DEVICE_NAME_CODEC_0);

    while(1)
    {
        Bytes = 0;

        do {
            Bytes += codecRead(Codec, (void *) (pSamples + Bytes), Size - Bytes);
        } while (Bytes < Size);

        codecWrite(Codec, (void *) pSamples, Size);
    }

    codecClose(Codec);
}

```


6.2.6 Codec Driver Tests

Currently, there are no specific tests provided for the Codec driver. However, many applications and demos use the Codec driver. In particular, the Codec driver applications in the next section exercise most Codec driver capabilities.

6.2.7 Codec Driver Applications

6.2.7.1 Codec Loopback Application

This application performs a simple audio loopback using codec hardware and a DSP56852 processor. It reads a small number of samples from the codec input and returns them to the codec output.

This loopback application provides a good example of how to use the Codec driver interface functions.

6.2.7.1.1 Set-up for Codec Loopback Application

EVM Jumper Settings:

- Make the following changes to the default jumper settings for the DSP56852EVM board:
 - The three switches on the dipswitch (S4) control the sampling rate for the codec; you may set these switches to any position
 - When all switches are off (open), the codec samples at an 8KHz rate; when all switches are on (closed), the codec samples at a 48KHz rate
- The complete list of jumper settings are shown in the **DSP56852 Evaluation Module User's Manual**
- Connect the audio source to the Line-In jack on the EVM
- Connect the Audio Output to the Line-Out or Headphone jacks on the EVM
- The default configuration of the SSI Driver as defined in *config.h* must not be changed in *appconfig.h*

NOTES:

- The source for the audio input can be any audio source (PC audio out, CD player, etc.)
- The audio output is commonly connected to an external power speaker, but any other audio devices can be used to output sound (head phones, PC audio input, etc.)
- The Headphone jack has more gain than the Line-Out jack

6.2.7.1.2 Procedure for Codec Loopback Application

- Using CodeWarrior, open:
 - ...\\nos\\applications\\bsp\\codec\\codec.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- Once the debugger is running, apply an audio signal to the audio input of the hardware set-up; see [Section 6.2.7.1.1](#)
- If successful, you will hear audio output from the speaker, which should be the same as the input audio
 - You should also observe a blinking green LED, which indicates that the application is running

- You may also dynamically change the sampling rate for the codec via the dipswitches (S4); note the difference in sound quality between an 8KHz sampling rate and a 48KHz sampling rate

6.3 FILE I/O Driver

6.3.1 Introduction

The File I/O driver allows a DSP application to send data from the target platform to a specified file located on a Personal Computer (PC). It also allows a DSP application to receive data from a specified file located on a PC into the target platform. This is accomplished with a Windows program on a PC running simultaneously with the File I/O driver application on the target platform. It should be noted that the Windows executable must be launched prior to the DSP File I/O driver application. The Windows executable can be found in:

..\Embedded SDK\src\x86\win32\applications\fileio\fileio.exe

The target platform and PC communicate with RS-232 protocol through the serial port, at 56000 baud, on the respective platforms. As a result of this driver's use of the SCI port, the File I/O driver prohibits the use of this port by a user's application for any other operation. **Figure 6-1** contains a basic diagram showing the set-up required to utilize the File I/O driver in an application.

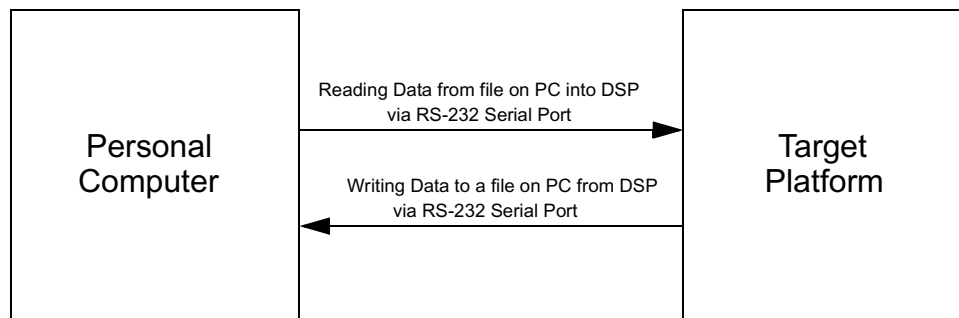


Figure 6-1. Basic Set-up for File I/O Utilization

NOTE: A serial communication port must be set up on a personal computer with the following settings:

Baud Rate	56000
Data Bits	8
Parity	None
Stop Bit	1
Flow Control	None

Demos and user applications will sometimes lock up due to a handshaking problem with the PC or the PC Application. To alleviate this problem, a user can minimize the number of resources the PC is using when running a demo or user application, including the docking station. A user can also try using a faster PC or lowering the baud rate of the File I/O PC application. If the demo or user application does lock up, kill the project, then try running it again.

6.3.2 Initialization

The SDK file *appconfig.h* provides the mechanism to statically configure the File I/O driver. The *appconfig.h* file allows the programmer to set C preprocessor variables which configure the File I/O driver default operation. Typically, these C preprocessor variables conditionally compile and link information into the application for use by the File I/O driver during initial operation.

To include the File I/O Driver in the application, insert the following line into the *appconfig.h* file:

```
#define INCLUDE_FILEIO
```

Defining the preprocessor variable, `INCLUDE_FILEIO`, directs the SDK to include the File I/O driver in the executable image. If the preprocessor variable `INCLUDE_FILEIO` is not defined, then the File I/O driver will not be included in the executable.

You may also define additional preprocessor variables in the *appconfig.h* file to statically configure and initialize the File I/O driver. The list of these variables is provided in [Table 6-19](#).

Table 6-19. File I/O Configuration Items for *appconfig.h*

File IO #define	Explanation and Default Value
FILEIO_BAUD_RATE	Sets the Baud Rate for the SCI port: 56000 (default)

[Code Example 6-6](#) configures and initializes the File I/O driver.

Code Example 6-6. Sample *appconfig.h* code

```
#define INCLUDE_FILEIO /* include File I/O driver (Required) */

#define FILEIO_BAUD_RATE 9600 /* Changes SCI rate to 9600 bps */
```

6.3.3 API Definition

This section defines the Application Programming Interface (API) by identifying all public interface functions and data structures.



Code to create and initialize the File I/O driver may automatically be included in an SDK Embedded Project by inserting the following line in the *appconfig.h* file:

```
#define INCLUDE_FILEIO
```

The following information may be found in the public header file *fileio.h*:

Public Interface Function(s):

```

/*****
 *      types_tHandle open(const char *pName, int OFlags);
 *      ssize_t write(types_tHandle FileDesc, const void * pBuffer, size_t NBytes);
 *      ssize_t read(types_tHandle FileDesc, void * pBuffer, size_t NBytes);
 *      int close(types_tHandle FileDesc);
 *****/

```

Public Data Structure(s):

None

Members:

None



6.3.4 API Specifications

The following pages specify the Application Programming Interface (API).

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

6.3.4.1 *open*

Call(s):

```
types_tHandle open(const char *pName, int OFlags);
```

Arguments:
Table 6-20. File I/O Driver Arguments - *open*

<i>pName</i>	<i>in</i>	The File I/O driver name; use: \\\\PC\\specificdrive\\yourfile.txt or \\\\PC\\Embedded SDK\\yourfile.txt
<i>OFlag</i>	<i>in</i>	Use O_RDONLY for a read operation Use O_WRONLY for a write operation
<i>FileDesc</i>	<i>out</i>	File I/O device descriptor returned by <i>open</i> call and used in <i>writelread</i> function calls

Description: The *open* call opens the File I/O for operations and returns a file descriptor used by *read/write* function calls. When \\\\PC\\Embedded SDK\\yourfile.txt is used as the *pName* parameter, it looks to the registry to get a relative path into the SDK. The registry path is found in the registry at:

HKEY_LOCAL_MACHINE\\SOFTWARE\\MOTOROLA\\Embedded SDK

The File I/O driver must be opened for *read/write* calls. A windows executable must also be launched before any *write/read* operations take place. The windows application can be found at:

Embedded SDK\\src\\x86\\win32\\applications\\fileio\\fileio.exe

The File I/O device driver opens the SCI0 device in order to communicate to the PC.

Returns: The *open* call returns a type *types_tHandle*. When the *open* call is successful, this *types_tHandle* is a file descriptor for the File I/O driver and is used by the File I/O driver's *read*, *write*, *close*, and *ioctl* functions. A "-1" is returned when the *open* call is not successful.

Code Example: See [Code Example 6-7](#).



6.3.4.2 write

Call(s):

```
ssize_t write(types_tHandle FileDesc, const void * pBuffer, size_t NBytes);
```

Arguments:

Table 6-21. File I/O Driver Arguments - write

FileDesc	in	File I/O device descriptor returned by open call and used in writelread function calls
pBuffer	in	Pointer to user buffer. The write function type casts pBuffer to be a (char *)
Size	in	Number of bytes of data to be written

Description: The write call writes the user buffer out of the SCI device to a user-specified file.

Returns: The write call returns a type ssize_t. This ssize_t is the number of transferred bytes.

Code Example: See Code Example 6-7.



6.3.4.3 read

Call(s):

```
ssize_t read(types_tHandle FileDesc, void * pBuffer, size_t Size);
```

Arguments:

Table 6-22. File I/O Driver Arguments - read

FileDesc	in	File I/O device descriptor returned by open call and used in writelread function calls
pBuffer	in	Pointer to user buffer. The read function type casts pBuffer to be a (char *)
Size	in	The number of bytes of data read

Description: The read call reads data from a specified file into a user buffer.

Returns: The read call returns a type ssize_t. This ssize_t is the number of received bytes.

Code Example: None



6.3.4.4 close

Call(s):

```
int close (types_tHandle FileDesc);
```

Arguments:

Table 6-23. File I/O Driver Arguments - close

FileDesc	in	File I/O device descriptor returned by open call and used in writelread function calls

Description: The *close* call closes the File I/O driver and releases the file descriptor handle. The File I/O device driver closes the SCI0 device if there are no longer any files being accessed.

Returns: The *close* call returns a type *int*. This *int* is always zero.

Code Example: See [Code Example 6-7](#).

6.3.4.5 ioctl

Currently, there are no File I/O *ioctl* Commands.

Code Example 6-7. File I/O Driver Usage

```

/*****
#include "port.h"
#include "fcntl.h"
#include "fileio.h"
#include "mem.h"

#define BUFFER_SIZE 10

void main(void)
{
    types_tHandle Fd;
    int I;

    char * pBuffer = malloc(BUFFER_SIZE * sizeof(char));
    char * pTemp = pBuffer;

    for(I = 0; I < BUFFER_SIZE; I++)
    {
        *pTemp = 0x30;
        pTemp += 1;
    }

    /* opens file on a PC in which to write data buffer to, gets relative path from
    registry */
    Fd = open("\\\\PC\\Embedded SDK\\test.txt", O_WRONLY);

    /* write buffer to test.txt */
    write(Fd, pBuffer, 10 * sizeof(char));

    close(Fd);
}
*****/

```

6.3.5 FILE I/O Driver Application

There are no dedicated applications for testing File I/O; however, all of the telephony, security, and modem test applications utilize the File I/O driver.

6.4 Keypad Driver

6.4.1 Introduction

The TDC1 Keypad is an off-chip, memory-mapped device. This device is selected in hardware by the Programmable Chip Select 3 pin of the DSP5685x EVMs. The Keypad driver provides keypress detection and debouncing. The TDC1 card has additional glue logic in an FPGA that provides the interface between the EVM and Keypad. The FPGA contains a read/write register used to scan the keypad for pressed keys. This is the same FPGA used for the TDC1-LC, so they share the same chip select. The FPGA design allows these devices to be used together or independently from each other. See the "TDC1 Hardware Design Document" found under Evaluation Boards for more design-specific details.

All Keypad driver API calls are re-entrant, as long as the parameters being passed in are not referring to the same global data.

Before using the Keypad device, the application must gain access to the device via an *open* call and save the device descriptor returned. The device may be opened in either Blocking or Non-Blocking mode. In Blocking mode, API calls return control to the application when the operation has completed. In Non-Blocking mode, the API calls perform processing in the background as much as possible, thereby returning processing time to the application.

In Blocking mode, typical driver usage would be to *open* the driver with the default configuration, detect key presses via the *keypadIoctl*(KEYPAD_GET_KEY) command, and detect key releases via the Key Release Callback function.

In Non-Blocking mode, typical driver usage would be to *open* the driver with the O_NONBLOCK configuration, detect key presses via the Key Press Callback function, and detect key releases via the Key Release Callback function.

The application may use an *ioctl* call to configure the driver and request status. A list of valid commands follows.

After completing all Keypad operations, a call to the *close* function will release the handle to the device and shut down.

Two APIs are provided for this driver: device-independent and device-dependent. The following sections explain the APIs in greater detail, as well as the advantages and considerations that should be made when selecting an API.

6.4.2 Initialization

The Keypad driver is statically configured using the *config.h* and *appconfig.h* files. The *config.h* file contains the default configuration for all drivers.

The driver is created and initialized by adding the following line(s) to the *appconfig.h* file associated with the SDK Embedded Project created in CodeWarrior:

```
For device-dependent API:
#define INCLUDE_KEYPAD
```

For device-independent API:

```
#define INCLUDE_IO
#define INCLUDE_KEYPAD
```

The defined constants used to statically configure the TDC1 keypad are shown below. The parameters defined in *config.h* are designated below by (default). The *config.h* file should **never** be modified. Instead, you can override these defaults in the *appconfig.h* file.

```
#define KEYPAD_BASE_ADDR          0x100000 (default)
#define KEYPAD_TIMER              KEYPAD_TIMER_A_1 (default)
                                   KEYPAD_TIMER_A_2
                                   KEYPAD_TIMER_A_3
                                   KEYPAD_TIMER_A_0
#define KEYPAD_KEY_PRESSED_CALLBACK NULL (default)
#define KEYPAD_KEY_PRESSED_CALLBACK_ARG NULL (default)
#define KEYPAD_KEY_RELEASED_CALLBACK NULL (default)
#define KEYPAD_KEY_RELEASED_CALLBACK_ARG NULL (default)
```

The *KEYPAD_BASE_ADDR* is used to define the memory-mapped address for the TDC1 keypad device. This device is attached to Programmable Chip Select 3. If you modify the default base address, take care not to specify a Base Address that overlaps a memory range defined for PCS0, PCS1, or PCS2. The Base Address **must also** be a multiple of the Block Size. The default base address of 0x100000 was selected for PCS3 since it does NOT overlap the default memory ranges of the PCS0, PCS1, or PCS2. Since the keypad and LCD share the same FPGA, their base addresses MUST be identical. The driver automatically enforces this constraint.

The *KEYPAD_TIMER* is used to debounce key presses and key releases as well as to periodically scan the keypad in Non-Blocking mode. *KEYPAD_TIMER_A_1* is the default timer defined for use by the driver. The user may override this default by selecting one of the other three timers: *KEYPAD_TIMER_A_0*, *KEYPAD_TIMER_A_2*, or *KEYPAD_TIMER_A_3*.

The *KEYPAD_KEY_PRESSED_CALLBACK* is used to define the user's function that will be called by the keypad driver when any key press is detected. The user function must be of type *keypad_tCallback*. A definition of this type follows.

The *KEYPAD_KEY_PRESSED_CALLBACK_ARG* is used to define an argument of type (*void **) that is a pointer to user-defined parameters which will be passed to the Key Pressed Callback function by the keypad driver.

The *KEYPAD_KEY_RELEASED_CALLBACK* is used to define the user's function that will be called by the keypad driver when a key release is detected. The user function must be of type *keypad_tCallback*. A definition of this type follows.

The *KEYPAD_KEY_RELEASED_CALLBACK_ARG* is used to define an argument of type (*void **) that is a pointer to user-defined parameters which will be passed to the Key Released Callback function by the keypad driver.

6.4.3 API Definition

The TDC1 keypad interface can be used at two alternative levels: a device-independent interface and a device-dependent interface.

The device-independent API follows the POSIX standard and provides portability at the expense of efficiency. The device-dependent API provides a non-standard interface that is more efficient than the device-independent, but is less portable. Your application may use either interface, depending on your specific goals for efficiency and portability. The application may not mix device-dependent and device-independent API calls, which are mutually exclusive.

The following information may be found in the public header file *keypad.h*:

Public Interface Function(s):

The device-independent API function prototypes:

```

/*****
 * types_tHandle open (const char *pName, int OFlags, ...);
 * UWord16      ioctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams);
 * int          close(types_tHandle FileDesc);
 *****/

```

The device-dependent API function prototypes:

```

/*****
 * types_tHandle keypadOpen (const char *pName, int OFlags);
 * UWord16      keypadIoctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams,
 *                          const char * pName);
 * int          keypadClose(types_tHandle FileDesc);
 *****/

```

Public Data Types and Structures:

```

/* Keypad timer select options */
#define      KEYPAD_TIMER_A_0 0
#define      KEYPAD_TIMER_A_1 1
#define      KEYPAD_TIMER_A_2 2
#define      KEYPAD_TIMER_A_3 3

/* The enumerate key names match the silk screened key identifiers
   on the TDC1 printed wiring board */
typedef enum
{
    KEYPAD_NO_KEY,
    KEYPAD_S2,
    KEYPAD_S3,
    KEYPAD_S4,
    KEYPAD_S5,
    KEYPAD_S6,
    KEYPAD_S7,
    KEYPAD_S8,
    KEYPAD_S9,
    KEYPAD_S10,
    KEYPAD_S11,
    KEYPAD_S12,
    KEYPAD_S13,
    KEYPAD_S14,
    KEYPAD_S15,
    KEYPAD_S16,
    KEYPAD_S17,
}

```



```
KEYPAD_S18,  
KEYPAD_S19,  
KEYPAD_S20,  
KEYPAD_S21,  
KEYPAD_S22,  
KEYPAD_S23,  
KEYPAD_S24,  
KEYPAD_S25,  
KEYPAD_S26  
} keypad_eKey;  
  
/* This Function Callback type is a function pointer with two arguments.  
void * pUserParam - allows the user to define a pointer for passing  
                    user defined parameters to the callback function.  
keypad_eKey key - allows the driver to return the key detected to  
                  the callback function. */  
typedef void (*keypad_tCallback)(void * pUserParam, keypad_eKey key);
```



6.4.4 Keypad Device-Independent API Specification

The following pages specify the device-independent Application Programming Interface (API).

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

The *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.

6.4.4.1 *open*

Call(s):

```
types_tHandle open(const char *pName, int OFlags, ...);
```

Arguments:

Table 6-24. KEYPAD Driver Arguments - *open*

<i>pName</i>	<i>in</i>	The TDC1 keypad device name: BSP_DEVICE_NAME_KEYPAD
<i>OFlag</i>	<i>in</i>	Open Mode Flags: O_RDONLY - read allowed (default) O_BLOCK - blocking mode (default) O_NONBLOCK - non-blocking mode
<i>FileDesc</i>	<i>out</i>	Keypad device descriptor of type (<i>int</i>) returned by the <i>open</i> call and used in <i>close/ioctl</i> function calls

Description: The *open* function is mapped to the *keypadOpen* function through the I/O interface Virtual Table (*keypadIOInterfaceVT*) at initialization. See *keypadOpen*, [Section 6.4.5.1](#).

Returns: See *keypadkeypadOpen*, [Section 6.4.5.1](#).

Code Example: [Code Example 6-8](#) opens the keypad for communication using default configuration settings.

Code Example 6-8. Use of *open*

```
File appconfig.h:
#define INCLUDE_BSP
#define INCLUDE_IO
#define INCLUDE_KEYPAD

User application:
#include "bsp.h"
#include "keypad.h"

main()
{
    types_tHandle keypadFD;

    keypadFD = open(BSP_DEVICE_NAME_KEYPAD, NULL);
}
```


6.4.4.2 *ioctl*

Call(s):

```
UWord16 ioctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams);
```

Arguments:

Table 6-25. Keypad Driver Arguments - *ioctl*

<i>FileDesc</i>	<i>in</i>	Keypad device descriptor returned by <i>open</i> call
<i>Cmd</i>	<i>in</i>	Command for driver; see Table 6-29
<i>pParams</i>	<i>in</i>	Parameters specific to each command; see Table 6-29

Description: The Keypad driver supports the same commands described for *keypadIoctl*; see *keypadIoctl*, [Section 6.4.5.2](#).

Return: See *keypadIoctl*, [Section 6.4.5.3](#).

Code Example: [Code Example 6-9](#) opens the driver in Blocking mode (default) and continually scans for the KEYPAD_S26 key to be pressed.

Code Example 6-9. Use of *ioctl*

```
File appconfig.h:
#define INCLUDE_BSP
#define INCLUDE_IO
#define INCLUDE_KEYPAD

User application:
#include "bsp.h"
#include "keypad.h"

main()
{
    types_tHandle keypadFD;
    UWord16 keypress = NULL;

    keypadFD = open(BSP_DEVICE_NAME_KEYPAD, NULL);

    while(keypress != KEYPAD_S26)
    {
        // Get keypress
        keypress = ioctl(KeypadFD, KEYPAD_GET_KEY, NULL);
    }
}
```

6.4.4.3 *close*

Call(s):

```
int close (types_tHandle FileDesc);
```

Arguments:

Table 6-26. Keypad Driver Arguments - *close*

<i>FileDesc</i>	<i>in</i>	Keypad device descriptor returned by <i>open</i> call

Description: The *close* function is mapped to the *keypadClose* function through the I/O interface Virtual Table (*keypadIOInterfaceVT*); see *keypadClose*, [Section 6.4.5.3](#).

Returns: This function returns “0” if successful and “-1” if unsuccessful.

Code Example: [Code Example 6-10](#) opens the driver in Blocking mode (default) and continually scans for the KEYPAD_S26 key to be pressed. Once found, it closes the driver.

Code Example 6-10. Use of *close*

File appconfig.h:

```
#define INCLUDE_BSP
#define INCLUDE_IO
#define INCLUDE_KEYPAD
```

User application:

```
#include "bsp.h"
#include "keypad.h"

main()
{
    types_tHandle keypadFD;
    UWord16 keypress = NULL;

    keypadFD = open(BSP_DEVICE_NAME_KEYPAD, NULL);

    while(keypress != KEYPAD_S26)
    {
        // Get keypress
        keypress = ioctl(KeypadFD, KEYPAD_GET_KEY, NULL);
    }

    close(keypadFD);
}
```



6.4.5 Keypad Device-Dependent API Specification

The following pages specify the device-dependent Application Programming Interface (API).

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

The *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.

6.4.5.1 keypadOpen

Call(s):

```
types_tHandle keypadOpen(const char *pName, int OFlags);
```

Arguments:

Table 6-27. Keypad Driver Arguments - keypadOpen

<i>pName</i>	<i>in</i>	The TDC1 keypad device name: BSP_DEVICE_NAME_KEYPAD
<i>OFlags</i>	<i>in</i>	Open Mode Flags: O_RDONLY - read allowed (default) O_BLOCK - Blocking mode (default) O_NONBLOCK - Non-Blocking mode
<i>FileDesc</i>	<i>out</i>	Keypad device descriptor of type (<i>int</i>) returned by the <i>keypadOpen</i> call and used in <i>keypadClose/keypadIoctl</i> function calls

Description: This function initializes the keypad peripheral for operation and establishes the handle for continued communication. It configures the driver for either Blocking or Non-Blocking mode, as defined in the *OFlag* parameter.

In Blocking mode, key press detection is returned via an *ioctl* command; key release detection may be enabled through the definition of the *KEYPAD_KEY_RELEASED_CALLBACK* configuration parameter.

In Non-Blocking mode, the keypad timer is enabled to periodically scan the keypad. In this mode, the user **MUST** define callback functions to be notified of key press and/or release detection.

Returns: The *keypadOpen* call returns a type *types_tHandle*. When the *keypadOpen* call is successful, this *types_tHandle* is a file descriptor for the TDC1 keypad device. A "-1" is returned when the *keypadOpen* call is not successful.

Code Example: [Code Example 6-11](#) opens the TDC1 keypad for Non-Blocking communication.

Code Example 6-11. Use of keypadOpen

File appconfig.h:

```
#define INCLUDE_BSP
#define INCLUDE_KEYPAD
```

User application:

```
#include "bsp.h"
#include "keypad.h"

main()
{
    types_tHandle keypadFD;

    keypadFD = keypadOpen(BSP_DEVICE_NAME_KEYPAD, O_NONBLOCK);
}
```

6.4.5.2 keypadIoctl

Call(s):

```
UWord16 keypadIoctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams,
                    const char * pName);
```

Arguments:

Table 6-28. TDC1 Keypad Driver Arguments - keypadIoctl

<i>FileDesc</i>	<i>in</i>	TDC1 keypad device descriptor returned by <i>keypadOpen</i> call
<i>Cmd</i>	<i>in</i>	Command for driver; see Table 6-29
<i>pParams</i>	<i>in</i>	Parameters specific to each command; see Table 6-29
<i>pName</i>	<i>in</i>	BSP device name from <i>bsp.h</i> : BSP_DEVICE_NAME_KEYPAD

Description: The TDC1 keypad driver supports many commands for dynamic configuration of the peripheral; see the list of commands described in [Table 6-29](#).

Returns: Command specific; see [Table 6-29](#).

Table 6-29. keypadIoctl Commands

<i>Cmd</i>	<i>pParams</i>	<i>Return</i>	<i>Result</i>
KEYPAD_SET_KEY_PRESS_CALLBACK	void (*keypad_tCallback) (void *, keypad_eKey)	None	This command installs the user-defined Callback function called by the driver when a key press has been detected. If <i>pParams</i> is defined as NULL, then this command disables the callback function.
KEYPAD_SET_KEY_PRESS_CALLBACK_ARG	void * <i>pUserParams</i>	None	This command is used to define an argument of type (void *) that is a pointer to user-defined parameters which will be passed to the key press callback function by the keypad driver.

Table 6-29. keypadIoctl Commands (Continued)

<i>Cmd</i>	<i>pParams</i>	<i>Return</i>	<i>Result</i>
KEYPAD_SET_KEY_RELEASE_CALLBACK	void (*keypad_tCallback) (void *, keypad_eKey)	None	This command installs the user-defined Callback function called by the driver when a key release has been detected. If <i>pParams</i> is defined as NULL, then this command disables the callback function.
KEYPAD_SET_KEY_RELEASE_CALLBACK_ARG	void * <i>pUserParams</i>	None	This command is used to define an argument of type (void *) that is a pointer to user-defined parameters which will be passed to the key release callback function by the keypad driver.
KEYPAD_GET_KEY	NULL	<i>keypad_eKey</i>	This command is only valid when the driver has been opened in Blocking mode. It will scan the keypad until a key press has been detected, when it will return the enumerated value defined for the key detected. If this function is called when the driver has been opened in Non-Blocking mode, it will return KEYPAD_INVALID and if in Debug mode, it will force an assertion and halt processing.

Code Example: [Code Example 6-12](#) opens the driver in Non-Blocking mode and sets the key press Callback function and argument. When a key press is detected, the key is stored in the Callback argument.

Code Example 6-12. Use of keypadIoctl

File appconfig.h:

```
#define INCLUDE_BSP
#define INCLUDE_KEYPAD
```

User application:

```
#include "port.h"
#include "bsp.h"
#include "keypad.h"

/* local Types and prototypes */
typedef struct
{
    keypad_eKey current;
    keypad_eKey prev;
}main_sKeyStatus;

/* Callback function for key 'press' detection */
void key_press_CB(void * pParams, keypad_eKey keypress)
{
    ((main_sKeyStatus *)pParams)->current = keypress;
}
main()
{
    types_tHandle keypadFD;
    main_sKeyStatus keyStatus = {KEYPAD_NO_KEY, KEYPAD_NO_KEY};

    keypadFD = keypadOpen(BSP_DEVICE_NAME_KEYPAD, 0_NONBLOCK);

    keypadIoctl(keypadFD, KEYPAD_SET_KEY_PRESS_CALLBACK, key_press_CB, \
        BSP_DEVICE_NAME_KEYPAD);
    keypadIoctl(keypadFD, KEYPAD_SET_KEY_PRESS_CALLBACK_ARG, &keyStatus, \
        BSP_DEVICE_NAME_KEYPAD);
}
```

6.4.5.3 keypadClose

Call(s):

```
int keypadClose (types_tHandle FileDesc);
```

Arguments:

Table 6-30. TDC1 Keypad Driver Arguments - keypadClose

<i>FileDesc</i>	<i>in</i>	Keypad device descriptor returned by <i>keypadOpen</i> call

Description: The *keypadClose* closes the KEYPAD device. It will immediately disable the timer for scanning and debouncing the keypad.

Returns: This function returns “0” if successful and “-1” if unsuccessful.

Code Example: **Code Example 6-13** opens the driver in Non-Blocking mode and sets the key press Callback function and argument. When a key press is detected, the key is stored in the Callback argument. The main function waits until the KEYPAD_S26 key has been detected and then closes the driver.

Code Example 6-13. use of keypadClose

File appconfig.h:

```
#define INCLUDE_BSP
#define INCLUDE_KEYPAD
```

User application:

```
#include "port.h"
#include "bsp.h"
#include "keypad.h"

/* local Types and prototypes */
typedef struct
{
    keypad_eKey current;
    keypad_eKey prev;
}main_sKeyStatus;

/* Callback function for key 'press' detection */
void key_press_CB(void * pParams, keypad_eKey keypress)
{
    ((main_sKeyStatus *)pParams)->current = keypress;
}

main()
{
    types_tHandle keypadFD;
    main_sKeyStatus keyStatus = {KEYPAD_NO_KEY, KEYPAD_NO_KEY};

    keypadFD = keypadOpen(BSP_DEVICE_NAME_KEYPAD, 0_NONBLOCK);

    keypadIoctl(keypadFD, KEYPAD_SET_KEY_PRESS_CALLBACK, key_press_CB,\
        BSP_DEVICE_NAME_KEYPAD);
    keypadIoctl(keypadFD, KEYPAD_SET_KEY_PRESS_CALLBACK_ARG, &keyStatus,\
        BSP_DEVICE_NAME_KEYPAD);
```




ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

```
while(keyStatus.current != KEYPAD_S26)
{
    if(keyStatus.prev != keyStatus.current)
    {
        keyStatus.prev = keyStatus.current;
    }
}

keypadClose(keypadFD);
}
```

ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

6.4.6 Demo Applications

The TDC1 Keypad Driver is used in the Feature Phone Demo Application, located in the following directory:

.../sdk/src/dsp56852evm/nos/applications/telephony/featurephone

6.4.7 Test Applications

The TDC1 Keypad Driver Test application verifies the correct operation of all driver APIs. This application is located in the following directory:

.../sdk/src/dsp56852evm/nos/bsp/test/tdc1-keypad/keypad.mcp

The Internal RAM target tests the device-independent interface and the External RAM target tests the device-dependent interface. Both tests verify correct driver operation in Blocking and Non-Blocking modes.

This is a manual test that requires a person to press keys on the TDC1 keypad. The first part of this test verifies the Blocking mode of operation, while the second part verifies the Non-Blocking mode. In both modes, the keypad buttons should be pressed in succession starting from S2 through S26 as designated on the TDC1 silk screened board. Each time a key is pressed, its value will be displayed in binary format on the EVM's LEDs (S2=0x01, S26=0x19) and in ASCII format on the TDC1 LCD. If executed on the DSP56858 EVM, the left-most Red LED is turned on when a key is pressed and turned off when it is released. This LED is not functional on the DSP56852 EVM.

Key releases are signaled by the driver through the *key_release_CB* Callback function, which is statically configured in *appconfig.h*.

In Blocking mode, key presses are detected via the *keypadIoctl*(KEYPAD_GET_KEY) command. The same key should not be pressed twice in a row, since this action signals a problem with the debounce timing of the driver. When S26 is detected, the driver is closed and reopened in Non-Blocking mode.

In Non-Blocking mode, key press and release detection is signaled by the driver through user-defined Callback functions. The application dynamically installs the *key_press_CB* Callback and its associated argument. The Key Released Callback is already statically installed. Key presses are captured in the *key_press_CB* Callback Argument and displayed in the application's *main()* function. When S26 is detected, the driver is closed and the test is complete.

At the completion of the test, the TDC1 LCD should display the following:

```
123456789;.<=>?@ABCD
123456789;.<=>?@ABCD
EFGHI
EFGHI
```

6.4.7.1 Hardware Set-up

This test requires a TDC1 attached to the DSP56852EVM via the daughter card connector. Production EVMs supply 5V and GND through the daughter card connector. Set jumper JG5 on the TDC1 card.

NOTE: Use default settings shown in the **DSP56852 Evaluation Module User's Manual**.

6.4.7.2 Procedure for TDC1 Keypad Test

- Using CodeWarrior, open:
...\\nos\\bsp\\test\\tdc1-keypad\\keypad.mcp
- Build and download the External RAM target; when the debugger reaches *main()*, it will stop
- To continue executing the test, select *Run* in the debugger
- Press the keypad buttons in succession starting at S2 and ending at S26, as marked on the TDC1 hardware
 - Verify that the LEDs display the binary equivalent of the button pressed (S2=0x01 and S26=0x19) and the LCD displays the ASCII equivalent
- When S26 is pressed, the test will switch to Non-Blocking mode
 - Repeat the button presses from S2 through S26 and verify the LEDs and TDC1 LCD display the equivalent of the button pressed
- When S26 is reached, the test will halt and the TDC1 LCD should display the following:


```
123456789;:<=>?@ABCD
123456789;:<=>?@ABCD
EFGHI
EFGHI
```
- If the test should halt at any point before delivering the preceding message, it is considered a failure, probably due to incorrect debounce timing of the keypad
- Repeat this procedure using the Internal RAM target

6.5 LCD Driver

6.5.1 Introduction

The Crystalfontz CFAH2004A-NYA-JP LCD is an off-chip, memory-mapped device that is included as part of the Telephony Daughter Card (TDC1). This device is selected in hardware by the Programmable Chip Select 3 (PCS3) pin of the DSP5685x EVMs. The LCD driver provides the software interface that allows the application to configure and write text messages to the device. The TDC1 card has additional glue logic in an FPGA that provides the interface between the EVM and LCD. See the "TDC1 Hardware Design Document" found under Evaluation Boards for more design specific details.

The LCD driver is not re-entrant. The driver is designed for single access since there is only a single display buffer. Additional functionality would typically be required at the application layer

to provide buffering of display messages and human interface features, such as delaying between displays to give the user time to read the display.

Before using the LCD device, the application must gain access to the device via an *open* call and save the device descriptor returned. The LCD may be opened in either Blocking or Non-Blocking mode. In Blocking mode, API calls return control to the application when the operation has completed. In Non-Blocking mode, the API calls perform processing in the background as much as possible, thereby returning processing time to the application.

The application should use a *write* call to send text messages to the LCD.

The application should use an *ioctl* call to dynamically change the LCD device modes. A list of valid commands follows.

After completing all LCD operations, a call to the *close* function will disable device operation and release its handle.

Two APIs are provided for this driver: device-independent and device-dependent. The following sections explain the APIs in greater detail, as well as the advantages and considerations that should be made when selecting an API.

6.5.2 Initialization

The LCD Driver is statically configured using the *config.h* and *appconfig.h* files. The *config.h* file contains the default configuration for all drivers.

The driver is created and initialized by adding the following line(s) to the *appconfig.h* file associated with the SDK Embedded Project created in CodeWarrior:

```
For device-dependent API:
#define INCLUDE_LCD
```

```
For device-independent API:
#define INCLUDE_IO
#define INCLUDE_LCD
```

The defined constants used to statically configure the LCD are shown below. The parameters defined in *config.h* are designated below by (default). The *config.h* file should **never** be modified. Instead, these defaults may be overridden by the user in the *appconfig.h* file.

```
#define LCD_BASE_ADDR          0x100000 (default)
#define LCD_NONBLOCK_TIMER     LCD_TIMER_A_0 (default)
```

The LCD is a memory mapped device attached to Programmable Chip Select 3 (PCS3). Therefore, a valid base address for PCS3 must be defined to enable the device. If the default base address is overridden in *appconfig.h*, take care not to specify a base address that overlaps a memory range already defined for PCS0, PCS1, or PCS2. The base address **MUST ALSO** be a multiple of the Block Size. The default base address of 0x100000 was selected for PCS3, since it does NOT overlap the default memory ranges of the PCS0, PCS1, or PCS2.

By default, LCD_TIMER_A_0 is defined for use by the driver to support Non-Blocking mode. This timer is used to periodically send display characters. The user may override this default by selecting one of the other three timers: LCD_TIMER_A_1, LCD_TIMER_A_2, or LCD_TIMER_A_3.

6.5.3 API Definition

The LCD interface can be used at two alternative levels: a device-independent interface and a device-dependent interface.

The device-independent API follows the POSIX standard and provides portability at the expense of efficiency. The device-dependent API provides a non-standard interface that is more efficient than the device-independent, but is less portable. Your application may use either interface, depending on your specific goals for efficiency and portability. The application may not mix device-dependent and device-independent API calls, which are mutually exclusive.

The following information may be found in the public header file *lcd.h*:

Public Interface Function(s):

The device-independent API function prototypes:

```

/*****
* types_tHandle open (const char *pName, int OFlags, ...);
* ssize_t write(types_tHandle FileDesc, const void * pBuffer, size_t NBytes);
* UWord16 ioctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams);
* int close(types_tHandle FileDesc);
*****/

```

The device-dependent API function prototypes:

```

/*****
* types_tHandle lcdOpen (const char *pName, int OFlags);
* ssize_t lcdwrite(types_tHandle FileDesc, const void * pBuffer, size_t NBytes);
* UWord16 lcdIoctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams,
* const char * pName);
* int lcdClose(types_tHandle FileDesc);
*****/

```



6.5.4 LCD Device-Independent API Specification

The following pages specify the device-independent Application Programming Interface (API).

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

The *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.

6.5.4.1 open

Call(s):

```
types_tHandle open(const char *pName, int OFlags, ...);
```

Arguments:

Table 6-31. LCD Driver Arguments - open

<i>pName</i>	<i>in</i>	The LCD device name: BSP_DEVICE_NAME_LCD
<i>OFlag</i>	<i>in</i>	Open Mode Flags: O_RDWR - read and write allowed (default) O_BLOCK - blocking mode (default) O_NONBLOCK - non-blocking mode
<i>FileDesc</i>	<i>out</i>	LCD device descriptor of type (<i>int</i>) returned by the <i>open</i> call and used in <i>close/write/ioctl</i> function calls

Description: The *open* function is mapped to the *lcdOpen* function through the I/O interface Virtual Table (*lcdIOInterfaceVT*) at initialization. See *lcdOpen*, [Section 6.5.5.1](#).

Returns: See *lcdOpen*, [Section 6.5.5.1](#).

Code Example: [Code Example 6-14](#) opens the LCD for communication using default configuration settings.

Code Example 6-14. Use of open

```
File appconfig.h:
#define INCLUDE_BSP
#define INCLUDE_IO
#define INCLUDE_LCD

User application:
#include "port.h"
#include "bsp.h"
#include "lcd.h"

main()
{
    types_tHandle lcdFD;

    lcdFD = open(BSP_DEVICE_NAME_LCD, NULL);
}
```


6.5.4.2 write

Call(s):

```
ssize_t write(types_tHandle FileDesc, const void * pBuffer, size_t NBytes);
```

Arguments:

Table 6-32. LCD Driver Arguments - write

<i>FileDesc</i>	<i>in</i>	LCD device descriptor returned by the <i>open</i> call
<i>pBuffer</i>	<i>in</i>	Pointer to NULL terminated character array
<i>NBytes</i>	<i>in</i>	Number of characters to be written including control characters

Description: The *write* function is mapped to the *lcdWrite* function through the I/O interface Virtual Table (*lcdIOInterfaceVT*); see *lcdWrite*, [Section 6.5.5.2](#).

Returns: See *lcdWrite*, [Section 6.5.5.2](#).

Code Example: [Code Example 6-15](#) opens the driver in Blocking mode and writes a message to the display.

NOTE: A delay must be inserted at the application level to allow time for the user to read the display message.

Code Example 6-15. Use of write

File appconfig.h:

```
#define INCLUDE_BSP
#define INCLUDE_IO
#define INCLUDE_LCD
#define INCLUDE_TIMER
```

User application:

```
#include "port.h"
#include "timer.h"
#include "fcntl.h"
#include "bsp.h"
#include "lcd.h"

const char hello[] = "      Hello\n      World!";

main()
{
    types_tHandle lcdFD;
    struct timespec delay1s = {0, 1000000000};

    lcdFD = open(BSP_DEVICE_NAME_LCD, NULL);

    // Write characters to display
    write(lcdFD, &hello[0], sizeof(hello));
    nanosleep(&delay1s, NULL);
}
```


6.5.4.3 *ioctl*

Call(s):

```
UWord16 ioctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams);
```

Arguments:

Table 6-33. LCD Driver Arguments - *ioctl*

<i>FileDesc</i>	<i>in</i>	LCD device descriptor returned by <i>open</i> call
<i>Cmd</i>	<i>in</i>	Command for driver (See table in <i>lcdIoctl</i>)
<i>pParams</i>	<i>in</i>	Parameters specific to each command (See table in <i>lcdIoctl</i>)

Description: The LCD driver supports the same commands described for *lcdIoctl*; see *lcdIoctl*, [Section 6.5.5.3](#).

Return: See *lcdIoctl*, [Section 6.5.5.3](#).

Code Example: [Code Example 6-16](#) opens the LCD driver and sets the cursor position to the last location on line 1.

Code Example 6-16. Use of *ioctl*

File appconfig.h:

```
#define INCLUDE_BSP
#define INCLUDE_IO
#define INCLUDE_LCD
```

User application:

```
#include "port.h"
#include "bsp.h"
#include "lcd.h"

main()
{
    types_tHandle lcdFD;

    lcdFD = open(BSP_DEVICE_NAME_LCD, NULL);

    // Set Cursor position
    ioctl(lcdFD, LCD_SET_CURSOR_POSITION, LCD_LINE1+19);
}
```

6.5.4.4 *close*

Call(s):

```
int close (types_tHandle FileDesc);
```

Arguments:

Table 6-34. LCD Driver Arguments - *close*

<i>FileDesc</i>	<i>in</i>	LCD device descriptor returned by <i>lcdOpen</i> call

Description: The *close* function is mapped to the *lcdClose* function through the I/O interface Virtual Table (*lcdIOInterfaceVT*); see *lcdClose*, [Section 6.5.5.4](#).

Returns: This function returns “0” if successful, “-1” if unsuccessful.

Code Example: [Code Example 6-17](#) opens the LCD driver in Non-Blocking mode and writes a message to the display. Since Non-Blocking mode returns before the entire message has been sent, it is necessary to resend the “world” message until it has been accepted by the driver.

NOTE: A delay must be inserted at the application level to allow time for the user to read the display message.

Code Example 6-17. Use of *close*

```
File appconfig.h:
#define INCLUDE_BSP
#define INCLUDE_IO
#define INCLUDE_LCD
#define INCLUDE_TIMER

User application:
#include "port.h"
#include "timer.h"
#include "fcntl.h"
#include "bsp.h"
#include "lcd.h"

const char hello[] = "      Hello\n";
const char world[] = "      World!";

main()
{
    types_tHandle lcdFD;
    struct timespec delay1s = {0, 1000000000};

    lcdFD = open(BSP_DEVICE_NAME_LCD, O_NONBLOCK);

    // Write characters to display
    write(lcdFD, &hello[0], sizeof(hello));
    while(write(lcdFD, &world[0], sizeof(world))!=0);
    nanosleep(&delay1s, NULL);

    // Close LCD device
    close(lcdFD);
}
```

6.5.5 LCD Device-Dependent API Specification

The following pages specify the device-dependent Application Programming Interface (API).

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

The *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.

6.5.5.1 *lcdOpen*

Call(s):

```
types_tHandle lcdOpen(const char *pName, int OFlags);
```

Arguments:

Table 6-35. LCD Driver Arguments - *lcdOpen*

<i>pName</i>	<i>in</i>	The LCD device name: BSP_DEVICE_NAME_LCD
<i>OFlag</i>	<i>in</i>	Open Mode Flags: O_RDWR - read and write allowed (default) O_BLOCK - blocking mode (default) O_NONBLOCK - non-blocking mode
<i>FileDesc</i>	<i>out</i>	LCD device descriptor of type (<i>int</i>) returned by the <i>lcdOpen</i> call and used in <i>lcdClose</i> / <i>lcdWrite</i> / <i>lcdIoctl</i> function calls

Description: This function initializes the LCD peripheral for operation and establishes the handle for continued communication. It configures the driver for either Blocking or Non-Blocking mode, as defined in the OFlags parameter. In Non-Blocking mode, the Quad Timer driver is used to open and configure a timer used for periodically sending characters to the LCD.

Returns: The *lcdOpen* call returns a type *types_tHandle*. When the *lcdOpen* call is successful, this *types_tHandle* is a file descriptor for the LCD device. A "-1" is returned when the *lcdOpen* call is not successful.

Code Example: [Code Example 6-18](#) opens the LCD for communication using default configuration settings.

Code Example 6-18. Use of *lcdOpen*

File appconfig.h:

```
#define INCLUDE_BSP
#define INCLUDE_LCD
```

User application:

```
#include "port.h"
#include "bsp.h"
#include "lcd.h"
```

```
main()
{
    types_tHandle lcdFD;

    lcdFD = lcdOpen(BSP_DEVICE_NAME_LCD, NULL);
}
```

6.5.5.2 *lcdWrite*

Call(s):

```
ssize_t lcdWrite(types_tHandle FileDesc, const void * pBuffer, size_t NBytes);
```

Arguments:

Table 6-36. LCD Driver Arguments - *lcdWrite*

<i>FileDesc</i>	<i>in</i>	LCD device descriptor returned by the <i>lcdOpen</i> call
<i>pBuffer</i>	<i>in</i>	Pointer to NULL terminated character array
<i>NBytes</i>	<i>in</i>	Number of characters to be written including control characters

Description: The *lcdWrite* function writes a character array to the LCD display; the array is pointed to by *pBuffer*. This function searches the character array for the control character “\n”, and when encountered, will automatically fill to the end of the line with blank spaces. This feature saves data space normally required for storing blanks in the display strings.

In Blocking mode, *lcdWrite* will not return until all data has been sent to the LCD.

In Non-Blocking mode, *lcdWrite* starts the transfer operation and returns control to the application. It does not wait until all the data is transferred. In this mode, a timer interrupt is enabled to send the data, thereby giving more processing time to the main application.

Returns: This function returns the number of characters written to the display. In Blocking mode, if the driver is busy and a resend is required, the function returns “0”. In Non-Blocking mode, the function always returns “0”, since data is only queued to be sent by this function call.

Code Example: [Code Example 6-19](#) opens the driver in Blocking mode and writes a message to the display.

NOTE: A delay must be inserted at the application level to allow time for the user to read the display message.

Code Example 6-19. Use of *lcdWrite*

File appconfig.h:

```
#define INCLUDE_BSP
#define INCLUDE_LCD
#define INCLUDE_TIMER
```

User application:

```
#include "port.h"
#include "timer.h"
#include "fcntl.h"
#include "bsp.h"
#include "lcd.h"

const char hello[] = "      Hello\n      World!";

main()
{
    types_tHandle lcdFD;
    struct timespec delay1s = {0, 1000000000};
```



```
lcdFD = lcdOpen(BSP_DEVICE_NAME_LCD, NULL);
```

```
// Write characters to display  
lcdWrite(lcdFD, &hello[0], sizeof(hello));  
nanosleep(&delay1s, NULL);
```

```
}
```

6.5.5.3 *lcdloctl*

Call(s):

```
UWord16 lcdIoctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams,
                  const char * pName);
```

Arguments:

Table 6-37. LCD Driver Arguments - *lcdloctl*

<i>FileDesc</i>	<i>in</i>	LCD device descriptor returned by <i>lcdOpen</i> call
<i>Cmd</i>	<i>in</i>	Command for driver; see Table 6-38
<i>pParams</i>	<i>in</i>	Parameters specific to each command; see Table 6-38
<i>pName</i>	<i>in</i>	BSP device name from <i>bsp.h</i> : BSP_DEVICE_NAME_LCD

Description: The LCD driver supports many commands for dynamic configuration of the peripheral; see the list of commands described in [Table 6-38](#).

Returns: See *port.h* for PASS/FAIL definition. This function returns:

- “PASS” if the driver is able to send a command
- “FAIL” if the driver is busy and resend is required

Table 6-38. *lcdloctl* Commands

<i>Cmd</i>	<i>pParams</i>	<i>Result</i>
LCD_CLEAR_DISPLAY	NULL	Clears the display and sets the cursor to the top/left position
LCD_BLANK_DISPLAY	NULL	Blanks characters from view.
LCD_RESTORE_DISPLAY	NULL	Restores characters to view with cursor off
LCD_CURSOR_OFF	NULL	Turns cursor display off
LCD_CURSOR_UNDERLINE_ON	NULL	Displays the cursor as an underline
LCD_CURSOR_BLINK_ON	NULL	Displays the cursor as a blinking 5x7 solid character
LCD_CURSOR_LEFT	NULL	Moves the cursor one character position left
LCD_CURSOR_RIGHT	NULL	Moves the cursor one character position right
LCD_SCROLL_LEFT	NULL	Scrolls one character to the left and lines 1 and 2 to lines 3 and 4. The first character on line 1 will appear as the last character on line 3. The first character on line 3 will appear as the last character on line 1. Lines 2 and 4 operate similarly. For an illustration, run the LCD Test described in Section 6.5.7.2 .

Table 6-38. *IcdIoctl* Commands (Continued)

<i>Cmd</i>	<i>pParams</i>	Result
LCD_SCROLL_RIGHT	NULL	Scrolls one character to the right and lines 1 and 2 to lines 3 and 4. The last character on line 1 will appear as the first character on line 3. The last character on line 3 will appear as the first character on line 1. Lines 2 and 4 operate similarly. For an illustration, run the LCD Test described in Section 6.5.7.2 .
LCD_SET_CURSOR_POSITION	LCD_LINE <i>n</i> +offset	Sets cursor position as specified by <i>pParams</i>. Use <i>LCD_LINE</i><i>n</i>+<i>offset</i> to specify the desired position where <i>LCD_LINE</i><i>n</i> is LCD_LINE1, LCD_LINE2, LCD_LINE3, or LCD_LINE4. Example: The last character position on line 1 is represented by (<i>LCD_LINE1</i>+19). Maximum offset is 19.

Code Example: [Code Example 6-20](#) opens the driver and sets the cursor position to the last location on line 1.

Code Example 6-20. Use of *IcdIoctl*

File appconfig.h:

```
#define INCLUDE_BSP
#define INCLUDE_LCD
```

User application:

```
#include "port.h"
#include "bsp.h"
#include "lcd.h"

main()
{
    types_tHandle lcdFD;

    lcdFD = lcdOpen(BSP_DEVICE_NAME_LCD, NULL);

    // Set Cursor position
    lcdIoctl(lcdFD, LCD_SET_CURSOR_POSITION, LCD_LINE1+19,
             BSP_DEVICE_NAME_LCD);
}
```


6.5.5.4 *lcdClose*

Call(s):

```
int lcdClose (types_tHandle FileDesc);
```

Arguments:

Table 6-39. LCD Driver Arguments - *lcdClose*

<i>FileDesc</i>	<i>in</i>	LCD device descriptor returned by <i>lcdOpen</i> call

Description: The *lcdClose* function closes the LCD device. It waits until all command and message processing is complete, then issues an LCD_CLEAR_DISPLAY command and closes all associated drivers.

Returns: This function returns “0” if successful and “-1” if unsuccessful.

Code Example: **Code Example 6-21** opens the driver in Non-Blocking mode and writes a message to the display. Since Non-Blocking mode returns before the entire message has been sent, it is necessary to resend the “world” message until it has been accepted by the driver.

NOTE: A delay must be inserted at the application level to allow time for the user to read the display message.

Code Example 6-21. use of *lcdClose*

File appconfig.h:

```
#define INCLUDE_BSP
#define INCLUDE_LCD
#define INCLUDE_TIMER
```

User application:

```
#include "port.h"
#include "timer.h"
#include "fcntl.h"
#include "bsp.h"
#include "lcd.h"

const char hello[] = "      Hello\n";
const char world[] = "      World!";

main()
{
    types_tHandle lcdFD;
    struct timespec delay1s = {0, 1000000000};

    lcdFD = lcdOpen(BSP_DEVICE_NAME_LCD, O_NONBLOCK);

    // Write characters to display
    lcdWrite(lcdFD, &hello[0], sizeof(hello));
    while(lcdWrite(lcdFD, &world[0], sizeof(world))!=0);
    nanosleep(&delay1s, NULL);

    // Close LCD device
    lcdClose(lcdFD);
}
```

6.5.6 Demo Applications

The LCD Driver is used in the Feature Phone Demo Application located in the following directory:

.../sdk/src/dsp56852evm/nos/applications/telephony/featurephone

6.5.7 Test Applications

The LCD Driver Test application verifies the correct operation of all driver APIs. This application is located in the following directory:

.../sdk/src/dsp56852evm/nos/bsp/test/tdc1-LCD/lcdtest.mcp

6.5.7.1 Hardware Set-up

This test requires a TDC1 attached to the DSP56852EVM via the daughter card connector. Production EVMs supply 5V and GND through the daughter card connector. Set jumper JG5 on the TDC1 card.

NOTE: Use default settings shown in the DSP56852 Evaluation Module User's Manual.

6.5.7.2 Procedure for LCD Test

- Using CodeWarrior, open:
.../nos/bsp/test/tdc1-LCD/lcdtest.mcp
- Build and download the External RAM target; when the debugger reaches *main()*, it will stop
- The display should be initialized and cleared at this point
- To continue executing the test, select *Run* in the debugger
- The LCD display will show the following test results:
 - LCD Driver Opened
!\"#\$%&'()*+,-./01234
56789:;<=>?@ABCDEFGH
KLMNOPQRST[\]^_`{|}~
 - Show Line 1 Clear 2
 - Clear 3 Show Line 4
 - Blank and Restore
 - Cursor Options Blink & Move Left
 - Cursor Options Underline Move Right
 - Scroll Left
 - Scroll Right
 - Clear the Entire Display
 - LCD Driver Closed
- Repeat this procedure using the Internal RAM target

6.6 LED Driver

6.6.1 Introduction

The LED interface manipulates the light emitting diodes (LEDs).

There are five LEDs on the DSP56852EVM, controlled by GPIO and accessible using the SDK LED Driver. Since there are no dedicated GPIO for the DSP56852, the user must decide carefully which LEDs to use to avoid interfering with the operation of a peripheral that is multiplexed with the same GPIO that controls the LED. [Table 6-40](#) shows which LEDs and peripherals are multiplexed.

Table 6-40. LEDs and Peripheral Multiplexing

LED	Peripheral
LED_RED2	SPI or SSI
LED_YELLOW	SPI or SSI
LED_YELLOW2	SCI or Codec
LED_GREEN	SPI or SSI
LED_GREEN2	SCI or Codec

6.6.2 Initialization

The LED driver currently has no default configurations that can be overwritten through a project's *appconfig.h* file.

To include the LED Driver in the application, insert the following line into the *appconfig.h* file:

```
#define INCLUDE_LED
```

Defining the preprocessor variable, INCLUDE_LED, directs the SDK to include the LED driver in the executable image. If the preprocessor variable INCLUDE_LED is not defined, then the LED driver will not be included in the executable.

6.6.3 API Definition

This section defines the Application Programming Interface (API) by identifying all public interface functions and data structures.

The LED interface can be used at two alternative levels, a common IO layer interface and a low-level LED driver interface. The common IO layer interface invokes the lower-level LED driver interface.

The IO layer API to the LED interface is standard and more portable than the low-level LED interface, but potentially less efficient. The low-level LED driver provides a non-standard interface that is potentially more efficient than the IO layer calls, but less portable.

Your application may use either the IO layer interface or the low-level LED driver interface to the LED driver, depending on your specific goals for efficiency and portability. However, you should not mix the LED driver APIs in your application. In particular, you **must not** use the File Descriptor returned by the IO layer *open* function with any low-level LED Driver function call, nor may you use the File Descriptor returned by the low-level driver *ledOpen* call with any IO layer interface function call.

You must include *led.h* in your application to utilize the LED Driver API. The following information may be found in the public header file *led.h*:

Public Interface Function(s):

```

/*****
* The IO layer interface defines functions as follows:
*   types_tHandle open(const char *pName, int OFlags);
*   int          close(types_tHandle FileDesc);
*   UWord16      ioctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams);
*
* The low level Button driver interface defines functions as follows:
*   types_tHandle ledOpen(const char *pName, int OFlags);
*   int          ledClose(types_tHandle FileDesc);
*   UWord16      ledIoctl(types_tHandle FileDesc, UWord16 Cmd, UWord16 led,
*                        const char *ledDevice);
*****/

```

Public Data Structure(s):

None

Members:

None

6.6.4 LED Device-Independent I/O API Specifications

The following pages specify device-independent Application Programming Interface (API).

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

6.6.4.1 *open*

Call(s):

```
types_tHandle open(const char *pName, int OFlags);
```

Arguments:

Table 6-41. LED Driver Arguments - *open*

<i>pName</i>	<i>in</i>	The LED device name; use BSP_DEVICE_NAME_LED_0
<i>OFlag</i>	<i>in</i>	Open Mode Flags which are ignored
<i>FileDesc</i>	<i>out</i>	LED device descriptor returned by <i>open</i> call and used in <i>ioctl</i> function calls

Description: The *open* call opens the LED driver for operations and returns a file descriptor used by *ioctl* function calls. The LED driver must be opened for *ioctl* calls.

Returns: The *open* call returns a type *types_tHandle*. When the *open* call is successful, this *types_tHandle* is a file descriptor for the LED driver and is used by the LED driver's *close* and *ioctl* functions. A “-1” is returned when the *open* call is not successful.

Code Example: See [Code Example 6-22](#).

6.6.4.2 *close*

Call(s):

```
int close (types_tHandle FileDesc);
```

Arguments:

Table 6-42. LED Driver Arguments - *close*

<i>FileDesc</i>	<i>in</i>	LED device descriptor returned by <i>open</i> call and used in <i>ioctl</i> function calls

Description: The *close* call closes the LED driver.

Returns: The *close* call returns a type *int*. This *int* is always zero.

Code Example: See [Code Example 6-22](#).

6.6.4.3 *ioctl*

Call(s):

```
UWord16 ioctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams);
```

Arguments:

Table 6-43. LED Driver Arguments - *ioctl*

<i>FileDesc</i>	<i>in</i>	LED device descriptor returned by <i>open</i> call and used in <i>ioctl</i> function calls
<i>Cmd</i>	<i>in</i>	Commands found in <i>led.h</i> which are used to modify specific LEDs on EVM board
<i>pParams</i>	<i>in</i>	Specific LED; see Table 6-45

Table 6-44. LEDs for Specific EVM Boards (*pParams* - input parameter)

DSP56852EVM
LED_RED2 LED_YELLOW LED_YELLOW2 LED_GREEN LED_GREEN2

Table 6-45. LED *ioctl* Commands

Cmd	<i>pParams</i>	Return	Result
LED_OFF	Specific LED; see Table 6-44	None	Turns specific LED off
LED_ON	Specific LED; see Table 6-44	None	Turns specific LED on
LED_TOGGLE	Specific LED; see Table 6-44	None	Toggles specific LED

Description: The *ioctl* call manipulates the LEDs on the EVM board.

Return: Although the *ioctl* function is specified to return a *UWord16* value, this return value is not returned by this driver and should not be checked by the application.

Code Example: [Code Example 6-22](#) toggles a green LED.

Code Example 6-22. LED Driver Usage

```
/* **** */
#include "io.h"
#include "fcntl.h"
#include "bsp.h"

#include "led.h"

types_tHandle LedFD;

/* open LED driver */
LedFD = open(BSP_DEVICE_NAME_LED_0, 0);

/* toggle Green LED */
ioctl(LedFD, LED_TOGGLE, LED_GREEN);

/* close LED driver */
close(LedFD);

/* **** */
```



6.6.5 Low-Level Device Drivers I/O API Specifications

The following pages specify the low-level Application Programming Interface (API).

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

6.6.5.1 *ledOpen*

Call(s):

```
types_tHandle ledOpen(const char *pName, int OFlags);
```

Table 6-46. Low-Level LED Driver Arguments - *ledOpen*

<i>pName</i>	in	The LED device name; use BSP_DEVICE_NAME_LED_0
<i>OFlag</i>	in	Open Mode Flags which are ignored

Description: The *ledOpen* call opens the LED driver for operations and returns a file descriptor used by *ledIoctl* function calls. The LED driver must be opened for *ledIoctl* calls.

Returns: The *open* call returns a type *types_tHandle*. When the *open* call is successful, this *types_tHandle* is a file descriptor for the LED driver and is used by the LED driver’s *close* and *ioctl* functions. If the *open* call is not successful, a “-1” is returned.

Code Example: See [Code Example 6-23](#).



6.6.5.2 *ledClose*

Call(s):

```
int ledClose (types_tHandle FileDesc);
```

Table 6-47. LED Driver Arguments - *ledClose*

<i>FileDesc</i>	<i>in</i>	LED device descriptor returned by <i>ledOpen</i> call and used in <i>ioctl</i> function calls

Description: The *ledClose* call closes the LED driver.

Returns: The *ledClose* call returns a type *int*. This *int* is always zero.

Code Example: See [Code Example 6-23](#).

6.6.5.3 *ledIoctl*

Call(s):

```
UWord16 ledIoctl(types_tHandle FileDesc, UWord16 Cmd, UWord16 led, const char
                  *ledDevice);
```

Table 6-48. Low-Level LED Driver Arguments

<i>FileDesc</i>	<i>in</i>	LED device descriptor returned by <i>ledOpen</i> call and used in <i>ledIoctl</i> function calls
<i>Cmd</i>	<i>in</i>	Commands found in <i>led.h</i> which are used to modify specific LEDs on EVM board
<i>led</i>	<i>in</i>	Pin on which to perform the command, denoted as LED_GREEN, LED_YELLOW, LED_RED2, LED_GREEN2, LED_YELLOW2
<i>ledDevice</i>	<i>in</i>	The LED device name from <i>bsp.h</i> ; typically, BSP_DEVICE_NAME_LED_0

Description: The *ledIoctl* call manipulates the LEDs on the EVM board.

Return: None

Code Example: See [Code Example 6-23](#).

Code Example 6-23. Low-Level LED Driver Usage

```

/*****/
#include "bsp.h"
#include "led.h"

void main(void)
{
    types_tHandle ledFD;

    /* Open LED driver */
    ledFD = ledOpen(BSP_DEVICE_NAME_LED_0,0);

    /* Toggle the Green LED */
    ledIoctl(ledFD, LED_TOGGLE, LED_GREEN, BSP_DEVICE_NAME_LED_0);

    /* Close the LED driver */
    ledClose(ledFD);
}
/*****/

```

6.6.6 LED Driver Tests

The LED Driver Tests verify the correct operation of the LED Driver provided by the SDK.

6.6.6.1 LED Driver Test

This test flashes all six LEDs at the same time.

6.6.6.1.1 Set-up for LED Driver Test

EVM Jumper Settings:

Use default settings as shown in the **DSP56852 Evaluation Module User's Manual**.

6.6.6.1.2 Procedure for LED Driver Test

- Using CodeWarrior, open:
...\\nos\\bsp\\test\\ledtest\\testled.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- All 6 LEDs should begin flashing at the same time

6.6.7 LED Driver Applications

There is not a specific application demonstrating just the LED driver. There are many applications that use the LED Driver including the Codec, Timer, and Quad Timer applications.

6.7 PC Master Software SCI Communication Driver

6.7.1 Introduction

The PC master software SCI communication driver communicates with the PC master software Windows application to develop and debug target applications. A description of the communication protocol can be found in the **PC Master Software Communication Library** Help files. The *pcmasterdrvIsr* function is the main function which is called from the SCI interrupt service routine. The *pcmasterdrvInit* initializes the SCI port to enable communication. The *pcmasterdrvGetAppCmdSts* and *pcmasterdrvWriteAppCmdSts* read and write the application command status for the user application. The *pcmasterdrvRecorder* samples data into an internal data buffer. This driver is automatically installed and initialized by defining INCLUDE_PCMASTER in the *appconfig.h* file. This driver requires the use of the SCI port, thereby prohibiting the use of this port by the application for any other use. The PC master software SCI communication driver utilizes the SCI driver, which is configured to run at a rate of 9600 baud. For an overview of PC master software, please refer to the **PC Master Software User's Manual**.

6.7.2 Initialization

The PC master software driver has default configurations, defined in *config.h*, which can be redefined in the *appconfig.h* for a project. See [Table 6-49](#) for details.

Table 6-49. PC Master Software *appconfig.h* Settings

PC Master Software Defined Constants	Default	Explanation
PC_MASTER_APPCMD_BUFF_LEN	0	Application Command buffer length
PC_MASTER_REC_BUFF_LEN	40	Recorder buffer length
PC_MASTER_RECORDER_TIME_BASE	0x8030	Recorder timebase = 48us

6.7.3 API Definitions

This section defines the Application Programming Interface (API) and public data structures. For a more detailed description of this interface, see [Section 6.7.4](#).

The header file *pcmasterdrv.h* includes all required prototypes and structure/type definitions.

Public Interface Function(s):

```

/*****
Initialization of PC Master Communication Algorithm
This function must be called first, before start of communication.
Parameter passed to this function is variable of sPCMasterComm type
*****/

```

```

Word16 pcmasterdrvInit(sPCMasterComm *p_sPCMasterComm);

/*****
Get Application Command Status
This function is used to check if an application command has been received from PC
*****/
UWord16 pcmasterdrvGetAppCmdSts(void);
/*****
Write Application Command Status
This function clears the flags in application command status word
and it says to PC that the last application command was served and
new application command from PC will be accepted
*****/
Word16 pcmasterdrvWriteAppCmdSts(UWord16 state);

/*****
Main PC Master Communication routine which provide receiving,
decoding of incoming message and sending response to PC
*****/
void pcmasterdrvIsr(void);
/*****
Recorder Routine
It performs sampling of data into buffer which is located
on the address p_recBuff item is pointing and its length
is determined by recSize item in the initialization structure
of sPCMasterComm type
*****/
void pcmasterdrvRecorder(void);

```

Public Data Structure(s):

The structure *sPCMasterComm* is defined as shown in [Table 6-50](#)

```

/*****
structure with SCI communication settings
*****/
typedef struct {
    UInt8      *p_dataBuff; /* pointer to input/output communication buffer */
    UInt8      dataBuffSize; /* size of input/output communication buffer */
    UInt8      *p_recBuff; /* pointer to recorder buffer */
    UWord16    recSize; /* recorder buffer size */
    UInt8      *p_recorder; /* structure with recorder settings and
                             temporary variables */
    UInt8      *p_scope; /* structure with scope settings */
    UWord16    timeBase; /* period of Recorder Routine launch */
    UInt8      *p_appCmdBuff; /* pointer to application command buffer */
    UInt8      appCmdSize; /* application command buffer size */
    UInt8      globVerMajor; /* board firmware version major number */
    UInt8      globVerMinor; /* board firmware version minor number */
    UInt8      idtString[PCMDRV_IDT_STRING_LEN]; /* device identification
                                                  string */
} sPCMasterComm;

/*****
recorder settings structure (recorder settings and temporary variables are stored
in this structure)

```




*****/

```
typedef struct{
    UWord16      cmd;
    UWord16      cmdLen;
    UWord16      trgMode;      /* 0x00 - manual, 0x01 - rising edge,
                                0x02 - falling edge */
    UWord16      totalSmps;    /* number of samples required */
    UWord16      postTrigger;  /* samples after trigger */
    UWord16      timeDiv;      /* time div */
    UWord32      trgVarAddr[1]; /* address of trigger variable */
    UWord16      trgVarSize;    /* size of variable (bytes) */
    UWord16      trgVarSigned; /* 0x00 - unsigned, 0x01 - signed */
    union{          /* union is used to access various types of threshold */
        UInt8      uch;        /* trgVarSize == 1 */
        Int8       ch;         /* trgVarSize == 1 */
        UWord16    uw;         /* trgVarSize == 2 */
        UWord32    ud;         /* trgVarSize == 2 */
        Word16     sw;         /* trgVarSize == 4 */
        Word32     sd;         /* trgVarSize == 4 */
    } trgThreshold; /* trigger threshold */
    UWord16      varCnt;        /* number of variables */
    struct{
        UWord16      varSize;    /* size of variable */
        UWord32      varAddr[1]; /* address of variable */
    } varDef[8];

    UWord16      recPos;        /* position in recorder buffer;
                                position of the next samples
                                (incremented with RecSetLen) */
    UWord16      recSetLen;     /* length of required set of variables
                                (in words) */
    UWord16      recToEnd;     /* position to end of buff (the
                                variable is decremented after trigger
                                & recorder stops at 0) */
    UWord16      recTime;       /* time div of Recorder */

    /*recorder last value (last value the triggering variable is stored
    after launch of Recorder routine) */
    union{
        UInt8      uch;
        Int8       ch;
        UWord16    uw;
        UWord32    ud;
        Word16     sw;
        Word32     sd;
    } recLastVal; /* last value of synchronizing variable */
} pcmdrv_sRecorder;

/*****
scope settings buffer (scope settings are stored in this structure)
*****/
typedef struct{
    unsigned char varCnt;
    struct{
```

```

    UInt8 varSize;          /* size of scope variable */
    UInt8 addrByte[4];      /* address of scope variable */
} varDef[8]; /* maximum number of variables is 8 */
} pcmdrv_sScope;

```

Table 6-50. sPCMasterComm structure

variable	type	explanation
<i>p_recBuff</i>	UInt8	Pointer pointing to start of recorder buffer
<i>recSize</i>	UWord16	Size of recorder buffer
<i>timeBase</i>	UWord16	Period of Recorder Routine launch; see table Table 6-51
<i>p_appCmdBuff</i>	UInt8	Pointer pointing to start of buffer for application command calls
<i>appCmdSize</i>	UInt8	Size of user functions buffer
<i>globVerMajor</i>	UInt8	Board firmware version major number
<i>globVerMinor</i>	UInt8	Board firmware version minor number
<i>idtString[25]</i>	UInt8	Identification string

The *timeBase* variable in the *sPCMasterComm* structure is coded in the format shown in [Table 6-51](#).

Table 6-51. timeBase format

bits	format	explanation
0-13	14-bits	Mantissa of Recorder Routine launch period
14-15	2-bits	Exponent of Recorder Routine launch period 0- 1 (10^0) 1 - m (10^{-3}) 2- u (10^{-6}) 3- n (10^{-9})

Example of *timeBase* variable coding:

- 1) Frequency of the *pcmasterdrvRecorder* routine call is 48us.
timeBase = 0x8030
- 2) Frequency of the *pcmasterdrvRecorder* routine call is 100ms.
timeBase = 0x4064



6.7.4 PC Master Software Low-Level I/O API Specification

The following pages specify the device-independent Application Programming Interface (API).

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

6.7.4.1 *pcmasterdrvInit* - Initialize PC Master Software Communication

Call(s):

```
Word16 pcmasterdrvInit(sPCMasterComm *p_sPCMasterComm);
```

Arguments:

Table 6-52. *pcmasterdrvInit* parameters

<i>p_sPCMasterComm</i>	<i>in</i>	Pointer to structure <i>sPCMasterComm</i> with initialization data

Description: The *pcmasterdrvInit* function initializes receiver functions and internal variables. The *pcmasterdrvInit* function is called in the *config.c* file; therefore, the user need not call this function from an application.

Returns: If initialization was successful, *pcmasterdrvInit* returns “0”. If the SCI device open failed, *pcmasterdrvInit* returns “-1”.

Range Issues: None

Special Issues: None

Code Example: None

6.7.4.2 *pcmasterdrvGetAppCmdSts* - Get Application Command Status

Call(s):

```
UWord16 pcmasterdrvGetAppCmdSts(void);
```

Arguments: None

Description: This function confirms whether a new application command call has been received. If a new application command has been received, data is in the *PCMasterCommApplicationCommandBuffer*; an application command code is at the first position and the parameters of the application command are in the buffer. The length of this buffer is defined by *PC_MASTER_APPCMD_BUFF_LEN* in the *config.h* file. The parameters are passed to the *pcmasterdrvInit* function during initialization; see [Table 6-50](#) for details.

To enable application command calls, define the length of the application command buffer by setting *PC_MASTER_APPCMD_BUFF_LEN* in the *appconfig.h* file. The application command buffer, *PCMasterCommApplicationCommandBuffer*, will be declared automatically.

Returns: If no application command has been received, the *pcmasterdrvGetAppCmdSts* returns “0”; if the *PCMasterCommApplicationCommandBuffer* contains data (a new application command has been received), the *pcmasterdrvGetAppCmdSts* returns “-1”.

Range Issues: None

Special Issues: The application command status returned by the *pcmasterdrvGetAppCmdSts* function must be rewritten with the *pcmasterdrvWriteAppCmdSts* function to enable receipt of the next application commands; see [Section 6.7.4.3](#).

Code Example: None

6.7.4.3 *pcmasterWriteAppCmdSts* - Write Application Command Status

Call(s):

```
Word16 pcmasterdrvWriteAppCmdSts(UWord16 state);
```

Table 6-53. *pcmasterdrvWriteAppCmdSts* arguments

<i>state</i>	<i>in</i>	Application command return code

Description: The *pcmasterdrvWriteAppCmdSts* function stores a *state* value to be sent to the PC as an application command return code. If the status of the application command call is not changed to the Range shown below after its execution, no new application command can be called.

To enable application command calls, define the length of the application command buffer by setting the *PC_MASTER_APPCMD_BUFF_LEN* in the *appconfig.h* file. The application command buffer *PCMasterCommApplicationCommandBuffer* will be declared automatically.

To service an application command call, follow these steps:

- To check whether a new command has been received, call the *pcmasterdrvGetAplCmdSts* function
- If a new command has been received, perform the action related to the application command received
- Write back an application command return code using the *pcmasterdrvWriteAppCmdSts* function

Returns: The *pcmasterdrvWriteAppCmdSts* function unconditionally returns “0”.

Range Issues: Only the low byte of *state* is used. The value of the *state* must be within the range: $0x0000 \leq state \leq 0x00FD$

Special Issues: None

Code Example: None

6.7.4.4 *pcmasterdrvIsr* - Main Communication Function**Call(s):**

```
void pcmasterdrvIsr(void);
```

Arguments: None

Description: The function *pcmasterdrvIsr* is the main communication function and decodes received messages and executes proper commands. This function is called in SCI0 receiver and transmitter interrupts.

Returns: None**Range Issues:** None

Special Issues: The *pcmasterdrvIsr* function is installed to be called in receiver and transmitter interrupts of the SCI0 module.

Code Example: None

6.7.4.5 *pcmasterdrvRecorder* - Sample Recorded Variables

Calls:

```
void pcmasterdrvRecorder(void);
```

Arguments: None

Description: The function *pcmasterdrvRecorder* provides a sampling of required variables. The *pcmasterdrvRecorder* function should be called periodically (e.g., using timer interrupt) to provide a constant sampling period. The *pcmasterdrvRecorder* function uses the *PCMasterCommRecorderBuffer* defined in *config.c* file. The length of the buffer is defined by *PC_MASTER_REC_BUFF_LEN* and the time base of the *pcmasterdrvRecorder* routine is defined by *PC_MASTER_RECORDER_TIME_BASE* in the *config.h* file. The parameters are passed to the *pcmasterdrvInit* function during initialization; see [Table 6-50](#) and [Table 6-51](#) for details.

To use the recorder follow these steps:

- Set *PC_MASTER_REC_BUFF_LEN* to define the recorder buffer length
- The recorder buffer *PCMasterCommRecorderBuffer* will be declared automatically
- Call the *pcmasterdrvRecorder* function periodically

Returns: Void

Range Issues: None

Special Issues: None

Code Example: None

6.7.5 PC Master Software Tests

Currently, there are no stand-alone PC master software test applications.

6.7.6 PC Master Software Applications

6.7.6.1 PC Master Software Demo

This application shows how to initialize PC master software communication to allow a PC to control the application. The PC master software communication functions use the SCI0 on-chip peripheral module.

The PC master software demo application for DSP56852EVM increments three internal variables (*test16*, *fast16*, *test32*) to obtain sawtooth signals, which it can display.

6.7.6.1.1 Set-up for PC Master Software Demo

EVM Jumper Settings:

Use default settings as shown in the **DSP56852 Evaluation Module User's Manual**.

6.7.6.1.2 Procedure for PC Master Software Demo

- Using CodeWarrior, open:
...\\sdk\\src\\dsp56852evm\\nos\\applications\\bsp\\pcmaster\\pcmaster_test.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- Open the PC master software application executable which can be found in:
...\\sdk\\pc_master\\PCMaster.exe
- Using the PC master software executable, open the PC master software project:
...\\sdk\\src\\dsp56852evm\\nos\\applications\\bsp\\pcmaster\\pcmaster_test.pmp
- At this time, all prepared variables should be read periodically. The scope is also prepared to show its functionality. An example of the PC master software GUI window with an opened demo project and activated scope is shown in [Figure 6-2](#).

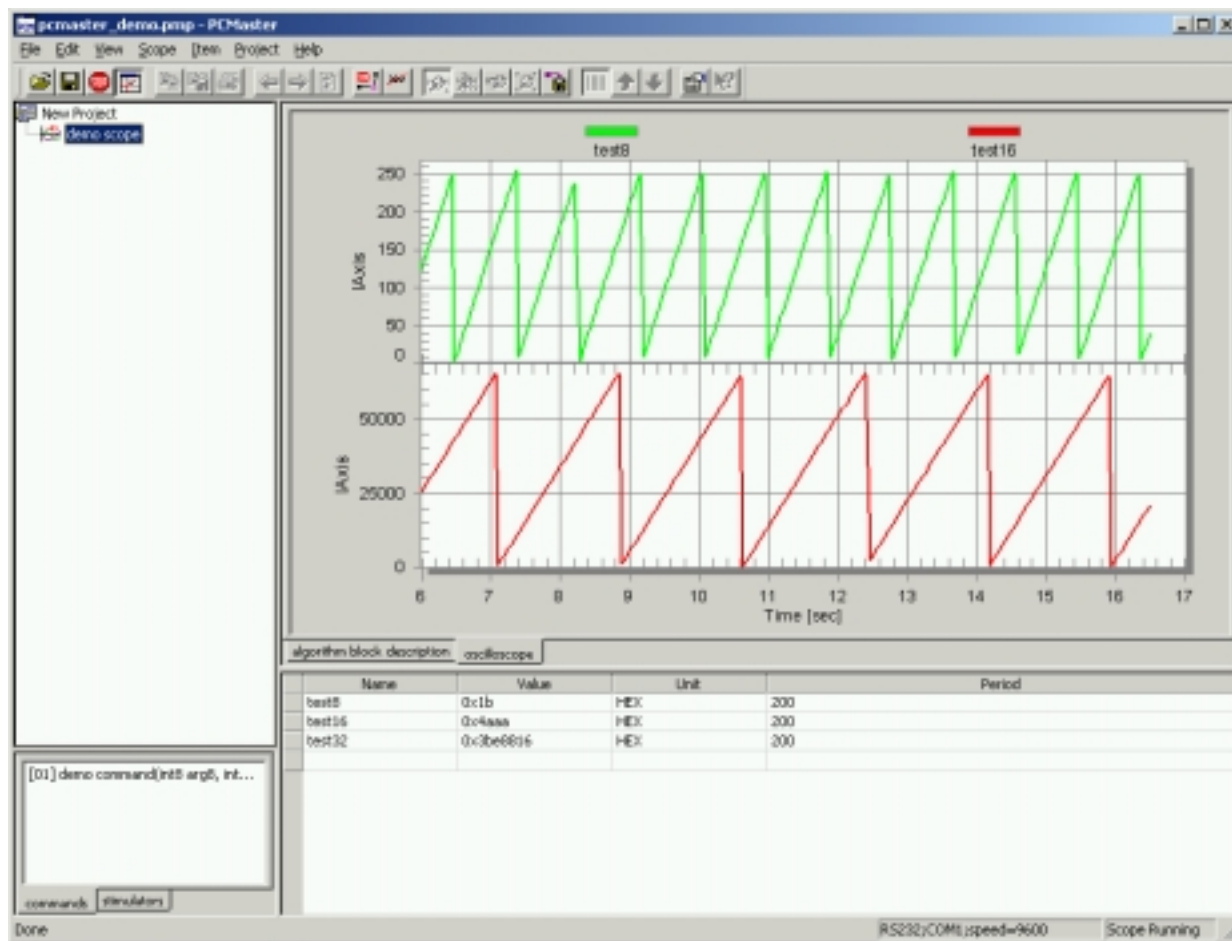


Figure 6-2. PC Master Software Application Window

6.8 Serial Data Flash Driver

The Atmel AT45DB011 device is a 1Mbit Serial Data Flash. It is connected to DSP56852 SPI0 interface on the DSP56852EVM board. The AT45DB011 Serial Data Flash has a 0x21000 * 8 bit organization, so the internal Serial Data Flash address range is from 0x0000 to 0x20fff. The Serial Data Flash driver deals with words, not with bytes. Word data written from the driver to the Serial Data Flash device is located MSB first.

6.8.1 Introduction

The Serial Data Flash driver is designed for the Atmel AT45DB011 SPI Data Flash hardware located on the DSP56852EVM board.

The Serial Data Flash driver supports internal Data Flash address to perform *read* and *write* operations. After the *open* and *reset* calls, the internal Data Flash address is equal to 0x0000. Each *read* or *write* call increments the internal Data Flash address by the data length multiplied by two.

If the user tries to access data beyond the 0x21fff internal Data Flash address boundary, the services operate only within this boundary and return the actual processed data length. To change the internal address, use the *ioctl* function, but note that the address should be even.

The Serial Data Flash Driver has a verification mode switch. If verification mode is off, *write* and *read* services work as usual. While verification mode is on, the *write* service performs write data into Serial Data Flash and read data from Serial Data Flash. If some read data is not equal to the written data, the service returns zero; in other cases, the service returns the actual written data length. While verification is on, the *read* service does not perform reading data from Serial Data Flash to the user buffer, but only compares data from the user buffer against the Serial Data Flash data. If the data is not equal, the service returns zero; in other cases, the service returns the actual verified data length. To change the verification mode, use the *ioctl* function.

6.8.2 Initialization

The Serial Data Flash device driver currently has no default configurations that can be overwritten through a project's *appconfig.h* file.

To include the Serial Data Flash Driver in the application, insert the following line into the *appconfig.h* file:

```
#define INCLUDE_SERIAL_DATAFLASH
```

Defining the preprocessor variable, *INCLUDE_SERIAL_DATAFLASH*, directs the SDK to include the Serial Data Flash driver into the executable image. If the preprocessor variable *INCLUDE_SERIAL_DATAFLASH* is not defined, then the Serial Data Flash driver will not be included in the executable.

6.8.3 API Definition

This section defines the Application Programming Interface (API) by identifying all public interface functions and data structures.

The following information may be found in the public header file *serialdataflash.h*. You must include *serialdataflash.h* in your application to utilize the Serial Data Flash Driver API.

Public Interface Function(s):

```

/*****
*      types_tHandle open(const char *pName, int OFlags);
*      ssize_t write(types_tHandle FileDesc, const void * pBuffer, size_t NBytes);
*      ssize_t read(types_tHandle FileDesc, void * pBuffer, size_t NBytes);
*      int close(types_tHandle FileDesc);
*      UWord16 ioctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams);
*****/

```

Public Data Structure(s):

None

Members:

None



6.8.4 Serial Data Flash Device-Independent I/O API Specifications

The following pages specify device-independent Application Programming Interface (API).

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

6.8.4.1 open

Call(s):

```
types_tHandle open(const char *pName, int OFlags);
```

Arguments:

Table 6-54. Serial Data Flash Driver Arguments - open

<i>pName</i>	<i>in</i>	The Serial Data Flash device name; use BSP_DEVICE_NAME_SERIAL_DATAFLASH_0
<i>OFlag</i>	<i>in</i>	Open Mode Flags which are ignored
<i>FileDesc</i>	<i>out</i>	Serial Data Flash device descriptor returned by the <i>open</i> call and used in <i>read/write</i> function calls

Description: The *open* call opens the Serial Data Flash device for operations and returns a file descriptor used by the *read/write/ioctl* function calls. The Serial Data Flash device needs to be opened for *read/write* calls.

Returns: The *open* call returns a type *types_tHandle*. When the *open* call is successful, this *types_tHandle* is a file descriptor for the Serial Data Flash device and is used by the Serial Data Flash driver's *read*, *write*, *close*, and *ioctl* functions. A "-1" is returned when the *open* call is not successful.

Code Example: See [Code Example 6-24](#).

6.8.4.2 **write**

Call(s):

```
ssize_t write(types_tHandle FileDesc, const void * pBuffer, size_t NBytes);
```

Arguments:

Table 6-55. Serial Data Flash Driver Arguments - *write*

<i>FileDesc</i>	<i>in</i>	Serial Data Flash device descriptor returned by the <i>open</i> call and used in <i>read/write</i> function calls
<i>pBuffer</i>	<i>in</i>	Pointer to user buffer
<i>NBytes</i>	<i>in</i>	Size of data to <i>read/write</i> in bytes

Description: The *write* call writes the data from the user buffer to the Serial Data Flash device.

Returns: The *write* call returns a type *ssize_t*. This *ssize_t* is the number of transferred bytes.

Code Example: See [Code Example 6-24](#).

6.8.4.3 *read*

Call(s):

```
ssize_t read(types_tHandle FileDesc, void * pBuffer, size_t NBytes);
```

Arguments:

Table 6-56. Serial Data Flash Driver Arguments - *read*

<i>FileDesc</i>	<i>in</i>	Serial Data Flash device descriptor returned by the <i>open</i> call and used in <i>read/write</i> function calls
<i>pBuffer</i>	<i>in</i>	Pointer to user buffer
<i>NBytes</i>	<i>in</i>	Size of data to <i>read/write</i> in bytes

Description: The *read* call reads data from the Serial Data Flash device into the user buffer.

Returns: The *read* call returns a type *ssize_t*, which is the number of received bytes.

Code Example: See [Code Example 6-24](#).



6.8.4.4 *close*

Call(s):

```
int close (types_tHandle FileDesc);
```

Arguments:

Table 6-57. Serial Data Flash Driver Arguments - *close*

<i>FileDesc</i>	<i>in</i>	Serial Data Flash device descriptor returned by the <i>open</i> call and used in <i>read/write</i> function calls

Description: The *close* call closes the Serial Data Flash peripheral and releases the file descriptor handle.

Returns: The *close* call returns a type *int*, which is always zero.

Code Example: See [Code Example 6-24](#).

6.8.4.5 *ioctl*

Call(s):

```
UWord16 ioctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams);
```

Arguments:

Table 6-58. Serial Data Flash Driver Arguments

<i>FileDesc</i>	<i>in</i>	Serial Data Flash device descriptor returned by the <i>open</i> call and used in <i>read/write</i> function calls
<i>Cmd</i>	<i>in</i>	Commands found in <i>serialdataflash.h</i> which are used to change operation modes of the Serial Data Flash driver
<i>pParams</i>	<i>in</i>	SERIAL_DATAFLASH_SEEK: A pointer to a new address value SERIAL_DATAFLASH_MODE_VERIFY: A pointer to a boolean value If "True", verification mode is on; if "False", verification is off

Table 6-59. Serial Data Flash *ioctl* Commands

Cmd	<i>pParams</i>	Return	Result
SERIAL_DATAFLASH_SEEK	Serial Data Flash address	None	Serial Data Flash address is adjusted to a word boundary and stored as the Serial Data Flash device address
SERIAL_DATAFLASH_MODE_VERIFY	Verification mode enabled ("True" or "False")	None	If "True", verification is on; if "False", verification is off
SERIAL_DATAFLASH_DEVICE_RESET	None	None	Resets the Serial Data Flash device driver address to 0 and disables verification mode

Description: The *ioctl* call is used to change operation modes of the Serial Data Flash driver.

Returns: The *ioctl* call returns a type *UWord16*, which is dependent on the command passed in as an input parameter. [Table 6-59](#) lists these returns based on the specific command.

Code Example: [Code Example 6-24](#) writes 0xA5C3 to the entire Data Flash, reads back, and verifies that the entire Data Flash contains 0xA5C3.

Code Example 6-24. Serial Data Flash Driver Usage

```

#include "port.h"
#include "io.h"
#include "bsp.h"
#include "fcntl.h"

#include "serialdataflash.h"

#define TST_READ_SIZE      0x4000

UWord16 ReadBuf    [TST_READ_SIZE];

/*****
int main()
{
    UWord16 ReturnSize;
    UWord16 i,j;
    types_tHandle SerialDataFlashHandle;
    UWord32 SerialDataFlashAddress = 0x0000;
    bool    VerifyMode = true;
    bool    ProtectMode = true;

    /* PREPARE DATA TO TEST*/

    for (i = 0;i < TST_READ_SIZE; i++)
    {
        ReadBuf [i] = 0xA5C3;
    }

    /* OPEN SERIAL Data Flash DEVICE */

    SerialDataFlashHandle = open(BSP_DEVICE_NAME_SERIAL_DATAFLASH_0,0);

    /*****
    /***** FILL THE ENTIRE SERIAL Data Flash WITH 0xA5C3 *****/
    /*****

    for (i=0; i< 4; i++)
    {
        for (j=0; j<16; j++)
        {
            ReturnSize = write(SerialDataFlashHandle,ReadBuf,TST_READ_SIZE);
        }
    }

    for (j=0; j<2; j++)
    {
        ReturnSize = write(SerialDataFlashHandle, ReadBuf, TST_READ_SIZE);
    }

    /*****
    /***** READ AND VERIFY THE ENTIRE SERIAL Data Flash *****/
    /***** IS FILLED WITH 0xA5C3 *****/
    /*****

```

```

SerialDataFlashAddress = 0x0000;
VerifyMode      = true;
ioctl(SerialDataFlashHandle, SERIAL_DATAFLASH_SEEK, &SerialDataFlashAddress);
ioctl(SerialDataFlashHandle, SERIAL_DATAFLASH_MODE_VERIFY, &VerifyMode);

for (i=0; i< 4; i++)
{
    for (j=0; j<16; j++)
    {
        ReturnSize = read(SerialDataFlashHandle, ReadBuf, TST_READ_SIZE);
        if (ReturnSize == 0)
        {
            /* ERROR */
        }

        else if (ReturnSize != TST_READ_SIZE)
        {
            /* ERROR */
        }
    }
}

for (j=0; j<2; j++)
{
    ReturnSize = write(SerialDataFlashHandle, ReadBuf, TST_READ_SIZE);

    if (ReturnSize == 0)
    {
        /* ERROR */
    }

    else if (ReturnSize != TST_READ_SIZE)
    {
        /* ERROR */
    }
}

/* CLOSE THE SERIAL Data Flash DEVICE */
close(SerialDataFlashHandle);

```

```

/*****

```

6.8.5 Low-Level Device Drivers I/O API Specifications

Currently, there are no low-level device driver I/O API specifications.

6.8.6 Serial Data Flash Driver Tests

The Serial Data Flash Driver Tests verify correct the operation of the Serial Data Flash driver provided by the SDK.

6.8.6.1 Serial Data Flash Driver Test

This test automatically tests the Serial Data Flash drivers by performing write, read, and verify operations on the Serial Data Flash on the DSP56852 EVM.

6.8.6.1.1 Set-up for Serial Data Flash Driver Test

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**. All four jumpers on JG6, SPI, should be shorted. This connects the pins from the Serial Data Flash to the DSP.

6.8.6.1.2 Procedure for Serial Data Flash Driver Test

- Using CodeWarrior, open:
`...\nos\bsp\test\serialdataflash\serialdataflash.mcp`
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- Once the debugger is running, the demo performs a write and a read operation to/from the Serial Data Flash and verifies the data against the Serial Data Flash
 - If verification is successful, a success message is displayed in a CodeWarrior console and the application terminates
 - If for any reason verification fails, a message indicating the failure is displayed

6.8.7 Serial Data Flash Driver Demo

Currently, there are no demos that use the Serial Data Flash driver.

6.9 TDC1 Driver

6.9.1 Introduction

This driver is designed for the Silicon Labs codec/DAA hardware located on the Telephony Daughter Card (TDC1). The TDC1 board consists of an audio codec and a DAA codec. The Silicon Labs codecs & DAA are configurable for many modes. Please refer to the Silicon Labs user's manual and the TDC1 user manual for detailed information about the board.

The TDC1 enables a developer to create voice and data telephony applications using the DSP5685xEVMs. The Telephony Daughter Card provides:

- PSTN DAA interface(s)
- High voltage isolation
- Off-hook and On-hook detection
- Caller-ID support
- Data transmission codec
- Voice transmission codec

The DSP56852EVM communicates to the TDC1 card via the SSI interface. The TDC1 driver configures the SSI to synchronize the receive and transmit channels, which means only one interrupt service routine (ISR) is needed. The TDC1 driver uses the SSI transmit interrupt to read and transmit samples to and from the TDC1. When an interrupt occurs, the SSI transmit FIFO is empty. The ISR transmits eight new samples and, since the receiver is synchronized to the transmitter, reads eight samples from the receiver. By combining the receive and transmit interrupts, less overhead is required when processing the interrupts.

The TDC1 driver provides the following features:

- Blocking and Non-Blocking modes
- Function callbacks to notify the user when data is received or when data is transmitted
- User-specified data levels to packetize the amount of data available to/from callback functions, thus minimizing processing overhead
- Static configuration parameters to establish initial TDC1 settings for gain; attenuation; mode (stereo/mono); buffer size; and user callback functions
- Dynamic control functions to change channel gain and attenuation; enable/disable the TDC1; reset TDC1 operation; and mute all sound

The following sections describe how to statically configure the TDC1 device driver via the SDK *appconfig.h* file, along with APIs for I/O and control functions.

6.9.2 Initialization

The SDK file *appconfig.h* provides the mechanism to statically configure the TDC1 driver. The *appconfig.h* file allows the programmer to set C preprocessor variables which configure the TDC1 driver default operation. Typically, these C preprocessor variables conditionally compile and link information into the application for use by the TDC1 driver during initial operation.

To include the TDC1 Driver in the application, insert the following line into the *appconfig.h* file:

```
#define INCLUDE_TDC1
```

Defining the preprocessor variable, `INCLUDE_TDC1`, directs the SDK to include the TDC1 driver in the executable image. If the preprocessor variable `INCLUDE_TDC1` is not defined, then the TDC1 driver will not be included in the executable.

You may also define additional preprocessor variables in the *appconfig.h* file to statically configure and initialize the TDC1 driver. A list of these variables is provided in [Table 6-60](#).

Table 6-60. TDC1 Configuration Items for *appconfig.h*

TDC1 #define	Explanation and Default Value
TDC1_DAA_FIFO_SIZE	Size of the TDC1 DAA FIFO in 16-bit words Default = 32
TDC1_DAA_RX_CALLBACK_LEVEL	Size (number of 16-bit samples) of the TDC1 DAA RX FIFO buffer used internally by the driver for the RX channel. The length of the TDC1 DAA RX FIFO buffer determines the maximum number of TDC1 input samples that can be buffered within the driver. Default = 8
TDC1_DAA_RX_CALLBACK	A user function for the TDC1 driver to call when it has received at least TDC1_DAA_RX_CALLBACK_LEVEL samples in its internal TDC1 DAA RX buffer. The user function must be of type <i>void</i> : <pre>typedef void (*tdc1_tCallback) (void *pCallbackArg, int MaxNBytes);</pre> The parameters to the callback function are: <i>pCallbackArg</i> - This is a pointer to the user's callback argument <i>MaxNBytes</i> - The number of 8-bit bytes currently in the driver's TDC1 DAA RX buffers; the user application may then <i>read</i> up to <i>MaxNBytes</i> bytes. Note that the maximum number of 16-bit samples is <i>MaxNBytes</i>/2 Default = NULL
TDC1_DAA_RX_CALLBACK_ARG	A value that the user would like passed to the TDC1_DAA_RX_CALLBACK function. This value can be anything desired by the user. It is typically defined as a pointer to a device specific data structure or an index into an application table. This argument is most useful for sharing code between multiple instances of a device, channel, or data structure. The driver does nothing with this value other than pass it back to the user application as a parameter when calling the TDC1_DAA_RX_CALLBACK function. Default = NULL

Table 6-60. TDC1 Configuration Items for *appconfig.h* (Continued)

TDC1 #define	Explanation and Default Value
TDC1_DAA_TX_CALLBACK_LEVEL	Size (number of 16-bit samples) of the TDC1 DAA TX FIFO buffer used internally by the driver for the TX channel. The length of the TDC1 DAA TX FIFO buffer determines the maximum number of TDC1 output samples that can be buffered within the driver. Default = 2
TDC1_DAA_TX_CALLBACK	A user function for the TDC1 driver to call when it has received at least TDC1_DAA_TX_CALLBACK_LEVEL samples in its internal TDC1 DAA TX buffer. The user function must be of type <i>void</i>: <pre>typedef void (*tdc1_tCallback) (void *pCallbackArg, int MaxNBytes);</pre> The parameters to the callback function are: <i>pCallbackArg</i> - This is a pointer to the user's callback argument <i>MaxNBytes</i> - The number of 8-bit bytes currently in the driver's TDC1 DAA TX buffers; the user application may then <i>write</i> up to <i>MaxNBytes</i> bytes. Note that the maximum number of 16-bit samples is <i>MaxNBytes</i>/2 Default = NULL
TDC1_DAA_TX_CALLBACK_ARG	A value that the user would like passed to the TDC1_DAA_TX_CALLBACK function. This value can be anything desired by the user. It is typically defined as a pointer to a device specific data structure or an index into an application table. This argument is most useful for sharing code between multiple instances of a device, channel, or data structure. The driver does nothing with this value other than pass it back to the user application as a parameter when calling the TDC1_DAA_TX_CALLBACK function. Default = NULL
TDC1_CODEC_FIFO_SIZE	Size of the TDC1 codec FIFO in 16-bit words Default = 32
TDC1_CODEC_RX_CALLBACK_LEVEL	Size (number of 16-bit samples) of the TDC1 RX FIFO buffer used internally by the driver for the RX channel. The length of the TDC1 RX FIFO buffer determines the maximum number of TDC1 output samples that can be buffered within the driver. Default = 8

Table 6-60. TDC1 Configuration Items for *appconfig.h* (Continued)

TDC1 #define	Explanation and Default Value
TDC1_CODEC_RX_CALLBACK	A user function for the TDC1 codec driver to call when it has received at least TDC1_CODEC_RX_CALLBACK_LEVEL samples in its internal RX buffer. The user function must be of type <i>void</i>: <pre>typedef void (*tdc1_tCallback) (void *pCallbackArg, int MaxNBytes);</pre> The parameters to the callback function are: <i>pCallbackArg</i> - This is a pointer to the user's callback argument <i>MaxNBytes</i> - The number of 8-bit bytes currently in the driver's TDC1 CODEC RX buffers; the user application may then <i>read</i> up to <i>MaxNBytes</i> bytes. Note that the maximum number of 16-bit samples is <i>MaxNBytes/2</i> Default = NULL
TDC1_CODEC_RX_CALLBACK_ARG	A value that the user would like passed to the TDC1_CODEC_RX_CALLBACK function. This value can be anything desired by the user. It is typically defined as a pointer to a device specific data structure or an index into an application table. This argument is most useful for sharing code between multiple instances of a device, channel, or data structure. The driver does nothing with this value other than pass it back to the user application as a parameter when calling the TDC1_CODEC_RX_CALLBACK function. Default = NULL
TDC1_CODEC_TX_CALLBACK_LEVEL	Size (number of 16-bit samples) of the TDC1 CODEC TX FIFO buffer used internally by the driver for the TX channel. The length of the TDC1 CODEC TX FIFO buffer determines the maximum number of TDC1 output samples that can be buffered within the driver. Default = 2

Table 6-60. TDC1 Configuration Items for *appconfig.h* (Continued)

TDC1 #define	Explanation and Default Value
TDC1_CODEC_TX_CALLBACK	A user function for the codec driver to call when it has received at least TDC1_CODEC_TX_CALLBACK_LEVEL samples in its internal TDC1 CODEC TX buffer. The user function must be of type <i>void</i>: <pre>typedef void (*tdc1_tCallback) (void *pCallbackArg, int MaxNBytes);</pre> The parameters to the callback function are: <i>pCallbackArg</i> - This is a pointer to the user's callback argument <i>MaxNBytes</i> - The number of 8-bit bytes currently in the driver's RX buffers; the user application may then <i>read</i> up to <i>MaxNBytes</i> bytes. Note that the maximum number of 16-bit samples is <i>MaxNBytes/2</i> Default = NULL
TDC1_CODEC_TX_CALLBACK_ARG	A value that the user would like passed to the TDC1_CODEC_TX_CALLBACK function. This value can be anything desired by the user. It is typically defined as a pointer to a device specific data structure or an index into an application table. This argument is most useful for sharing code between multiple instances of a device, channel, or data structure. The driver does nothing with this value other than pass it back to the user application as a parameter when calling the TDC1_CODEC_TX_CALLBACK function. Default = NULL
TDC1_3000_LINE_IN_GAIN	TDC1 Audio Codec Line Input Gain Value Possible values are 0db, 10db and 20db: TDC1_3000_LINE_IN_20DB TDC1_3000_LINE_IN_10DB TDC1_3000_LINE_IN_0DB Default = TDC1_3000_LINE_IN_0DB
TDC1_3000_LINE_IN_MUTE_STATE	TDC1 Audio Codec Line In Mute State Possible values are: TDC1_3000_LINE_IN_MUTE and TDC1_3000_LINE_IN_NOT_MUTE Default = TDC1_LINE_IN_NOT_MUTE

Table 6-60. TDC1 Configuration Items for *appconfig.h* (Continued)

TDC1 #define	Explanation and Default Value
TDC1_3000_MIC_GAIN	TDC1 Audio Codec Microphone Gain value Possible values are 0db, 10db, 20db and 30db: TDC1_3000_MIC_30DB TDC1_3000_MIC_20DB TDC1_3000_MIC_10DB TDC1_3000_MIC_0DB Default = TDC1_3000_MIC_0DB
TDC1_3000_MIC_MUTE_STATE	TDC1 Audio Codec Microphone Mute Possible values are: TDC1_3000_MIC_MUTE TDC1_3000_MIC_NOT_MUTE Default = TDC1_3000_MIC_MUTE
TDC1_3000_HNDST_MUTE_STATE	TDC1 Audio Codec Handset Mute Possible values are: TDC1_3000_HNDST_MUTE TDC1_3000_HNDST_NOT_MUTE Default = TDC1_3000_HNDST_MUTE
TDC1_3000_ADC_GAIN_VALUE	TDC1 Audio ADC Gain The gain can be adjusted from -34.5db to 12db with 1.5db per step by using the macro "TDC1_3000_GAIN(Gain)", where "Gain" is a value in dB, or by using the macro "TDC1_CODEEC_GAIN_FROM_PERCENT(x)", where "x" is a value from 0-100. Default = 0db
TDC1_3000_LINE_OUT_MUTE_STATE	TDC1 Audio Codec Line Out Mute State Possible values are: TDC1_3000_LINE_OUT_MUTE_ENABLE TDC1_3000_LINE_OUT_MUTE_DISABLE Default = TDC1_3000_LINE_OUT_MUTE_ENABLE
TDC1_3000_HNDST_OUT_MUTE_STATE	TDC1 Audio Codec Handset Out Mute State Possible values are: TDC1_HNDST_OUT_MUTE_ENABLE TDC1_HNDST_OUT_MUTE_DISABLE Default = TDC1_HNDST_OUT_MUTE_DISABLE

Table 6-60. TDC1 Configuration Items for *appconfig.h* (Continued)

TDC1 #define	Explanation and Default Value
TDC1_3000_DAC_GAIN_VALUE	TDC1 Audio DAC Gain The gain can be adjusted from -34.5db to 12db with 1.5db per step using the macro "TDC1_3000_GAIN(Gain)", where "Gain" is a value in dB, or by using the macro "TDC1_CODEC_GAIN_FROM_PERCENT(x)", where "x" is a value from 0-100. Default = 0db
TDC1_3000_SPKR_LEFT_MUTE_STATE	TDC1 Audio Codec Speaker Left Mute State Possible values are: TDC1_3000_SPKR_LEFT_MUTE_ENABLE TDC1_3000_SPKR_LEFT_MUTE_DISABLE Default = TDC1_3000_SPKR_LEFT_MUTE_ENABLE
TDC1_3000_SPKR_RIGHT_MUTE_STATE	TDC1 Audio Codec Speaker Left Mute State Possible values are: TDC1_3000_SPKR_RIGHT_MUTE_ENABLE TDC1_3000_SPKR_RIGHT_MUTE_DISABLE Default = TDC1_3000_SPKR_LEFT_MUTE_ENABLE
TDC1_3000_LINE_OUT_ATT	TDC1 Audio Codec Line Out Attenuation Possible values are: TDC1_3000_LINE_OUT_ATT_0DB TDC1_3000_LINE_OUT_ATT_6DB TDC1_3000_LINE_OUT_ATT_12DB TDC1_3000_LINE_OUT_ATT_18DB Default = TDC1_3000_LINE_OUT_ATT_0DB
TDC1_3000_SPK_OUT_ATT	TDC1 Audio Codec Speaker Out Attenuation Possible values are: TDC1_3000_SPKR_OUT_ATT_0DB TDC1_3000_SPKR_OUT_ATT_6DB TDC1_3000_SPKR_OUT_ATT_12DB TDC1_3000_SPKR_OUT_ATT_18DB Default = TDC1_3000_SPKR_OUT_ATT_0DB
TDC1_3021_AOUT_TX_ATT	TDC1 DAA Codec TX Attenuation Possible values are: TDC1_3021_AOUT_TX_20DB_ATT TDC1_3021_AOUT_TX_26DB_ATT TDC1_3021_AOUT_TX_32DB_ATT TDC1_3021_AOUT_TX_MUTE Default = TDC1_3021_AOUT_TX_20DB_ATT

Table 6-60. TDC1 Configuration Items for *appconfig.h* (Continued)

TDC1 #define	Explanation and Default Value
TDC1_3021_AOUT_RX_ATT	TDC1 DAA Codec RX Attenuation Possible values are: TDC1_3021_AOUT_RX_20DB_ATT TDC1_3021_AOUT_RX_26DB_ATT TDC1_3021_AOUT_RX_32DB_ATT TDC1_3021_AOUT_RX_MUTE Default = TDC1_3021_AOUT_RX_20DB_ATT
TDC1_DAA_N2_M2_DIVIDE_REG	TDC1 DAA N2 M2 divider Possible values are: TDC1_3021_SAMPLE_AT_7200 TDC1_3021_SAMPLE_AT_8000 TDC1_3021_SAMPLE_AT_8229 TDC1_3021_SAMPLE_AT_8400 TDC1_3021_SAMPLE_AT_9000 TDC1_3021_SAMPLE_AT_9600 TDC1_3021_SAMPLE_AT_10286 Default =TDC1_3021_SAMPLE_AT_7200
TDC1_DAA_CGM_DIVIDE_REG	TDC1 DAA CGM Divide Register Possible values are: 0 1 Default = 1
TDC1_3021_TX_MUTE_STATE	TDC1 DAA TX Mute State Possible values are: TDC1_TX_MUTE TDC1_TX_NOT_MUTE Default = TDC1_TX_NOT_MUTE
TDC1_3021_RX_MUTE_STATE	TDC1 DAA RX Mute State Possible values are: TDC1_RX_MUTE TDC1_RX_NOT_MUTE Default = TDC1_RX_NOT_MUTE
TDC1_3021_TX_ATT_VALUE	TDC1 DAA TX Attenuation Possible values are: TDC1_3021_TX_ATT_0DB TDC1_3021_TX_ATT_3DB TDC1_3021_TX_ATT_6DB TDC1_3021_TX_ATT_9DB TDC1_3021_TX_ATT_12DB Default = TDC1_3021_TX_ATT_0DB

ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005
Table 6-60. TDC1 Configuration Items for *appconfig.h* (Continued)

TDC1 #define	Explanation and Default Value
TDC1_3021_RX_GAIN_VALUE	TDC1 DAA RX Gain Value Possible values are: TDC1_3021_RX_GAIN_0DB TDC1_3021_RX_GAIN_3DB TDC1_3021_RX_GAIN_6DB TDC1_3021_RX_GAIN_9DB TDC1_3021_RX_GAIN_12DB Default = TDC1_3021_RX_GAIN_0DB

Code Example: [Code Example 6-25](#) configures and initializes the TDC1 driver.

Code Example 6-25. Sample *appconfig.h* code

```

#define INCLUDE_TDC1          /* TDC1 driver */
#define INCLUDE_IO            /* I/O support */
#define INCLUDE_BSP          /* BSP support */

extern void Tdc1DaaRXISR (void *pCallbackArg, int MaxNum);
extern void Tdc1DaaTXISR (void *pCallbackArg, int MaxNum);
extern void Tdc1CodecRXISR (void *pCallbackArg, int MaxNum);
extern void Tdc1CodecTXISR (void *pCallbackArg, int MaxNum);

#define FIFO_SIZE              32
#define RX_FULL_SIZE           ((int)((3*FIFO_SIZE)/4))
#define TX_EMPTY_SIZE          ((int)(FIFO_SIZE/4))
#define TDC1_DAA_FIFO_SIZE     FIFO_SIZE

#define TDC1_DAA_FIFO_SIZE     FIFO_SIZE

#define TDC1_DAA_RX_CALLBACK    Tdc1DaaRXISR
#define TDC1_DAA_RX_CALLBACK_LEVEL RX_FULL_SIZE
#define TDC1_DAA_RX_CALLBACK_ARG NULL

#define TDC1_DAA_TX_CALLBACK    Tdc1DaaTXISR
#define TDC1_DAA_TX_CALLBACK_LEVEL TX_EMPTY_SIZE
#define TDC1_DAA_TX_CALLBACK_ARG NULL

#define TDC1_CODEC_FIFO_SIZE    FIFO_SIZE

#define TDC1_CODEC_RX_CALLBACK  Tdc1CodecRXISR
#define TDC1_CODEC_RX_CALLBACK_LEVEL RX_FULL_SIZE
#define TDC1_CODEC_RX_CALLBACK_ARG NULL

#define TDC1_CODEC_TX_CALLBACK  Tdc1CodecTXISR
#define TDC1_CODEC_TX_CALLBACK_LEVEL TX_EMPTY_SIZE
#define TDC1_CODEC_TX_CALLBACK_ARG NULL

```

6.9.3 API Definition

This section defines the Application Programming Interface (API) by identifying all public interface functions and data structures.

The TDC1 interface can be used at either of two levels: a device-independent layer interface and a device-dependent driver interface. The device-independent layer interface invokes the device-dependent TDC1 driver interface.

The device-independent API to the TDC1 interface is standard and more portable than the device-dependent TDC1 interface, but less efficient. The device-dependent TDC1 driver provides a non-standard interface that is more efficient than the device-independent calls, but less portable.

Your application may use either the device-independent interface or the device-dependent driver interface to the TDC1 driver, depending on your specific goals for efficiency and portability. However, you should not mix the TDC1 driver APIs in your application. In particular, you **must not** use the File Descriptor returned by the device-independent *open* function with any device-dependent TDC driver function calls, nor may you use the File Descriptor returned by the device-dependent driver *tdc1Open* call with any device-independent interface function call.

The following information may be found in the public header file *tdc1.h*. You must include *tdc1.h* in your application to utilize the driver TDC1 API.

Public Interface Function(s):

```

/*****
* The device-independent interface defines functions as follows:
*   types_tHandle open(const char *pName, int OFlags);
*   ssize_t read(types_tHandle FileDesc, void * pBuffer, size_t NBytes);
*   ssize_t write(types_tHandle FileDesc, const void * pBuffer, size_t NBytes);
*   UWord16 ioctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams);
*   int      close(types_tHandle FileDesc);
*
* The low level TDC1 driver interface defines functions as follows:
*   types_tHandle tdc1Open(const char *pName, int OFlags);
*   ssize_t tdc1Read(types_tHandle FileDesc, void * pBuffer, size_t NBytes);
*   ssize_t tdc1Write(types_tHandle FileDesc, const void * pBuffer, size_t NBytes);
*   UWord16 tdc1Ioctl(types_tHandle FileDesc, UWord16 Cmd, tdc1_sParams * pParams,
*                     const char *pName);
*   int tdc1Close (types_tHandle FileDesc);
*
*****/

```

Public Data Structure(s):

None

Members:

None

6.9.4 Device-Independent API Specification

The following pages specify the Application Programming Interface (API) for the device-independent interface to the TDC1 driver.

To use the device-independent interface to the TDC1 driver, you must define both the `INCLUDE_TDC1` variable and the `INCLUDE_IO` variable in the `appconfig.h` file of the SDK Embedded Project you created in CodeWarrior:

```
#define INCLUDE_IO
#define INCLUDE_TDC1
```

The user's application code then uses the TDC1 driver API as follows:

- An *open* call is made to initialize the TDC1 device
- *ioctl* calls are made to control the TDC1 device
- *write* calls are made to write data to the TDC1 (transmit)
- *read* calls are made to read data from the TDC1 (receive)
- After all TDC1 operations are completed, the TDC1 driver is closed via a *close* call

The following sections describe each function of the device-independent API for the TDC1 driver.

In the following specifications, function arguments for each routine are described as *in*, *out*, or *inout*. An *in* argument means that the parameter value is an input only to the function. An *out* argument means that the parameter value is an output only from the function. 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.

The *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.

6.9.4.1 *open*

Call(s):

```
types_tHandle open(const char *pName, int OFlags);
```

Arguments:

Table 6-61. TDC1 Driver Arguments - *open*

<i>pName</i>	<i>in</i>	The TDC1 device name; see the <i>bsp.h</i> file for target-specific values, such as: BSP_DEVICE_NAME_TDC1_DAA_0 BSP_DEVICE_NAME_TDC1_CODEC_0
<i>OFlag</i>	<i>in</i>	Open Mode Flags Valid values are: O_RDWR O_NONBLOCKING O_BLOCK (Default)
<i>FileDesc</i>	<i>out</i>	TDC1 device descriptor returned by <i>open</i> call and used in subsequent <i>read/write/close/ioctl</i> function calls

Description: The *open* call opens the TDC1 driver for operations and returns a file descriptor used by subsequent *read/write/close/ioctl* function calls. The TDC1 device must be opened before any calls are made to other TDC1 Driver functions.

Return: The *open* call returns a type *types_tHandle*. When the *open* call is successful, this *types_tHandle* is a file descriptor (*FileDesc*) for the TDC1 device and is used by the TDC1 driver's *read*, *write*, *close*, and *ioctl* functions. A "-1" is returned when the *open* call is not successful.

Code Example: See [Code Example 6-26](#).

6.9.4.2 write

Call(s):

```
ssize_t write(types_tHandle FileDesc, const void * pBuffer, size_t NBytes);
```

Arguments:

Table 6-62. TDC1 Driver Arguments - write

<i>FileDesc</i>	<i>in</i>	TDC1 device descriptor returned by the <i>open</i> call
<i>pBuffer</i>	<i>in</i>	Pointer to user buffer of at least <i>NBytes</i> bytes
<i>NBytes</i>	<i>in</i>	Number of 8-bit bytes to write to the TDC1; note that the number of 16-bit samples is <i>NBytes</i> /2

Description: This function is mapped to *tdc1Write()* via the *tdc1drvIOInterfaceVT* structure. Please see the description of *tdc1Write()* for a description of this function.

Returns: The *write* call returns a value of type *ssize_t*. This return value is the actual number of 8-bit bytes transferred. Note that the number of 16-bit samples written is half the return value.

Code Example: See [Code Example 6-26](#).



6.9.4.3 read

Call(s):

```
ssize_t read(types_tHandle FileDesc, void * pBuffer, size_t NBytes);
```

Arguments:

Table 6-63. TDC1 Driver Arguments - read

<i>FileDesc</i>	<i>in</i>	TDC1 device descriptor returned by the <i>open</i> call
<i>pBuffer</i>	<i>inout</i>	Pointer to a user buffer of at least <i>NBytes</i> bytes
<i>NBytes</i>	<i>in</i>	Number of 8-bit bytes to read; note that the number of 16-bit samples is <i>NBytes/2</i>

Description: This function is mapped to *tdc1Read()* via the *tdc1drvIOInterfaceVT* structure. Please see the description of *tdc1Read()* for a description of this function.

Returns: The *read* call returns a value of type *ssize_t*. This return value is the actual number of 8-bit bytes transferred. Note that the number of 16-bit samples read is half the return value.

Code Example: See [Code Example 6-26](#).

6.9.4.4 *close*

Call(s):

```
int close (types_tHandle FileDesc);
```

Arguments:

Table 6-64. TDC1 Driver Arguments - *close*

<i>FileDesc</i>	<i>in</i>	TDC1 device descriptor returned by <i>open</i> call

Description: This function is mapped to *tdc1Close()* via the *tdc1drvIOInterfaceVT* structure. Please see the description of *tdc1Close()* for a description of this function.

Returns: The *close* call returns a type *int*; it returns “0” if successful and “-1” if unsuccessful.

Code Example: See [Code Example 6-26](#).

6.9.4.5 *ioctl*

Call(s):

```
UWord16 ioctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams);
```

Arguments:

Table 6-65. TDC1 Driver Arguments - *ioctl*

<i>FileDesc</i>	<i>in</i>	TDC1 device descriptor returned by <i>open</i> call
<i>Cmd</i>	<i>in</i>	Commands found in <i>tdc1.h</i> which are used to manage the TDC1 driver
<i>pParams</i>	<i>in</i>	<i>ioctl</i> command parameter

For a list of commands, see [Table 6-71](#) which contains the list of commands and macros for the *tdc1ioctl* function, identical to those for the *ioctl* function.

Macros: The macro `TDC1_CODEC_GAIN_FROM_PERCENT(x)`, where $0 \leq x \leq 100$, can be used as a portable method to calculate a target-specific gain value from the percentage “x”. A value of 0% returns the maximum amount of attenuation available from the TDC1 codec; a value of 100% returns the maximum amount of gain available from the TDC1 codec.

The macro `TDC1_DAA_TX_GAIN_FROM_PERCENT(x)`, where $0 \leq x \leq 100$, can be used as a portable method to calculate a target-specific TX gain value from the percentage “x”. A value of 0% returns the maximum amount of denotation available from the TDC1 DAA codec transmitter; a value of 100% returns the maximum amount of gain available from the TDC1 DAA codec transmitter.

The macro `TDC1_DAA_RX_GAIN_FROM_PERCENT(x)`, where $0 \leq x \leq 100$, can be used as a portable method to calculate a target-specific TX gain value from the percentage “x”. A value of 0% returns the maximum amount of denotation available from the TDC1 DAA codec receiver; a value of 100% returns the maximum amount of gain available from the TDC1 DAA codec receiver.

The macro `TDC1_3000_GAIN(Gain)` can be used to input gain values in dB. The macro accepts gain values in db from -34.5db to 12db.

The macro `TDC1_3021_TX_GAIN(Gain)` can be used to input gain values in db. This macro accepts gain values in db from 0db to 12db.

The macro `TDC1_3021_RX_GAIN(Gain)` can be used to input gain values in db. This macro accepts gain values in db from 0db to 12db.

Description: The *ioctl* call provides various control functions for the TDC1. See the list of commands in [Table 6-71](#) for specific control functions.

Returns: The *ioctl* call returns a value dependent on the command passed in as an input parameter. [Table 6-71](#) lists these returns based on the specific command.

Code Example: [Code Example 6-27](#) reads and writes eight codec samples (16 bytes) at a time, using the device-independent API to the TDC1 driver in Non-Blocking mode.

Code Example 6-26. TDC1 Driver Device-Independent API Example

```

#include "bsp.h"
#include "tdc1.h"
#include "led.h"

#define NUM_SAMPLES    RX_FULL_SIZE
UWord16 Status;
volatile types_tHandle Tdc1Daa, Tdc1Codec;
volatile UWord16 Tdc1DaaBufferFull, Tdc1DaaBufferEmpty;
volatile UWord16 Tdc1CodecBufferFull, Tdc1CodecBufferEmpty;
volatile UWord16  LoopCount = 0;
volatile UWord16  LedCount = 0;
UWord16 pDaaSamples[FIFO_SIZE];
UWord16 pCodecSamples[FIFO_SIZE];

/*
    Forward declarations
*/
void Tdc1DaaRXISR (void *pCallbackArg, int MaxNBytes);
void Tdc1DaaTXISR (void *pCallbackArg, int MaxNBytes);
void Tdc1CodecRXISR (void *pCallbackArg, int MaxNBytes);
void Tdc1CodecTXISR (void *pCallbackArg, int MaxNBytes);

/*****
void Tdc1DaaRXISR (void *pCallbackArg, int MaxNBytes)
{
    Tdc1DaaBufferFull = true;
    LoopCount++;
    LedCount += MaxNBytes/2;
}

/*****
void Tdc1DaaTXISR (void *pCallbackArg, int MaxNBytes)
{
    Tdc1DaaBufferEmpty = true;
}

/*****
void Tdc1CodecRXISR (void *pCallbackArg, int MaxNBytes)
{
    Tdc1CodecBufferFull = true;
}

/*****
void Tdc1CodecTXISR (void *pCallbackArg, int MaxNBytes)
{
    Tdc1CodecBufferEmpty = true;
}

/*****
void main(void)
{
    types_tHandle Led;
    Word16 samples;
    Word16 i,j;
    int samp = 0;

```



```

bool OffHook = true;
tdc1_sRegister Register;
UWord16 rate[] = {TDC1_3021_SAMPLE_AT_7200, TDC1_3021_SAMPLE_AT_8000,
                  TDC1_3021_SAMPLE_AT_8229, TDC1_3021_SAMPLE_AT_8400,
                  TDC1_3021_SAMPLE_AT_9000, TDC1_3021_SAMPLE_AT_9600,
                  TDC1_3021_SAMPLE_AT_10286};

Tdc1DaaBufferFull = false;
Tdc1DaaBufferEmpty = false;
Tdc1CodecBufferFull = false;
Tdc1CodecBufferEmpty = false;

/* Set up LEDs*/
Led = open(BSP_DEVICE_NAME_LED_0, 0);

/* Initialize the buffer */
for (i=0; i<FIFO_SIZE; i++) {
    pDaaSamples[i] = 0;
    pCodecSamples[i] = 0;
}

/* Set up Tdc1 */

/* open both the DAA and the codec */
Tdc1Daa = open(BSP_DEVICE_NAME_TDC1_DAA_0, O_RDWR | O_NONBLOCK);
Tdc1Codec = open(BSP_DEVICE_NAME_TDC1_CODEC_0, O_RDWR | O_NONBLOCK);

/* Set the si3000 codec's speaker drive and line drive */
/* to normal operation */
Register.Register = 1;
Register.Data = 0x18;
ioctl(Tdc1Codec, TDC1_DEVICE_WRITE_REG, &Register);

/* Unmute the si3000 codec's line out */
/* The while loop is too allow the previous write to finish before */
/* issuing another write. If the previous write is not done the */
/* function will simply return "false" and not issue the write */
/* request. */

while (!ioctl(Tdc1Codec, TDC1_DEVICE_MUTE_LINE_OUT, false));

/* Set the si3000 codec's Rx gain to 0dB and unmute the */
/* left and right speaker */
while (!ioctl(Tdc1Codec, TDC1_DEVICE_MUTE_SPEAKERS, false));
while (!ioctl(Tdc1Codec, TDC1_DEVICE_SET_RX_GAIN, TDC1_3000_GAIN(0)));

/* The following sequence of control commands is not really */
/* necessary, but is provided as an example. */
ioctl(Tdc1Codec, TDC1_DEVICE_DISABLE, NULL);
while (!ioctl(Tdc1Codec, TDC1_DEVICE_MUTE_LINE_IN, false));

```

ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

```

while (!ioctl (Tdc1Codec, TDC1_DEVICE_SET_RX_GAIN,
               TDC1_CODEC_GAIN_FROM_PERCENT(75)));
while (!ioctl (Tdc1Codec, TDC1_DEVICE_SET_TX_GAIN,
               TDC1_CODEC_GAIN_FROM_PERCENT(75)));
while (!ioctl (Tdc1Daa, TDC1_DEVICE_SET_RX_GAIN,
               TDC1_DAA_RX_GAIN_FROM_PERCENT(75)));
while (!ioctl (Tdc1Daa, TDC1_DEVICE_SET_TX_GAIN,
               TDC1_DAA_TX_GAIN_FROM_PERCENT(75)));
ioctl (Tdc1Daa, TDC1_DEVICE_SET_RX_CALLBACK_LEVEL, 32);
ioctl (Tdc1Daa, TDC1_DEVICE_SET_TX_CALLBACK_LEVEL, 0);

/* The following Enable command actually starts tdc1 data transfer */
ioctl (Tdc1Daa, TDC1_DEVICE_ENABLE, NULL);
ioctl (Tdc1Codec, TDC1_DEVICE_ENABLE, NULL);

/* The executive loop just toggles the LED depending on the number of */
/* Samples received from the Tdc1Daa. Loopback is done in the ISRs */
/* or below depending on the state of "USE_CALLBACK" in the */
/* appconfig.h file. */

ioctl(Led, LED_ON, LED_GREEN2);
while(1)
{
    if (Tdc1DaaBufferFull) {
        /* Read speech samples from the tdc1. Returns number of */
        /* bytes read */

        samples = read(Tdc1Daa, (void*)pDaaSamples,
                      (NUM_SAMPLES<<1));
        Tdc1DaaBufferFull = false;
    }

    if (Tdc1DaaBufferEmpty) {
        /* Write speech samples to the tdc1. Returns number of */
        /* bytes written */
        samples = write(Tdc1Daa, (void*)pDaaSamples,
                      (NUM_SAMPLES<<1));
        Tdc1DaaBufferEmpty = false;
    }

    if (Tdc1CodecBufferFull) {
        /* Read speech samples from the tdc1_2. Returns number */
        /* of bytes read */
        samples = read(Tdc1Codec, (void*)pCodecSamples,
                      (NUM_SAMPLES<<1));
        Tdc1CodecBufferFull = false;
    }

    if (Tdc1CodecBufferEmpty) {
        /* Write speech samples to the tdc1_2. Returns number */
        /* of bytes written */
        samples = write(Tdc1Codec, (void*)pCodecSamples,
                      (NUM_SAMPLES<<1));
        Tdc1CodecBufferEmpty = false;
    }
}

```

```

if (LedCount > 7200) {
    ioctl(Led, LED_TOGGLE, LED_GREEN2);
    LedCount = 0;
}

if (LoopCount > 2250)
{ /* approximately every 10 seconds at 7200 samples/s */

    /* Read the DAA's sample rate register */
    Register.Register = 9;
    Register.Data = 0;
    while(!ioctl(Tdc1Daa, TDC1_DEVICE_READ_REG, &Register));
    while(!Register.bDataValid);
    Status = Register.Data;

    /* Verify that the DAA's register contents are correct */
    if (Status != rate[samp]) {
        asm(nop);
        asm(nop);
    }

    /* Read the Codec's DAC volume control register */
    Register.Register = 7;
    Register.Data = 0;
    while(!ioctl(Tdc1Codec, TDC1_DEVICE_READ_REG, &Register));
    while(!Register.bDataValid);
    Status = Register.Data;

    /* Verify that the Codec's register contents are correct */
    if (Status != 0x5f) {
        asm(nop);
        asm(nop);
    }

    /* Read the Codec's status register */
    Register.Register = 8;
    Register.Data = 0;
    while(!ioctl(Tdc1Codec, TDC1_DEVICE_READ_REG, &Register));
    while(!Register.bDataValid);
    Status = Register.Data;

    /* Verify that the Codec's status register does not */
    /* indicate an error */
    if (Status != 0x00) {
        asm(nop);
        asm(nop);
    }

    /* Toggle the Off-Hook bit */
    while (!ioctl(Tdc1Daa, TDC1_DEVICE_OFF_HOOK, OffHook));
    if (OffHook) {
        OffHook = false;
    }
    else {
        OffHook = true;
    }
}

```


ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

```

        /* Increment to the next sample rate and load the DAA's */
        /* register */
        samp++;
        if (samp > 6) {
            samp = 0;
        }
        while (!ioctl(Tdc1Daa, TDC1_DEVICE_SET_SAMPLE_RATE,
            rate[samp]));

        /* Read the Codec's status register */
        Register.Register = 8;
        Register.Data = 0;
        while(!ioctl(Tdc1Codec, TDC1_DEVICE_READ_REG, &Register));
        while(!Register.bDataValid);
        Status = Register.Data;

        LoopCount = 0;
    }
}

/* Execution should never actually reach here */

ioctl(Led, LED_OFF, LED_GREEN2);

close(Led);
close(Tdc1Codec);
close(Tdc1Daa);
}

```

6.9.5 Device-Dependent TDC1 Driver API

The following pages specify the Application Programming Interface (API) for the device-dependent interface to the TDC1 driver.

To use the device-dependent interface to the TDC1 driver, you must define the `INCLUDE_TDC1` variable in the `appconfig.h` file of the SDK Embedded Project you created in CodeWarrior:

```
#define INCLUDE_TDC1
```

The user's application code then uses the TDC1 driver API as follows:

- *tdc1Open* call is made to initialize the TDC1 device
- *tdc1Ioctl* calls are made to control the TDC1 device
- *tdc1Write* calls are made to write data to the TDC1 (transmit)
- *tdc1Read* calls are made to read data from the TDC1 (receive)
- After all TDC1 operations are completed, the TDC1 driver is closed via a *tdc1Close* call

The following sections describe each function of the device-dependent API for the TDC1 driver.

In the following specifications, function arguments for each routine are described as *in*, *out*, or *inout*. An *in* argument means that the parameter value is an input only to the function. An *out* argument means that the parameter value is an output only from the function. 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.

The *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.

6.9.5.1 tdc1Open

Call(s):

```
types_tHandle tdc1Open(const char *pName, int OFlags);
```

Arguments:

Table 6-66. TDC1 Driver Arguments - *tdc1Open*

<i>pName</i>	<i>in</i>	The TDC1 device name; see the <i>bsp.h</i> file for target specific values, such as: BSP_DEVICE_NAME_TDC1_DAA_0 BSP_DEVICE_NAME_TDC1_CODEC_0
<i>OFlag</i>	<i>in</i>	Open Mode Flags Valid values are: O_RDWR O_NONBLOCKING O_BLOCK (Default)
<i>FileDesc</i>	<i>out</i>	TDC1 device descriptor returned by <i>tdc1Open</i> call and used in subsequent <i>tdc1Read/tdc1Write/tdc1Close/tdc1Ioctl</i> function calls

Description: The *tdc1Open* call opens the TDC1 driver for operations and returns a file descriptor used by subsequent *tdc1Read/tdc1Write/tdc1Close/tdc1Ioctl* function calls. The TDC1 device must be opened before any calls are made to other TDC1 driver functions.

Return: The *tdc1Open* call returns a type *types_tHandle*. When the *tdc1Open* call is successful, this *types_tHandle* is a file descriptor (*FileDesc*) for the TDC1 device and is used by the TDC driver's *tdc1Read*, *tdc1Write*, *tdc1Close*, and *tdc1Ioctl* functions. A "-1" is returned when the *tdc1Open* call is not successful.

Code Example: See [Code Example 6-27](#).

6.9.5.2 *tdc1Write*

Call(s):

```
ssize_t tdc1Write(types_tHandle FileDesc, const void * pBuffer, size_t NBytes);
```

Arguments:

Table 6-67. TDC1 Driver Arguments - *tdc1Write*

<i>FileDesc</i>	<i>in</i>	TDC1 device descriptor returned by the <i>tdc1Open</i> call
<i>pBuffer</i>	<i>in</i>	Pointer to user buffer of at least <i>NBytes</i> bytes
<i>NBytes</i>	<i>in</i>	Number of 8-bit bytes to write to the TDC1; note that the number of 16-bit samples is <i>NBytes</i> /2

Description: The *tdc1Write* call writes the data from the user buffer (*pBuffer*) to the TDC1 driver's internal data structures. User data is 16-bit signed integers representing linear PCM values. Note that the input parameter *NBytes* specifies the size of the data to write as a number of 8-bit bytes.

The actual number of data bytes written will be equal to *NBytes* if the driver is operating in Blocking mode. In Blocking mode, the TDC1 Driver will not return from the *tdc1Write* function until *NBytes* bytes from the user's buffer are transferred to the driver's internal data structures.

However, the number of data bytes written may be less than or equal to *NBytes* if operating in Non-Blocking mode. In Non-Blocking mode, the TDC1 driver will return immediately from the *write* function after transferring as much of the user's data as possible into the driver's internal data structures.

The *tdc1Write* function may be called from normal application code, from an ISR, or from the user's callback function, which is called in turn from the TDC1 Driver's ISR. However, avoid calling *tdc1Write* in Blocking mode from an ISR or from a callback function with more data than the TDC1 driver can immediately transfer (i.e., more than the *MaxNBytes* parameter of the callback function); otherwise, deadlock may occur if TDC1 interrupts are disabled during the *tdc1Write*.

Avoid using *tdc1Write* in both normal application code and from an ISR or callback function. The asynchronous nature of the ISR and callback may lead to re-entrancy issues if you call *tdc1Write* from normal application code as well.

Returns: The *tdc1Write* call returns a value of type *ssize_t*. This return value is the actual number of 8-bit bytes transferred. Note that the number of 16-bit samples written is half the return value.

Code Example: See [Code Example 6-27](#).

6.9.5.3 *tdc1Read*

Call(s):

```
ssize_t tdc1Read(types_tHandle FileDesc, void * pBuffer, size_t NBytes);
```

Arguments:

Table 6-68. TDC1 Driver Arguments - *tdc1Read*

<i>FileDesc</i>	<i>in</i>	TDC1 device descriptor returned by the <i>tdc1Open</i> call
<i>pBuffer</i>	<i>inout</i>	Pointer to a user buffer of at least <i>NBytes</i> bytes
<i>NBytes</i>	<i>in</i>	Number of 8-bit bytes to read; note that the number of 16-bit samples is <i>NBytes/2</i>

Description: The *tdc1Read* call reads from the TDC1 driver's internal data structures into the user buffer (*pBuffer*). The data is 16-bit signed integers representing linear PCM values. Note that the input parameter *NBytes* specifies the size of the data to write as a number of 8-bit bytes.

The actual number of data bytes read will be equal to *NBytes* if the Driver is operating in Blocking mode. In Blocking mode, the TDC1 driver will not return from the *tdc1Read* function until *NBytes* bytes are transferred from the driver's internal data structures to the user's buffer (*pBuffer*).

However, the number of data bytes written may be less than or equal to *NBytes* if operating in Non-Blocking mode. In Non-Blocking mode, the TDC1 driver will return immediately from the *tdc1Read* function after transferring as much data as possible from the driver's internal data structures to the user buffer (*pBuffer*).

The *tdc1Read* function may be called from normal application code, from an ISR, or from the user's callback function (which is called in turn from the TDC1 Driver's ISR). However, avoid calling *tdc1Read* in Blocking mode from an ISR or from a callback function with more data than the TDC1 driver can immediately transfer (i.e., more than the *MaxNBytes* parameter of the callback function); otherwise, deadlock may occur if TDC1 interrupts are disabled during the *tdc1Read*.

Avoid using *tdc1Read* in both normal application code and from an ISR or callback function. The asynchronous nature of the ISR and callback may lead to re-entrancy issues if you call *tdc1Read* from normal application code as well.

Returns: The *tdc1Read* call returns a value of type *ssize_t*. This return value is the actual number of 8-bit bytes transferred. Note that the number of 16-bit samples read is half the return value.

Code Example: See [Code Example 6-27](#).



6.9.5.4 *tdc1Close*

Call(s):

```
int tdc1Close (types_tHandle FileDesc);
```

Arguments:

Table 6-69. TDC1 Driver Arguments - *tdc1Close*

<i>FileDesc</i>	<i>in</i>	TDC1 device descriptor returned by <i>tdc1Open</i> call

Description: The *tdc1Close* call closes the TDC1 driver and releases the file descriptor handle.

Returns: The *tdc1Close* call returns a type *int*. If successful, this *int* is “0” and “-1” is returned if unsuccessful.

Code Example: See [Code Example 6-27](#).

6.9.5.5 *tdc1loctl*

Call(s):

```
UWord16 tdc1loctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams, const
char *pName);
```

Arguments:

Table 6-70. TDC1 Driver Arguments - *tdc1loctl*

<i>FileDesc</i>	<i>in</i>	TDC1 device descriptor returned by the <i>tdc1Open</i> call
<i>Cmd</i>	<i>in</i>	Commands found in <i>tdc1.h</i> , which are used to control the TDC1 driver and TDC1 device
<i>pParams</i>	<i>in</i>	<i>tdc1loctl</i> command parameter
<i>pName</i>	<i>in</i>	The TDC1 device name; see the <i>bsp.h</i> file for target specific values, such as: BSP_DEVICE_NAME_TDC1_DAA_0 BSP_DEVICE_NAME_TDC1_CODEEC_0

Table 6-71. TDC1 *tdc1loctl* Commands

Cmd	<i>pParams</i>	Ret urn	Result
TDC1_DEVICE_ENABLE	<i>NULL</i>	<i>void</i>	Enables the TDC1 device and starts data transfer
TDC1_DEVICE_DISABLE	<i>NULL</i>	<i>void</i>	Disables the TDC1 device and stops data transfer
TDC1_DEVICE_SET_RX_CALLBACK_LEVEL	<i>unsigned int</i>	<i>void</i>	Dynamically sets the TDC1 DEVICE RX Callback Level; must be less than or equal to TDC1_DAA_FIFO_SIZE or TDC1_CODEEC_FIFO_SIZE, depending on the device addressed by FileDesc
TDC1_DEVICE_SET_TX_CALLBACK_LEVEL	<i>unsigned int</i>	<i>void</i>	Dynamically sets the TX Callback Level; must be less than or equal to TDC1_DAA_FIFO_SIZE or TDC1_CODEEC_FIFO_SIZE depending on the device addressed by FileDesc

Table 6-71. TDC1 *tdc1loctl* Commands (Continued)

Cmd	pParams	Ret urn	Result
TDC1_DEVICE_READ_REG	<i>tdc1_sRegister</i> *	<i>bool</i>	Reads a value from the TDC1 device's register and stores it in the <i>Data</i> element of the structure passed in. Returns "true" if successful or "false" if unsuccessful. The <i>tdc1_sRegister</i> structure is defined in <i>tdc1.h</i>. The user must define the element, <i>Register</i>, to be read. The user must then wait until the element <i>bDataValid</i> is set to "true" before reading the element <i>Data</i>.
TDC1_DEVICE_WRITE_REG	<i>tdc1_sRegister</i> *	<i>bool</i>	Writes a value to the TDC1 device's register. Returns "true" if successful or false if unsuccessful. The <i>tdc1_sRegister</i> structure is defined in <i>tdc1.h</i>. The user must define the element, <i>Register</i>, to be written and the element, <i>Data</i>, to be written.
TDC1_DEVICE_SET_SAMPLE_RATE	<i>int</i> Sample rate values allowed are 7200, 8000, 8229, 8400, 9000, 9600 and 10286 TDC1_3021_SAMPLE_AT_7200 TDC1_3021_SAMPLE_AT_8000 TDC1_3021_SAMPLE_AT_8229 TDC1_3021_SAMPLE_AT_8400 TDC1_3021_SAMPLE_AT_9000 TDC1_3021_SAMPLE_AT_9600 TDC1_3021_SAMPLE_AT_10286	<i>bool</i>	Sets the TDC1 DAA and Codec sample rates. Returns "true" if successful or "false" if unsuccessful.
TDC1_DEVICE_MUTE_HDST_IN	<i>bool</i> TRUE = Mute FALSE = Unmute	<i>bool</i>	Sets the state of the TDC1 codec's handset input mute. Returns "true" if successful or "false" if unsuccessful.
TDC1_DEVICE_MUTE_HDST_OUT	<i>bool</i> TRUE = Mute FALSE = Unmute	<i>bool</i>	Sets the state of the TDC1 codec's handset output mute. Returns "true" if successful or "false" if unsuccessful.

Table 6-71. TDC1 *tdc1ioctl* Commands (Continued)

Cmd	pParams	Return	Result
TDC1_DEVICE_MUTE_MIC	<i>bool</i> TRUE = Mute FALSE = Unmute	<i>bool</i>	Sets the state of the TDC1 codec's microphone mute. Returns "true" if successful or "false" if unsuccessful.
TDC1_DEVICE_MUTE_LINE_IN	<i>bool</i> TRUE = Mute FALSE = Unmute	<i>bool</i>	Sets the state of the TDC1 codec's line in mute. Returns "true" if successful or "false" if unsuccessful.
TDC1_DEVICE_MUTE_LINE_OUT	<i>bool</i> TRUE = Mute FALSE = Unmute	<i>bool</i>	Sets the state of the TDC1 codec's line out mute. Returns "true" if successful or "false" if unsuccessful.
TDC1_DEVICE_MUTE_SPEAKERS	<i>bool</i> TRUE = Mute FALSE = Unmute	<i>bool</i>	Sets the state of the TDC1 codec's right and left speakers mute. Returns "true" if successful or "false" if unsuccessful.
TDC1_DEVICE_SET_RX_GAIN	<i>unsigned int</i> For the Codec device the gain can be adjusted from -34.5db to 12db with 1.5db per step using the macro "TDC1_3000_GAIN(Gain)", where "Gain" is a value in dB, or by using the macro "TDC1_CODEC_GAIN_FROM_PERCENT(x)", where "x" is a value from 0-100. For the DAA device, the gain can be adjusted from 0dB to 12dB with 3dB per step, using the macro "TDC1_3021_RX_GAIN(Gain)", where Gain is a value in dB, or by using the macro "TDC1_DAA_RX_GAIN_FROM_PERCENT(x)", where "x" is a value from 0-100.	<i>bool</i>	Sets the gain of the TDC1 codec's DAC volume or the analog receive gain control field of the TDC1 DAA. Returns "true" if successful or "false" if unsuccessful.

Table 6-71. TDC1 *tdc1loctl* Commands (Continued)

Cmd	pParams	Ret urn	Result
TDC1_DEVICE_SET_TX_GAIN	<i>unsigned int</i> For the Codec device, the gain can be adjusted from -34.5db to 12db with 1.5db per step using the macro "TDC1_3000_GAIN(Gain)", where "Gain" is a value in dB, or by using the macro "TDC1_CODEC_GAIN_FROM_PERCENT(x)", where "x" is a value from 0-100. For the DAA device, the gain can be adjusted from 0dB to 12dB with 3dB per step, using the macro "TDC1_3021_TX_GAIN(Gain)", where "Gain" is a value in dB, or by using the macro "TDC1_DAA_TX_GAIN_FROM_PERCENT(x)", where "x" is a value from 0-100.	<i>bool</i>	Sets the gain of the TDC1 codec's ADC volume or the analog transmit gain control field of the TDC1 DAA. Returns "true" if successful or "false" if unsuccessful.
TDC1_DEVICE_OFF_HOOK	<i>bool</i> TRUE = OFF HOOK FALSE = ON HOOK	<i>bool</i>	Sets the off-hook state of the line side chip. Returns "true" if successful or "false" if unsuccessful.
TDC1_DEVICE_MUTE_TX	<i>bool</i> TRUE = Mute FALSE = Unmute	<i>bool</i>	Sets the state of the TDC1 DAA's Tx signal mute. Returns "true" if successful or "false" if unsuccessful.
TDC1_DEVICE_MUTE_RX	<i>bool</i> TRUE = Mute FALSE = Unmute	<i>bool</i>	Sets the mute state of the TDC1 DAA's Rx signal. Returns "true" if successful or "false" if unsuccessful.
TDC1_DEVICE_RING_DETECT	<i>none</i>	<i>bool</i>	Checks for a ring and returns "true" or "false"
TDC1_DEVICE_FRAME_DETECT	<i>none</i>	<i>bool</i>	Checks for frame lock on the ISOcap link and returns "true" or "false".

Macros: The macro TDC1_CODEC_GAIN_FROM_PERCENT(x), where $0 \leq x \leq 100$, can be used as a portable method to calculate a target-specific gain value from the percentage "x". A value of 0% returns the maximum amount of attenuation available from the TDC1 codec; a value of 100% returns the maximum amount of gain available from the TDC1 codec.

The macro TDC1_DAA_TX_GAIN_FROM_PERCENT(x), where $0 \leq x \leq 100$, can be used as a portable method to calculate a target-specific TX gain value from the percentage "x". A value of 0% returns the maximum amount of denotation available from the TDC1 DAA codec transmitter;

a value of 100% returns the maximum amount of gain available from the TDC1 DAA codec transmitter.

The macro `TDC1_DAA_RX_GAIN_FROM_PERCENT(x)`, where $0 \leq x \leq 100$, can be used as a portable method to calculate a target-specific TX gain value from the percentage “x”. A value of 0% returns the maximum amount of denotation available from the TDC1 DAA codec receiver; a value of 100% returns the maximum amount of gain available from the TDC1 DAA codec receiver.

The macro `TDC1_3000_GAIN(Gain)` can be used to input gain values in dB. The macro accepts gain values in db from -34.5db to 12db.

The macro `TDC1_3021_TX_GAIN(Gain)` can be used to input gain values in db. This macro accepts gain values in db from 0db to 12db.

The macro `TDC1_3021_RX_GAIN(Gain)` can be used to input gain values in db. This macro accepts gain values in db from 0db to 12db.

Description: The *ioctl* call provides various control functions for the TDC1. See the list of commands in [Table 6-71](#) for specific control functions.

Returns: The *ioctl* call returns a value dependent on the command passed in as an input parameter. [Table 6-71](#) lists these returns based on the specific command.

Code Example: [Code Example 6-27](#) reads and writes eight codec samples (16 bytes) at a time, using the device-dependent layer API to the TDC1 driver in a Non-Blocking mode.

Code Example 6-27. TDC1 Driver Low Level API Example

```

/*****
#include "tdc1.h"
#include "bsp.h"
#include "led.h"

#define NUM_SAMPLES    RX_FULL_SIZE

UWord16 Status;

volatile types_tHandle Tdc1Daa, Tdc1Codec;

volatile UWord16 Tdc1DaaBufferFull, Tdc1DaaBufferEmpty;
volatile UWord16 Tdc1CodecBufferFull, Tdc1CodecBufferEmpty;

volatile UWord16  LoopCount = 0;
volatile UWord16  LedCount = 0;

UWord16 pDaaSamples[FIFO_SIZE];
UWord16 pCodecSamples[FIFO_SIZE];

/*
        Forward declarations
*/

```



```

void Tdc1DaaRXISR (void *pCallbackArg, int MaxNBytes);
void Tdc1DaaTXISR (void *pCallbackArg, int MaxNBytes);
void Tdc1CodecRXISR (void *pCallbackArg, int MaxNBytes);
void Tdc1CodecTXISR (void *pCallbackArg, int MaxNBytes);

/*****/
void Tdc1DaaRXISR (void *pCallbackArg, int MaxNBytes)
{
    Tdc1DaaBufferFull = true;
    LoopCount++;
    LedCount += MaxNBytes/2;
}

/*****/
void Tdc1DaaTXISR (void *pCallbackArg, int MaxNBytes)
{
    Tdc1DaaBufferEmpty = true;
}
/*****/
void Tdc1CodecRXISR (void *pCallbackArg, int MaxNBytes)
{
    Tdc1CodecBufferFull = true;
}

/*****/
void Tdc1CodecTXISR (void *pCallbackArg, int MaxNBytes)
{
    Tdc1CodecBufferEmpty = true;
}

/*****/
void main(void)
{
    types_tHandle Led;
    Word16 samples;
    Word16 i,j;
    int samp = 0;
    bool OffHook = true;
    tdc1_sRegister Register;
    UWord16 rate[] = {TDC1_3021_SAMPLE_AT_7200, TDC1_3021_SAMPLE_AT_8000,
        TDC1_3021_SAMPLE_AT_8229, TDC1_3021_SAMPLE_AT_8400,
        TDC1_3021_SAMPLE_AT_9000, TDC1_3021_SAMPLE_AT_9600,
        TDC1_3021_SAMPLE_AT_10286};

    Tdc1DaaBufferFull = false;
    Tdc1DaaBufferEmpty = false;
    Tdc1CodecBufferFull = false;
    Tdc1CodecBufferEmpty = false;

    /* Set up LEDs */
    Led = ledOpen(BSP_DEVICE_NAME_LED_0, 0);

    /* Initialize the buffer */
    for (i=0; i<FIFO_SIZE; i++)
    {

```

ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

```

        pDaaSamples[i] = 0;
        pCodecSamples[i] = 0;
    }

    /* Set up Tdc1 */

    /* open both the DAA and the codec */

    Tdc1Daa = tdc1open(BSP_DEVICE_NAME_TDC1_DAA_0, O_RDWR | O_NONBLOCK);
    Tdc1Codec = tdc1open(BSP_DEVICE_NAME_TDC1_CODEC_0, O_RDWR | O_NONBLOCK);

    /* Set the si3000 codec's speaker drive and line drive */
    /* to normal operation */

    Register.Register = 1;
    Register.Data = 0x18;
    tdc1Ioctl(Tdc1Codec, TDC1_DEVICE_WRITE_REG, &Register);

    /* Unmute the si3000 codec's line out */
    /* The while loop is too allow the previous write to finish before */
    /* issuing another write. If the previous write is not done the */
    /* function will simply return "false" and not issue the write */
    /* request. */

    while (!tdc1Ioctl(Tdc1Codec, TDC1_DEVICE_MUTE_LINE_OUT, false));

    /* Set the si3000 codec's Rx gain to 0dB and unmute the */
    /* left and right speaker */

    while (!tdc1Ioctl(Tdc1Codec, TDC1_DEVICE_MUTE_SPEAKERS, false));
    while (!tdc1Ioctl(Tdc1Codec, TDC1_DEVICE_SET_RX_GAIN, TDC1_3000_GAIN(0)));

    /* The following sequence of control commands is not really */
    /* necessary, but is provided as an example. */

    tdc1Ioctl (Tdc1Codec, TDC1_DEVICE_DISABLE, NULL);
    while (!tdc1Ioctl (Tdc1Codec, TDC1_DEVICE_MUTE_LINE_IN, false));
    while (!tdc1Ioctl (Tdc1Codec, TDC1_DEVICE_SET_RX_GAIN,
        TDC1_CODEC_GAIN_FROM_PERCENT(75)));
    while (!tdc1Ioctl (Tdc1Codec, TDC1_DEVICE_SET_TX_GAIN,
        TDC1_CODEC_GAIN_FROM_PERCENT(75)));
    while (!tdc1Ioctl (Tdc1Daa, TDC1_DEVICE_SET_RX_GAIN,
        TDC1_DAA_RX_GAIN_FROM_PERCENT(75)));
    while (!tdc1Ioctl (Tdc1Daa, TDC1_DEVICE_SET_TX_GAIN,
        TDC1_DAA_TX_GAIN_FROM_PERCENT(75)));
    tdc1Ioctl (Tdc1Daa, TDC1_DEVICE_SET_RX_CALLBACK_LEVEL, 32);
    tdc1Ioctl (Tdc1Daa, TDC1_DEVICE_SET_TX_CALLBACK_LEVEL, 0);

    /* The following Enable command actually starts tdc1 data transfer */
    tdc1Ioctl (Tdc1Daa, TDC1_DEVICE_ENABLE, NULL);
    tdc1Ioctl (Tdc1Codec, TDC1_DEVICE_ENABLE, NULL);

    /* The executive loop just toggles the LED depending on the number of */
    /* Samples received from the Tdc1Daa. Loopback is done in the ISRs */
    /* or below depending on the state of "USE_CALLBACK" in the */
    /* appconfig.h file. */

```



```

ledIoctl(Led, LED_ON, LED_GREEN2, BSP_DEVICE_NAME_LED_0);
while(1)
{
    if (Tdc1DaaBufferFull)
    {
        /* Read speech samples from the tdc1. Returns number of */
        /* bytes read */
        samples = tdc1Read(Tdc1Daa, (void*)pDaaSamples,
                           (NUM_SAMPLES<<1));
        Tdc1DaaBufferFull = false;
    }

    if (Tdc1DaaBufferEmpty)
    {
        /* Write speech samples to the tdc1. Returns number of */
        /* bytes written */
        samples = tdc1Write(Tdc1Daa, (void*)pDaaSamples,
                           (NUM_SAMPLES<<1));
        Tdc1DaaBufferEmpty = false;
    }

    if (Tdc1CodecBufferFull)
    {
        /* Read speech samples from the tdc1_2. Returns number */
        /* of bytes read */
        samples = tdc1Read(Tdc1Codec, (void*)pCodecSamples,
                           (NUM_SAMPLES<<1));
        Tdc1CodecBufferFull = false;
    }

    if (Tdc1CodecBufferEmpty)
    {
        /* Write speech samples to the tdc1_2. Returns number */
        /* of bytes written */
        samples = tdc1Write(Tdc1Codec, (void*)pCodecSamples,
                           (NUM_SAMPLES<<1));
        Tdc1CodecBufferEmpty = false;
    }

    if (LedCount > 7200)
    {
        ledIoctl(Led, LED_TOGGLE, LED_GREEN2,
                 BSP_DEVICE_NAME_LED_0);
        LedCount = 0;
    }

    if (LoopCount > 2250)
    { /* approximately every 10 seconds at 7200 samples/s */

        /* Read the DAA's sample rate register */
        Register.Register = 9;
        Register.Data = 0;
        while(!tdc1Ioctl(Tdc1Daa, TDC1_DEVICE_READ_REG, &Register));
        while(!Register.bDataValid);
        Status = Register.Data;
    }
}

```



ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

```
/* Verify that the DAA's register contents are correct */
if (Status != rate[samp])
{
    asm(nop);
    asm(nop);
}

/* Read the Codec's DAC volume control register */
Register.Register = 7;
Register.Data = 0;
while(!tdc1Ioctl(Tdc1Codec, TDC1_DEVICE_READ_REG,
    &Register));
while(!Register.bDataValid);
Status = Register.Data;

/* Verify that the Codec's register contents are correct */
if (Status != 0x5f)
{
    asm(nop);
    asm(nop);
}

/* Read the Codec's status register */
Register.Register = 8;
Register.Data = 0;
while(!tdc1Ioctl(Tdc1Codec, TDC1_DEVICE_READ_REG,
    &Register));
while(!Register.bDataValid);
Status = Register.Data;

/* Verify that the Codec's status register does not */
/* indicate an error */
if (Status != 0x00) {
    asm(nop);
    asm(nop);
}

/* Toggle the Off-Hook bit */
while (!tdc1Ioctl(Tdc1Daa, TDC1_DEVICE_OFF_HOOK, OffHook));
if (OffHook)
{
    OffHook = false;
}
else
{
    OffHook = true;
}

/* Increment to the next sample rate and load the */
/* DAA's register */
samp++;
if (samp > 6)
{
```



```
        samp = 0;
    }
    while (!tdc1Ioctl(Tdc1Daa, TDC1_DEVICE_SET_SAMPLE_RATE,
        rate[samp]));

    /* Read the Codec's status register */
    Register.Register = 8;
    Register.Data = 0;
    while(!tdc1Ioctl(Tdc1Codec, TDC1_DEVICE_READ_REG,
        &Register));
    while(!Register.bDataValid);
    Status = Register.Data;

    LoopCount = 0;

}
}
```


6.9.6 TDC1 Driver Tests

Currently, there are no specific tests provided for the TDC1 driver. However, many applications and demos use the TDC1 driver. In particular, the TDC1 driver applications in the next section exercise most TDC1 driver capabilities.

6.9.7 TDC1 Driver Applications

6.9.7.1 TDC1 Loopback Application

This application performs a simple audio loopback using TDC1 hardware and a DSP56852 processor. The application loops the data going into the TDC1 audio codec to the output of the TDC1 audio codec.

This loopback application provides a good example of how to use the TDC1 driver interface functions.

6.9.7.1.1 Set-up for TDC1 Loopback Application

EVM Jumper Settings:

- Make the following changes to the default jumper settings for the DSP56852EVM board:
 - Remove all the jumpers from Jumper Group JG9. This disables the SSI signals to the onboard codec device. This also enables control of the TDC1 codecs via the SSI port of the DSP56852.
- The complete list of jumper settings are shown in the **DSP56852 Evaluation Module User's Manual**

TDC1 Jumper Settings:

- Make the following changes to the default jumper settings for the TDC1 board:
 - Set Switch Position 1 to On
 - Set Switch Position 2 and 3 to Off
- Plug the TDC1 daughter card on to the DSP56852 Evaluation Module.
- Connect a line-level audio source to the “Ch 1 Line In” post on the TDC1 card
- Connect the line-level audio source ground to the “Ch 1 Line GND” post on the TDC1 card
- Connect a speaker or headphone to the “Ch 1 Speaker” jack on the TDC1 card

NOTES:

- The audio input source can be any line-level audio source (PC audio out, CD player, etc.)
- The audio output is commonly connected to an external power speaker, but any other audio devices can be used to output sound (head phones, PC audio input, etc.)

6.9.7.1.2 Procedure for TDC1 Loopback Application

- Using CodeWarrior, open:
 - ...\\nos\\applications\\bsp\\tdc1\\tdc1.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- Once the debugger is running, apply an audio signal to the audio input of the hardware set-up; see [Section 6.9.7.1.1](#)
- If successful, you will hear audio output from the speaker, which should be the same as the input audio
 - You should also observe a blinking green LED, which indicates that the application is running



- The program cycles through all the sample rates of the TDC1 codec from 7200 to 10286
 - You should hear a click on the speaker along with the audio when the sample rate changes

Chapter 7

Libraries

This chapter describes all SDK libraries that are not part of the core libraries (see **SDK Programmer's Guide**), and are part of the SDK DSP56852 release. The domain-specific libraries are:

- DSP Functional Library (*dspfunc.lib*)
- V.8bis Library (*v8bis.lib*)
- V.21 Library (*v21.lib*)
- V.22bis Library (*v22.lib*)
- V.42bis Library (*V42BIS.lib*)
- Data Encryption Standard (DES) Library (*des.lib*)
- Triple Data Encryption Standard (3DES) Library (*3des.lib*)
- RSA Library (*rsa.lib*)
- Acoustic Echo Canceller Library (*aec.lib*)
- Automatic Gain Control (*agc.lib*)
- CAS Detect Library (*cas_detect.lib*)
- Call Progress Tones Detect Library (*cpt.lib*)
- Caller ID Library (*caller_id.lib*)
- DTMF Detection Library (*dtmf_det.lib*)
- DTMF Generation Library (*dtmf_gen.lib*)
- G.165 Library (*g165.lib*)
- G.168 Library (*g168.lib*)
- G.711 Library (*g711.lib*)
- G.726 Library (*g726.lib*)
- MFCR2 Detection Library (*mfc2.lib*)
- Noise Suppression Library (*ns.lib*)
- Voice Activity Detector Library (*vad.lib*)
- VRLite-1 Library (*vrlite.lib*)
- Generic Echo Canceller Library (*gec.lib*)
- Type 1 Telephony Parser Library (*cidparser.lib*)
- Full Duplex Speakerphone Library (*fdspk.lib*)
- Type 1 Telephony Features Library (*Type1CID.lib*)
- Type 1 and 2 Telephony Features Library (*Type12CID.lib*)

7.1 Signal Processing Libraries

Signal processing libraries provide functionality to general-purpose DSP algorithms.

7.1.1 DSP Function Library

The Digital Signal Processing Function Library provides mathematical algorithms for digital signal processing applications using fractional data types. The fractional data type is a fixed point representation encompassing the range $-1 \leq \text{fractional range} < 1$. The fractional data type is one of the strengths of Motorola's architecture because it can be used efficiently for such computationally-intensive algorithms as digital filters, speech coders, vector and array processing, digital control, and other signal processing tasks.

The Function Library includes over 50 general-purpose algorithms. These algorithms share a standard interface definition across many different processors, yet have been highly optimized in their implementation for DSP56800E architecture. These algorithms cover the following functional categories:

- Basic Fractional Math Primitives
- Trigonometric Functions
- Vector Processing Functions
- Matrix Operation Functions
- Fast Fourier Transforms (FFT)
- Finite Impulse Response (FIR) filters
- Infinite Impulse Response (IIR) filters
- Correlation

A test that verifies the operation of the DSP Function library is described in [Section 8.3](#).

This library is described in more detail in the **Digital Signal Processing Function Library**.

7.1.2 Performance

This section details the performance of each of the DSP Function Library functions on the target DSP56852.

7.1.2.1 Trigonometric Math Function Performance

This section documents the performance of the trigonometric functions on the DSP56852.

[Table 7-1](#) shows the counts taken over the input range $[-1, 1]$.

Table 7-1. Trigonometric Math Function Performance

Trigonometric Function	Instruction Cycle Count	Program Memory (words)	Data Memory (words)	Data ROM (words)	Stack Depth (words)
sinPlx	900	92	42	0	0

Table 7-1. Trigonometric Math Function Performance

Trigonometric Function	Instruction Cycle Count	Program Memory (words)	Data Memory (words)	Data ROM (words)	Stack Depth (words)
cosPlx	1078	48	42	0	0
AsinOverPI	1237	89	42	0	0
AcosOverPI	1375	30	42	0	0
AtanOverPI	2337	85	42	0	0
Atan2OverPI	2606	84	63	0	0
tfr16SineWaveGenPAM	22+325n	86	0	0	2
tfr16SineWaveGenIDTL	44+9n	153	0	M	0
tfr16SineWaveGenRDTL	36+16n	121	0	M	2
tfr16SineWaveGenRDITL	19+86n	202	0	M	0
tfr16SineWaveGenDOM	35+8n	139	0	0	2
tfr16CosPlxLUT	55n	94	0	M	0
tfr16SinPlxLUT	58n	140	0	M	0
tfr16SineWaveGenRDITLQ	24+67n	195	0	M	2
tfr16WaveGenRDITLQ	78n	167	0	M	0

n corresponds to the number of samples to be generated

M corresponds to sine table length

$M = f / f_s$, where f = fundamental frequency and f_s = sampling frequency

7.1.2.2 Signal Processing Function Performance

This section documents the performance of the Digital Signal Processing functions on the DSP56852.

7.1.2.2.1 *autoCorr*

Table 7-2 gives the approximate worst-case cycle counts and machine instructions.

Table 7-2. *autoCorr* Performance

Options	Instruction Cycle Count	Program Memory (words)	Data Memory (words)	Data ROM (words)	Stack Depth (words)
CORR_RAW	$n^2+28n+63$	96	0	0	0
CORR_BIAS	$n^2+28n+90$				
CORR_UNBIAS	$n^2+51n+74$				

n = Input vector length

7.1.2.2.2 *cbitrev*

Table 7-3 details the approximate worst-case instruction cycle count for Complex Bit-Reverse ASM Code.

Table 7-3. *cbitrev* Performance

Size of Complex FFT, N	Instruction Cycle Count	Program Memory (words)	Data Memory (words)	Data ROM (words)	Stack Depth (words)
8	209	43	0	0	0
16	429				
32	887				
64	1791				
128	3627				
256	7267				
512	14595				
1024	29179				
2048	58435				

7.1.2.2.3 *cfft*

Table 7-4 shows the approximate worst-case instruction cycle counts for the ASM code. The figures in **Table 7-4** do not include instruction cycles needed by the *cbitrev* function.

Table 7-4. *cfft* Performance

Size of Complex FFT, N	Instruction Cycle Count for Core Complex FFT			Program Memory (words)	Data Memory (words)	Data ROM (words)	Stack Depth (words)
	AS	BFP	NS				
8	267	497	267	530	0	8	17
16	647	1097	647			16	
32	1495	2438	1495			32	
64	3391	5435	3391			64	
128	7607	13096	7607			128	
256	16143	26789	16143			256	
512	35737	58954	35737			512	
1024	78527	128875	78527			1024	
2048	171223	279956	171223			2048	

AS stands for Auto Scale

BFP stands for Block Floating Point

NS stands for No Scale

7.1.2.2.4 *cifft*

Table 7-5 details the approximate worst-case instruction cycle counts for the ASM code. The figures in **Table 7-5** do not include instruction cycles needed by the *cbitre* function.

Table 7-5. *cifft* Performance

Size of Complex IFFT, N	Instruction Cycle Count for Core Complex IFFT			Program Memory (words)	Data Memory (words)	Data ROM (words)	Stack Depth (words)
	AS	BFP	NS				
8	277	508	277	540	0	8	17
16	669	1120	669			16	
32	1541	2485	1541			32	
64	3485	5530	3485			64	
128	7797	13287	7797			128	
256	16525	27172	16525			256	
512	36503	59721	36503			512	
1024	80062	130410	80062			1024	
2048	174293	283027	174293			2048	

AS stands for Auto Scale

BFP stands for Block Floating Point

NS stands for No Scale

7.1.2.2.5 *corr*

Table 7-6 shows the approximate worst-case cycle counts.

Table 7-6. corr Performance

Options	Instruction Cycle Count	Program Memory (words)	Data Memory (words)	Data ROM (words)	Stack Depth (words)
CORR_RAW	$52+30(nX+nY)+(nX)^2+(nY)^2$	144	0	0	2
CORR_BIAS	$95+30(nX+nY)+(nX)^2+(nY)^2$				
CORR_UNBIAS	$31+51(nX+nY)+(nX)^2+(nY)^2$				

nX = length of the first input vector
nY = length of the second input vector

7.1.2.2.6 *fir*

Performance of the *fir* functions vary, depending on whether *firCreate* was able to align the history buffer in memory to perform modulo addressing and to allocate the coefficient buffer within internal memory.

For the performance formulas below, n = the number of input samples to be processed by the filter and f = the number of filter coefficients.

Case 1: History buffer aligned for modulo addressing; coefficients in internal memory

Number of machine instructions (uses REP): $29 + 13n$
Number of oscillator cycles: $132 + n(2f + 50)$
Number of oscillator cycles interrupts blocked: $2f$ (due to REP instruction)

Case 2: History buffer aligned for modulo addressing; coefficients in external memory

Number of machine instructions: $31 + n(2f + 11)$
Number of oscillator cycles: $140 + n(6f + 50)$
Number of oscillator cycles interrupts blocked: 0

Case 3: History buffer not aligned for modulo addressing; coefficients in external memory

Number of machine instructions: $29 + n(6f + 22)$
Number of oscillator cycles: $144 + n(22f + 86)$
Number of oscillator cycles interrupts blocked: 0

7.1.2.2.7 *iir*

Performance of the *iir* functions vary, depending on whether *iirCreate* was able to align the history buffer in memory to perform modulo addressing and to allocate the coefficient buffer within internal memory.

The following cases are defined:

Case 1: History buffer aligned for modulo addressing; coefficients in internal memory
Case 2: History buffer aligned for modulo addressing; coefficients in external memory
Case 3: History buffer **not** aligned for modulo addressing; coefficients in external memory

Table 7-7. *iir* Performance

Case	Instruction Cycle Count	Program Memory (words)	Data Memory (words)	Data ROM (words)	Stack Depth (words)
Case 1	$50 + n(16 + 11nbq)$	121	0	0	4
Case 2	$50 + n(20 + 11nbq)$				
Case 3	$55 + n(22 + 11nbq)$				

n = the number of input samples to be processed by the filter

nbq = the number of filter biquads; each biquad uses 5 coefficients

7.1.2.2.8 *rfft*

Table 7-8 details the approximate worst-case instruction cycle counts for the ASM code. This instruction cycle count does not include the instruction cycles needed by the *cbitrev* function.

Table 7-8. *rfft* Performance

Size of Real FFT, N	Instruction Cycle Count for Core Real FFT			Program Memory (words)	Data Memory (words)	Data ROM (words)	Stack Depth (words)
	AS	BFP	NS				
8	198	194	181	578	0	4	17
16	430	436	417			8	
32	928	954	923			16	
64	2014	2080	2025			32	
128	4388	4534	4431			64	
256	9562	9868	9669			128	
512	20784	21410	21019			256	
1024	44998	46264	45489			512	
2048	96988	99534	97991			1024	

7.1.2.2.9 *rifft*

Table 7-9 shows the approximate worst-case instruction cycle counts for the ASM code. This instruction cycle count does not include the instruction cycles needed by the *rifft* function.

Table 7-9. *rifft* Performance

Size of Real IFFT, N	Instruction Cycle Count for Core Real IFFT			Program Memory (words)	Data Memory (words)	Data ROM (words)	Stack Depth (words)
	AS	BFP	NS				
8	194	356	197	565	0	4	17
16	436	713	439			8	
32	954	1478	957			16	
64	2080	3151	2083			32	
128	4534	6824	4537			64	
256	9868	14865	9871			128	
512	21410	32378	21413			256	
1024	46264	70307	46267			512	
2048	99534	152012	99537			1024	

7.2 Modem Libraries

Modem libraries provide modem functionality.

7.2.1 V.8bis Library

The V.8bis allows DCEs and DTEs with multiple modes of operation over the PSTN and on leased telephone-type circuits to perform these functions:

- selection of the desired mode of operation at automatic call establishment on the PSTN, controlled by either the calling or answering station
- selection of the desired mode of operation while in telephony mode on an already-established connection, controlled by either station
- determination by either station of whether the remote station supports V.8bis, with minimum disturbance to a voice caller
- exchange of available capabilities between stations on a connection at call establishment or while in telephony mode
- graceful recovery in the event of transmission of errors or selection of an unavailable mode of operation

These functions are provided by defining a set of signals, messages and procedures. Signals are intended to be detected in the presence of an interfering voice or other audio; to turn around any echo suppressors in the network prior to the beginning of information transmission; and to

indicate the initiation of a V.8bis transaction to the receiving station, while not appearing to the user and receiver as an indication of a data or facsimile device.

Messages convey significantly more information than signals, but may be used only when they will not cause disruption to a voice caller. They are intended to be used only in the absence of an interfering voice or other audio.

This Recommendation provides for error detection and rejection of corrupted messages, and rejection of mode selections that are unavailable.

7.2.1.1 Performance

Table 7-10 details the V.8bis memory and MIPS requirements for internal memory on the DSP56852.

Table 7-10. V.8bis Memory and MIPS Requirements for Internal Memory on the DSP56852

	Program Memory (words)	Data ROM (words)	Software Stack Per Channel (words)	Data RAM Per Channel (words)	MIPS
Library allocates memory [includes <i>v8bisCreate()</i> and <i>v8bisDestroy()</i>]	9573	557	80	1899	N/A
User application allocates memory [does not include <i>v8bisCreate()</i> and <i>v8bisDestroy()</i>]	9465	557	80	1899	N/A

Notes:

- Data RAM figures in **Table 7-10** do not include the gaps (unused memory) that will be introduced due to circular buffers. These buffers take advantage of the modulo buffer indexing feature of the Motorola DSP architecture, which has strict memory alignment requirements).
- **Table 7-10** does not include memory requirements for the input and output buffers and the structures which must be passed to the calling modules.

The v.21 modem and the tone generator and detector algorithms are MIPS-intensive modules. The MIPS for these threads for the DSP56852 are shown in **Table 7-11**.

Table 7-11. MIPS on the DSP56852

Module Name	MIPS at 7200Hz sampling rate
V.21 modem Transmitter	0.1134
V.21 modem Receiver	1.2516
Single Tone Generation	0.0647
Single Tone Detection	0.8656
DTMF Generation	0.1655
DTMF Detection	0.9992

Notes:

- In [Table 7-11](#), MIPS specified for baud processing are the peak values
- In [Table 7-11](#), MIPS specified are for the period while the complete program and data are kept in internal memory

This library is described in more detail in the document [V.8bis Library](#).

7.2.2 V.21 Library

V.21 is a 300 bits-per-second duplex modem for use in the general switched telephone network. The V.21 uses a binary modulation obtained by frequency shift, resulting in a modulation rate equal to the data signaling rate (i.e., baud rate = bit rate). Since V.21 is a duplex modem, each unit has two channels, one for transmission and one for reception. The frequencies at which the channels work are shown in the following table. The V.21 library works at 7200Hz sampling rate.

Table 7-12. Frequencies Used in V.21

Channel Number	Nominal Mean Frequency	Mark Frequency	Zero Frequency
1	1080Hz	980Hz	1180Hz
2	1750Hz	1650Hz	1850Hz

Even high-end modems like V.90 might require a fallback to lower data-rate modes (e.g., 300bps, 75bps). V.21 provides a solution to this requirement by allowing a data rate of 300bps.

7.2.2.1 Performance

[Table 7-13](#) shows V.21 Memory and MIPS figures for internal memory on the DSP56852.

Table 7-13. V.21 Memory and MIPS Requirements for Internal Memory on the DSP56852

	Program Memory (words)	Data ROM (words)	Software Stack Per Channel (words)	Data RAM Per Channel (words)	MIPS at 7200Hz Sampling Rate	
					Transmitter	Receiver
Library allocates memory [includes <i>v21Create()</i> and <i>v21Destroy()</i>]	2325	535	24	248	0.14	1.0
User application allocates memory [does not include <i>v21Create()</i> and <i>v21Destroy()</i>]	2081	535	24	248	0.14	1.0

Notes:

- Memory resources in [Table 7-13](#) do not include the gaps (unused memory) that are introduced due to circular buffers
- [Table 7-13](#) does not include memory requirements for the input and output buffers and the structures which must be passed to the calling modules

This library is described in more detail in the document **V.21 Library**.

7.2.3 V.22bis Library

V.22bis conforms to the V.22bis Low-Speed Modem ITU-T Standard.

Lower-end internet telephony and e-cash applications require a low-speed modem for exchanging data. ITU-T standard V.22bis with 2400 bits per second is usually the preferred modem for these applications. Motorola has internally developed a single library module containing the V.22bis modem. The modem was tested over all possible telephone networks available in continental North America using a network simulator and is available for the Motorola DSP568xx range of processors.

7.2.3.1 Performance

Processing requirements for the V.22bis are shown in [Table 7-14](#).

Table 7-14. V.22bis Memory and MIPS Requirements for Internal Memory on the DSP56852

	Program Memory (words)	Data ROM (words)	Software Stack Per Channel (words)	Data RAM Per Channel (words)	MIPS	
					Transmitter	Receiver
Library allocates memory [includes <i>v22bisCreate()</i> and <i>v22bisDestroy()</i>]	5778	1065	23	Only one channel exists Data RAM per channel = 310	0.94	6.21
User application allocates memory [does not include <i>v22bisCreate()</i> and <i>v22bisDestroy()</i>]	5744	1065	23	Only one channel exists Data RAM per channel = 310	0.94	6.21

Notes:

- MIPS specified for baud processing are the peak values
- Data RAM figures in [Table 7-14](#) do not include the gaps (unused memory) that will be introduced due to circular buffers. These buffers take advantage of the modulo buffer indexing feature of the Motorola DSP architecture, which has strict memory alignment requirements. The table does not include memory requirement for the input and output buffers, nor for the structures which are to be passed to the calling modules.

This library is described in more detail in the document **V.22bis Library**.

7.2.4 V.42bis Library

The V.42bis recommendation describes a data compression procedure that can be used while transferring data to improve throughput. The DCE then uses the error-correcting procedures defined in the Recommendation.

The standard also provides options for transferring data in un-compressed (transparent) mode, which is useful when the transferred data is already compressed. The compressibility of the data is tested periodically to choose the most efficient mode.

The compression algorithm used is the LZW algorithm, a version of Lempel and Ziv's LZ2, modified by Terry Welch. The algorithm makes use of the data's repeating patterns. Every subsequent occurrence of a pattern can be encoded as an index to its previous occurrence, rather than encoding the entire pattern itself. These frequently-occurring strings are stored in a dictionary.

For successful decoding operation, the decoder must construct a dictionary, which is identical to the dictionary used by the encoder. Since the dictionary is constructed dynamically, the encoder cannot use entries which were added before the decoder's construction process. The standard

specifies the dictionary construction procedure under normal operations and under such special modes of operation as transition from transparent to compressed modes, flush request, etc.

7.2.4.1 Performance

Processing requirements for the V.42bis are shown in [Table 7-15](#). and [Table 7-16](#).

Table 7-15. V.42bis Memory Requirements on the DSP56852

	Program Memory (words)	Data ROM (words)	Software Stack Per Channel (words)	Data RAM Per Channel (words)	MIPS
Library allocates memory [includes <i>v42bisCreate()</i> and <i>v42bisDestroy()</i>]	2381	0	35	DictionarySize * 4 + 72	NA
User application allocates memory [does not include <i>v42bisCreate()</i> and <i>v42bisDestroy()</i>]	2171	0	35	DictionarySize * 4 + 72	NA

Table 7-16. V42bis Cycle Count for Internal Memory on the DSP56852

Dictionary Parameters	Encoder Cycle Count	Decoder Cycle Count
P1 = 512, P2 = 6	MAX count = 7000 MIN count = 176	MAX count = 7000 MIN count = 144
P1 = 1024, P2 = 10	MAX count = 7000 MIN count = 176	MAX count = 7000 MIN count = 144

P1 is the dictionary size and P2 is the string length.

Note: The data memory in [Table 7-15](#) includes the memory for both encoder and decoder dictionaries. Both share equal amounts of memory. The dictionary size for the memory requirement is P1 nodes, and each node requires four locations. The encoder's dictionary requirement is P1 * 4 words. The decoder takes another P1 * 4 words. [Table 7-16](#) includes cycle count for V.42bis encoder and decoder for different dictionary sizes. Cycle Count in [Table 7-16](#) = Oscillator Cycle Count.

This library is described in more detail in the document **V.42bis Library**.

7.3 Security Libraries

Security libraries provide information and transmission security algorithms.

7.3.1 DES Library

Data Encryption Standard (DES) conforms to the Federal Information Processing Standards Publications 46-2 and 81 (FIPS PUB 46-2 and FIPS PUB 81).

Any secure communication/storage requires converting the data to be transmitted into an unintelligible form, called cipher, and reconvertng the data back to its original form, called plaintext. The FIPS PUB 46 standard describing DES is the most widely-used encryption standard for these applications. Motorola has internally developed a single library module containing the DES in CFB and CBC modes.

7.3.1.1 Performance

Table 7-17 details DES memory requirements for internal memory on the DSP56852.

Table 7-17. DES Memory and MIPS Requirements for Internal Memory on the DSP56852

	Program Memory (words)	Data ROM (words)	Software Stack Per Channel (words)	Data RAM Per Channel (words)	MIPS
Library allocates memory [includes <i>desCreate()</i> and <i>desDestroy()</i>]	2752	742	Encrypt (CBC Mode) 42 Decrypt (CBC Mode) 42 Encrypt (CFB Mode) 42 Decrypt (CFB Mode) 42	901	N/A
User application allocates memory [does not include <i>desCreate()</i> and <i>desDestroy()</i>]	2292	742	Encrypt (CBC Mode) 42 Decrypt (CBC Mode) 42 Encrypt (CFB Mode) 42 Decrypt (CFB Mode) 42	901	N/A

Note: Data RAM figures in **Table 7-18** do not include the gaps (unused memory) that will be introduced due to circular buffers. These buffers take advantage of the modulo buffer indexing feature of the Motorola DSP architecture, which has strict memory alignment requirements.

Table 7-18 contains DES cycle count information for internal memory on the DSP56852.

Table 7-18. DES Cycle Counts for Internal Memory on the DSP56852

Module	Instruction Cycle Count for CFB Mode	Instruction Cycle Count for CBC Mode
Initialization	20795	20795
Encryption	238217	116596
Decryption	239262	117460

Notes:

- Instruction Cycle Count = Oscillator Clock Cycles
- The cycle counts in [Table 7-18](#) are obtained for an input block (data to be encrypted or decrypted) of size 128 characters

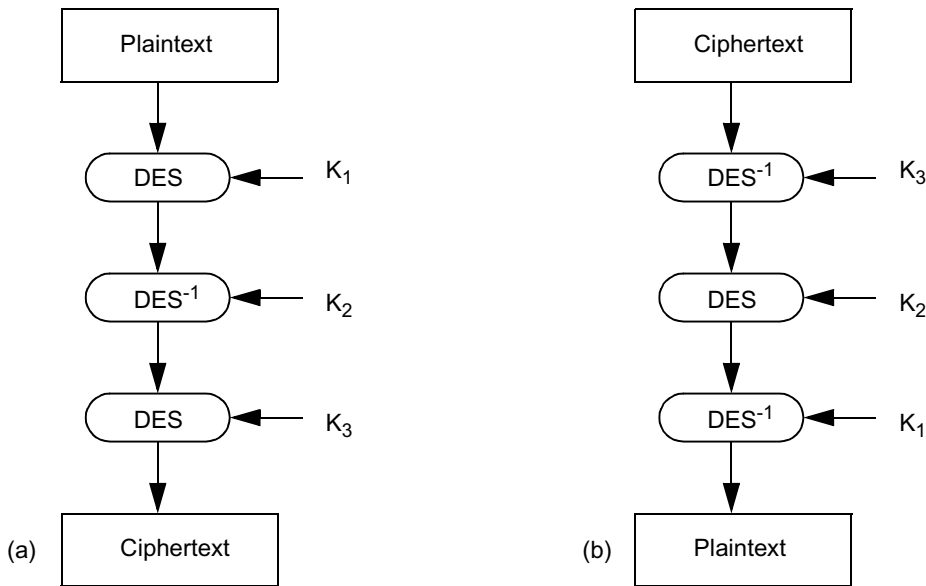
This library is described in more detail in the document **Data Encryption Standard (DES) Library**.

7.3.2 Triple DES Library

The Data Encryption Standard core used in Triple DES (3DES) conforms to the Federal Information Processing Standards Publication 46-2, (FIPS PUB 46-2).

Any secure communication/storage system requires converting data to be transmitted into an unintelligible form called ciphertext, and reconvert the data to its original form, called plaintext. The FIPS PUB 46-2 standard describing DES is the most widely-used encryption standard for these applications. A variant of this algorithm is 3DES, where DES is used three times, with different keys each time. Motorola has internally developed a single library module containing the 3DES.

[Figure 7-1](#) shows the structure of 3DES using DES. The figure itself describes the encryption and decryption processes.



NOTE:
 K¹, K² and K³ = Keys
 DES = DES Encryption
 DES⁻¹ = DES Decryption

Figure 7-1. Triple-DES: (a) Encryption and (b) Decryption

7.3.2.1 Performance

Table 7-19 details the 3DES basic performance for internal memory on the DSP56852.

Table 7-19. 3DES Memory and MIPS Requirements for Internal Memory on the DSP56852

	Program Memory (words)	Data ROM (words)	Software Stack Per Channel (words)	Data RAM Per Channel (words)	MIPS
Library allocates memory [includes <i>triDesCreate()</i> and <i>triDesDestroy()</i>]	2033	742	Encrypt 41 Decrypt 41	1012	N/A
User application allocates memory (does not include <i>triDesCreate()</i> and <i>triDesDestroy()</i>)	1572	742	Encrypt 41 Decrypt 41	1012	N/A

Note: Data RAM figures in **Table 7-19** do not include the gaps (unused memory) that will be introduced due to circular buffers. These buffers take advantage of the modulo buffer indexing feature of the Motorola DSP architecture which has strict memory alignment requirements.

Table 7-20 shows the 3DES cycle count for internal memory on the DSP56852.

Table 7-20. 3DES Cycle Count for Internal Memory on the DSP56852

Module	Instruction Cycle Count
Initialization	46726
Encryption	340431
Decryption	339651

Instruction Cycle Count = Oscillator Clock Cycles

Notes:

- The cycle counts in **Table 7-20** are obtained for an input block (data to be encrypted or decrypted) of size 128 characters

This library is described in more detail in the document **Triple Data Encryption Standard (3DES) Library**.

7.3.3 RSA Library

Discrete exponentiation has been employed in a different way by Rivest, Shamir and Adleman (RSA) to produce a public key cryptosystem. They make use of the fact that finding large prime numbers is computationally easy, but that factoring the product of two such numbers appears to be computationally infeasible.

User A selects two large prime numbers at random, p and q , and multiplies them together to obtain a number, n . The number n is made public, but the factors p and q are kept secret. Using p and q , User A can compute the Euler's Totient function:

$\Phi(n)$ (the number of integers less than n and relatively prime to n as)

$$\Phi(n) = (p-1)(q-1)$$

User A then chooses another number, e , at random from the interval 2 through $\Phi(n)$, such that e and $\Phi(n)$ are relatively prime. This number is also made public. Enciphering is carried out on each block as m , using the public information e and n as:

$$c = m^e \pmod{n}$$

In this example, c represents the ciphertext. Using the secret number $\Phi(n)$, User A can easily calculate a number, d , such that:

$$(e \cdot d) \pmod{\Phi(n)} = 1$$

If e has a common factor with $\phi(n)$ then d does not exist. The user 'B' receives c and computes $c^d \pmod{n}$ which is equivalent to:

$$m \cdot \exp(k\phi(n)+1) \pmod{n}$$

for some k . From Euler's theorem, $x \cdot \exp(\phi(n)) \pmod{n} = 1$, if x and n are relatively prime. Also, $x \cdot \exp(k\phi(n)+1) \pmod{n} = x$, for any integer x lying between 0 and $n-1$, where $n=pq$. Hence, User B gets the original message, m , by exponentiating c with d and finding modulo with respect to n .

Similarly, we can explain the signing process with the following equations:

$$m^d \pmod{n} = s$$

$$m = s^e \pmod{n}$$

7.3.3.1 Performance

Table 7-21 details the basic performance of RSA for internal memory on the DSP56852.

Table 7-21. RSA Memory and MIPS Requirements for Internal Memory on the DSP56852

	Program Memory (words)	Data ROM (words)	Software Stack Per Channel (words)	Data RAM Per Instance (words)	MIPS
Library allocates memory [includes <i>rsaEncCreate()</i> , <i>rsaEncDestroy()</i> , <i>rsaDecCreate()</i> and <i>rsaDecDestroy()</i>]	2652	0	Encrypt 20 Decrypt 18	$153 + n \cdot 26$ n = length of modulo buffer	N/A
User application allocates memory [does not include <i>rsaEncCreate()</i> , <i>rsaEncDestroy()</i> , <i>rsaDecCreate()</i> and <i>rsaDecDestroy()</i>]	2350	0	Encrypt 20 Decrypt 18	$153 + n \cdot 26$ n = length of modulo buffer	N/A

Table 7-22 shows the cycle count for RSA on the DSP56852, when data memory is internal.

Table 7-22. RSA Cycle Count for Internal Memory on the DSP56852

Module	Instruction Clock Cycles
Initialization	16824
Encryption	8401626
Decryption	8401626

Instruction Cycle Count = Oscillator Clock Cycles

Note: The cycle counts in [Table 7-22](#) are obtained for 512 bit modulo (or 32 words)

This library is described in more detail in the document **RSA Library**.

7.4 Telephony Libraries

Telephony libraries include algorithms that are used in telephone applications.

7.4.1 Acoustic Echo Canceller Library

Acoustic echo cancellers (AECs) are voice-operated devices which eliminate acoustic echoes and protect the communication from howling due to acoustic feedback from loudspeaker to microphone. AECs are placed in audio terminals on the customer premises.

Many applications, such as full-duplex speaker phones and mobile telephones, require AECs with high performance. In a speaker-mic telephone system, a part of the speaker output gets picked up by the microphone, either directly or indirectly, causing annoying echoes heard by the far-end telephone user. AECs circumvent these echoes.

AECs perform echo cancellation by estimating the signal fed back to the microphone from the speaker, then subtracting it from the microphone input. Room impulse response must also be estimated and is approximated by a linear transversal filter. The coefficients of this filter constitute the room impulse response. The existing echo cancellers use Normalized Mean Squares (NLMS) algorithms for estimating the transversal filter coefficients. The identification, or adaptation, of the transversal filter coefficients is possible only in the absence of near-end speech. Also, when there is a near-end signal present, the estimation of the filter must be frozen; a double talk detection algorithm provides this functionality.

Some of the parameters for evaluating the performance of the AEC are:

- **ERLE** - Echo Return Loss Enhancement is the ratio of the reference input to the residual echo expressed in dBm. The larger the ERLE, the better the performance of the AEC. Convergence of the adaptive algorithm determines the ERLE.
- **Rate of Convergence** - Acoustic echo is characterized by changing echo paths. Rate of convergence of the adaptive algorithm determines how fast these changes can be tracked. The faster the tracking, the better the performance of the AEC.

The basic requirements for the echo cancellers are:

- Rapid convergence
- Low echo return level during single talk
- Low divergence during double talk

7.4.1.1 Performance

[Table 7-23](#) shows Memory and MIPS requirements for the DSP56852.

Table 7-23. AEC Memory and MIPS Requirements for Internal Memory on the DSP56852

	Program Memory (words)	Data ROM (words)	Software Stack Per Channel (words)	Data RAM Per Channel (words)	MIPS
Library allocates memory [includes <i>aecCreate()</i> and <i>aecDestroy()</i>]	1103	0	27	57 + 3N N = Filter length (or echo tail length in taps) Multiple channels can exist	$(673 + 2N)F_s \times 10^{-6}$ N = Filter length (or echo tail length in taps) F _s = Sampling Frequency in Hz
User application allocates memory [does not include <i>aecCreate()</i> and <i>aecDestroy()</i>]	880	0	27	57 + 3N N = Filter length (or echo tail length in taps) Multiple channels can exist	$(673 + 2N)F_s \times 10^{-6}$ N = Filter length (or echo tail length in taps) F _s = Sampling Frequency in Hz

Notes:

- Memory resources in [Table 7-23](#) include the gaps (unused memory) that are introduced due to circular buffers. These buffers take advantage of the modulo buffer indexing feature of Motorola's DSP architecture, which has strict memory alignment requirements.
- [Table 7-23](#) does not include memory requirements for the input and output buffers and the structures which must be passed to the calling modules.

A test that verifies the operation of the AEC library is described in [Chapter 8](#).

This library is described in more detail in the document **Acoustic Echo Canceller Library**.

7.4.2 Automatic Gain Control (AGC) Library

[Table 7-24](#) details basic performance of Automatic Gain Control on the DSP56852 for Small Memory Model.

Table 7-24. AGC Memory and MIPS Requirements for Internal Memory on the DSP56852 for Small Memory Model

	Program Memory (words)	Data ROM (words)	Software Stack Per Channel (words)	Data RAM Per Channel (words)	MIPS at 8000Hz Sampling Rate
Library allocates memory [includes <i>agcCreate()</i> and <i>agcDestroy()</i>]	117	0	13	4	0.34
User application allocates memory [does not include <i>agcCreate()</i> and <i>agcDestroy()</i>]	87	0	13	4	0.34

Table 7-25 details basic performance of Automatic Gain Control on the DSP56852 for Large Memory Model.

Table 7-25. AGC Memory and MIPS Requirements for Internal Memory on the DSP56852 for Large Memory Model

	Program Memory (words)	Data ROM (words)	Software Stack Per Channel (words)	Data RAM Per Channel (words)	MIPS at 8000Hz Sampling Rate
Library allocates memory [includes <i>agcCreate()</i> and <i>agcDestroy()</i>]	117	0	13	4	0.34
User application allocates memory [does not include <i>agcCreate()</i> and <i>agcDestroy()</i>]	87	0	13	4	0.34

Notes:

- The MIPS numbers quoted in **Table 7-24** and **Table 7-25** are measured using **100 input samples per call**.
- For an input buffer length of “N” (N samples in the input buffer), MIPS is calculated with the following formula. Sampling rate is assumed to be **8000Hz**.

$$\frac{8000[43 + N \times 42]}{N \times 10^6}$$

This library is described in more detail in the document **AGC Library**.

7.4.3 CAS Detect Library

To transmit data in the off-hook mode, a stable call must be interrupted and a clear, voice-free channel established. The Stored Program Controlled-switching System (SPCS) initiates the process by muting transmission to and from the far end and transmitting the alerting signal to the service customer. The alerting signal consists of a Subscriber Alerting Signal (SAS) and a Customer Premises Equipment Alerting Signal (CAS). The SAS alerts the subscriber that there are new calls waiting.

The CAS is a machine-detectable signal used to alert the Customer Premises Equipment (CPE) to prepare for data reception. Typically, data transmission is the transmission of the Caller ID signal between the central telephone office and the customer's equipment. Before a data transmission starts, the central office sends the CAS to the CPE. When the CPE detects the CAS, the CPE is muted and returns a Dual Tone Multiple Frequency-D (DTMF) tone to the Central Office as acknowledgement. The data transmission begins only after SPCS detects the DTMF-D tone sent by the CPE. The electrical characteristics of CAS are detailed in [Table 7-26](#).

Table 7-26. CAS Electrical Characteristics

Frequency Limits	Lower Tone: 2130 Hz +/- 0.5% Upper Tone: 2750 Hz +/- 0.5%
Dynamic Range	-32 to -14 dBm per tone
Twist or Power Differential Within Dynamic Range	0 to 6 dB between tones
Tone Duration at CPE	75 to 85 ms

Note: Signal levels are referenced at 600 ohm termination at the CPE tip and ring interface

7.4.3.1 Performance

[Table 7-27](#) shows CAS Detect performance for internal memory on the DSP56852.

Table 7-27. CAS Detect Memory and MIPS Requirements for Internal Memory on the DSP56852

	Program Memory (words)	Data ROM (words)	Software Stack Per Channel (words)	Data RAM Per Channel (words)	MIPS at 8000Hz Sampling Rate
Library allocates memory [includes <i>casDetectCreate()</i> and <i>casDetectDestroy()</i>]	1413	Internal/External = 54 Internal = 28	27	400	1.328
User application allocates memory [does not include <i>casDetectCreate()</i> and <i>casDetectDestroy()</i>]	1325	82	27	400	1.328

Notes:

- **Table 7-27** does not include memory requirements for the input and output buffers or for the structures which are to be passed to the calling modules.
- MIPS in **Table 7-27** are for an input buffer length of **80** samples.
- For an input buffer length of “N” (N samples in the input buffer, where **N ≤ 80**), MIPS is calculated with the following formula. Sampling rate is assumed to be **8000Hz**.

$$\frac{8000[11843 + N \times 18]}{N \times 10^6}$$

This library is described in more detail in the document **Customer Premises Equipment Alerting Signal (CAS) Library**.

7.4.4 Call Progress Tones Detect Library

The detection of Call Progress Tones (CPT) is used in the telephone network. The aim is to detect a signal comprised of two frequencies that represent a specific tone. The present CPT library complies with the North American Telecom standards.

CPT is a set of tones which are used to indicate the current status of a call. A computer system that has a DSP and is equipped for phone-line communications can use the DSP for generating and decoding CPT signals, and for handling typical DSP functions.

Each tone consists of two frequencies and a specific temporal pattern, ON & OFF periods.

The specifications of each tone are listed in **Table 7-28**.

Table 7-28. Tone Specifications

Signal Name	Frequency (Hz)	Temporal Pattern	Amplitude
Dial Tone	350+440 (+/- 0.7%)	Steady On	-29 to -10dBm
Message Waiting Indicator Tone	350+440 (+/- 0.7%)	10 Bursts (0.1Seconds On; 0.1 Seconds Off) then steady On	-29 to -10dBm
Recall Dial Tone/Confirmation Tone	350+440 (+/- 0.7%)	3 Bursts (0.1 Seconds On; 0.1 Seconds Off) then steady On	-29 to -10dBm
Line Busy	480+620 (+/- 1.5%)	0.5 Seconds On, 0.5 Seconds Off; repeating	-52 to -21dBm

Table 7-28. Tone Specifications

Signal Name	Frequency (Hz)	Temporal Pattern	Amplitude
Reorder	480+620 (+/- 1.5%)	0.25 Seconds On, 0.25 Seconds Off; repeating	-52 to -21dBm
Audible Ringing	440+480 (+/- 1.6%)	2 Seconds On, 4 Seconds Off; repeating	-47 to -16dBm

7.4.4.1 Performance

Table 7-29 details CPT memory and MIPS requirements for internal memory on the DSP56852.

Table 7-29. CPT Memory and MIPS Requirements for Internal Memory on the DSP56852

	Program Memory (words)	Data ROM (words)	Software Stack Per Channel (words)	Data RAM Per Channel (words)	MIPS at 8000Hz Sampling Rate
Library allocates memory [includes <i>CPTDetCreate()</i> and <i>CPTDetDestroy()</i>]	1214	6	23	142	0.55
User application allocates memory [does not include <i>CPTDetCreate()</i> and <i>CPTDetDestroy()</i>]	1078	6	23	142	0.55

Notes:

- Figures in **Table 7-29** do not include the gaps (unused memory) that will be introduced due to circular buffers.
- MIPS in **Table 7-29** are for an input buffer length of **160** samples.
- For an input buffer length of “N” (N samples in the input buffer, where $N \leq 160$), MIPS are calculated with the following formula. Sampling rate is assumed to be **8000Hz**.

$$\frac{8000[5383 + N \times 35]}{N \times 10^6}$$

This library is described in more detail in the document **CPT Detection Library**.

7.4.5 Caller ID Library

This algorithm implements the GR-CORE-30 standard, which is a Bellcore standard to receive the on-hook/off-hook data transmission of Caller ID information.

7.4.5.1 Performance

Table 7-30 details basic performance information for Caller ID on the DSP56852.

Table 7-30. Caller ID Memory and MIPS Requirements for Internal Memory on the DSP56852

	Program Memory (words)	Data ROM (words)	Software Stack Per Channel (words)	Data RAM Per Channel (words)	MIPS at 8000Hz Sampling Rate
Library allocates memory [includes <i>CallerIDCreate()</i> and <i>callerIDDestroy()</i>]	1372	28	22	393	1.181
User application allocates memory [does not include <i>CallerIDCreate()</i> and <i>callerIDDestroy()</i>]	1096	28	22	393	1.181

Notes:

- Data RAM figures in **Table 7-30** do not include the gaps (unused memory) that will be introduced due to circular buffers. These buffers take advantage of the modulo buffer indexing feature of the Motorola DSP architecture, which has strict memory alignment requirements.
- MIPS in **Table 7-30** are for an input buffer length of **20** samples.
- For an input buffer length of “N” (N samples in the input buffer, where $N \leq 20$), MIPS are calculated with the following formula. Sampling rate is assumed to be **8000HZ**.

$$\frac{8000[2573 + N \times 19]}{N \times 10^6}$$

This library is described in more detail in the document **Caller ID Library**.

7.4.6 Common Tone Generation (CTG) Library

The CTG library is capable of generating any tone with any number of simultaneous frequencies. These frequencies can have different starting and ending times and can be repeated for a varying number of cycles.

7.4.6.1 Performance

Table 7-31 details basic performance information for CTG on the DSP56852 for Small Memory Model.

Table 7-31. CTG Memory and MIPS Requirements for Internal Memory on the DSP56852 for Small Memory Model

	Program Memory (words)	Data ROM (words)	Software Stack Per Channel (words)	Data RAM Per Channel (words)	MIPS at 8000Hz Sampling Rate
Library allocates memory [includes <i>ctgCreate()</i> and <i>ctgDestroy()</i>]	768	0	12	20 + 16 * (number of frequencies)	2.95
User application allocates memory [does not include <i>ctgCreate()</i> and <i>ctgDestroy()</i>]	568	0	12	20 + 16 * (number of frequencies)	2.95

Table 7-32 details basic performance information for CTG on the DSP56852 for Large Memory Model.

Table 7-32. CTG Memory and MIPS Requirements for Internal Memory on the DSP56852 for Large Memory Model

	Program Memory (words)	Data ROM (words)	Software Stack Per Channel (words)	Data RAM Per Channel (words)	MIPS at 8000Hz Sampling Rate
Library allocates memory [includes <i>ctgCreate()</i> and <i>ctgDestroy()</i>]	768	0	12	22 + 20* (number of frequencies)	2.95
User application allocates memory [does not include <i>ctgCreate()</i> and <i>ctgDestroy()</i>]	568	0	12	22 + 20* (number of frequencies)	2.95

Notes:

- **Table 7-31** and **Table 7-32** does not include memory requirements for the input and output buffers or for the structures which are to be passed to the calling modules.
- MIPS in **Table 7-31** and **Table 7-32** are for an input buffer length of **16** samples. The MIPS are calculated for call progress tones, which involve two simultaneous frequencies.

This library is described in more detail in the document **Common Tone Generation Library**.

7.4.7 DTMF Detection Library

The detection of Dual Tone Multiple Frequency (DTMF) signals (also known as Touch Tone signals) is used in the telephone network. The aim is to detect a signal that is comprised of two frequencies that represent a digit on the telephone keyboard.

7.4.7.1 Performance

The DTMF Detection memory and MIPS requirements for internal memory on the DSP56852 are shown in [Table 7-33](#).

Table 7-33. DTMF Detection Memory and MIPS for Internal Memory on the DSP56852

	Program Memory (words)	Data ROM (words)	Software Stack Per Channel (words)	Data RAM Per Channel (words)	MIPS at 8000Hz Sampling Rate
Library allocates memory [includes <i>DTMFDetCreate()</i> and <i>DTMFDetDestroy()</i>]	1048	48	29	186	0.659
User application allocates memory [includes <i>DTMFDetCreate()</i> and <i>DTMFDetDestroy()</i>]	917	48	29	186	0.659

Notes:

- Data memory figures in [Table 7-33](#) do not include the gaps (unused memory) that will be introduced due to circular buffers. These buffers take advantage of the modulo buffer indexing feature of the Motorola DSP architecture which has strict memory alignment requirements.
- MIPS in [Table 7-33](#) are for an input buffer length of **80** samples.
- For an input buffer length of “N” (N samples in the input buffer, where $N \leq 80$), MIPS are calculated with the following formula. Sampling rate is assumed to be **8000Hz**.

$$\frac{8000[5230 + N \times 17]}{N \times 10^6}$$

A test that verifies the operation of the DTMF Detection library is described in [Chapter 8](#).

This library is described in more detail in the document **DTMF Detection Library**.

7.4.8 DTMF Generation Library

The generation of Dual Tone Multiple Frequency (DTMF) signals (also known as Touch Tone signals) is used in the telephone network. The aim is to generate a signal that is comprised of two frequencies that represent a digit on the telephone keyboard.

7.4.8.1 Performance

Table 7-34 details basic performance features of the DTMF Generation Library for internal memory on the DSP56852 for Small Memory Model.

Table 7-34. MIPS and Memory Requirements for Internal Memory on the DSP56852 for Small Memory Model

	Program Memory (words)	Data ROM (words)	Software Stack Per Channel (words)	Data RAM Per Channel (words)	MIPS at 8000Hz Sampling Rate
Library allocates memory [includes <i>dtmfCreate()</i> and <i>dtmfDestroy()</i>]	273	0	10	23	0.1456
User application allocates memory [doesn't include <i>dtmfCreate()</i> and <i>dtmfDestroy()</i>]	244	0	10	23	0.1456

Table 7-35 details basic performance features of the DTMF Generation Library for internal memory on the DSP56852 for Large Memory Model.

Table 7-35. MIPS and Memory Requirements for Internal Memory on the DSP56852 for Large Memory Model

	Program Memory (words)	Data ROM (words)	Software Stack Per Channel (words)	Data RAM Per Channel (words)	MIPS at 8000Hz Sampling Rate
Library allocates memory [includes <i>dtmfCreate()</i> and <i>dtmfDestroy()</i>]	273	0	10	23	0.1456
User application allocates memory [doesn't include <i>dtmfCreate()</i> and <i>dtmfDestroy()</i>]	244	0	10	23	0.1456

Notes:

- MIPS in **Table 7-34** and **Table 7-35** are for a input buffer length of **40** samples.

- For an input buffer length of “N” (N samples in the input buffer, where N ≤ 40), MIPS are calculated with the following formula. Sampling rate is assumed to be 8000Hz.

$$\frac{8000[48 + N \times 17]}{N \times 10^6}$$

A test that verifies the operation of the DTMF Generation library is described in [Chapter 8](#).

This library is described in more detail in the document **DTMF Generation Library**.

7.4.9 G.165 Library

G.165-based echo cancellers conform to the ITU-T Recommendation (Previously “CCITT Recommendation”) G.165.

Echo cancellers are voice-operated devices placed in the 4-wire portion of the circuit. They are used to reduce the echo by subtracting the echo estimate from the circuit echo. In this recommendation, the echo cancellers are assumed to be “half” echo cancellers; i.e., those in which cancellation takes place only in the send path, due to signals present in the receive path of the circuit.

7.4.9.1 Performance

Table 7-36 details basic performance of the G.165 Echo Canceller for internal memory on the DSP56852.

Table 7-36. G.165 Memory and MIPS Requirements for Internal Memory on the DSP56852

	Program Memory (words)	Data ROM (words)	Software Stack Per Channel (words)	Data RAM Per Channel (words)	MIPS at 8000Hz Sampling Rate (echo tail (msec))
Library allocates memory [includes <i>g165Create()</i> and <i>g165Destroy()</i>]	2211	234	20	282 + 3 * Echo-Span (number of taps)	5.2429 (8) 6.0077(16) 7.2685 (24) 8.5613 (32) 9.8541 (40)
User application allocates memory [does not include <i>g165Create</i> and <i>g165Destroy()</i>]	1716	234	20	282 + 3 * Echo-Span (number of taps)	5.2429 (8) 6.0077(16) 7.2685 (24) 8.5613 (32) 9.8541 (40)

Notes:

- Data RAM figures in [Table 7-36](#) do not include the gaps (unused memory) that will be introduced due to circular buffers. These buffers take advantage of the modulo buffer indexing feature of

Motorola's DSP architecture, which has strict memory alignment requirements. Data ROM has a circular buffer of length 4. Data RAM has circular buffers of length 3, 3, 3, 3, 6, 6, EchoSpan (number of filter taps at an 8KHz sampling rate).

- MIPS in [Table 7-36](#) are for **80 input samples per call**.

A test that verifies the operation of the G.165 library is described in [Chapter 8](#).

This library is described in more detail in the document **G.165 Library**.

7.4.10 G.168 Library

G.168-based echo cancellers conform to the ITU-T Recommendation (Previously "CCITT Recommendation") G.168.

Echo cancellers are voice-operated devices placed in the 4-wire portion of the circuit. They are used for reducing the echo by subtracting the echo estimate from the circuit echo. In this recommendation, the echo cancellers are assumed to be "half" echo cancellers, i.e., those in which cancellation takes place only in the send path, due to signals present in the receive path of the circuit.

7.4.10.1 Performance

[Table 7-37](#) details basic performance of the G.168 Echo Canceller on the DSP56852.

Table 7-37. G.168 Memory and MIPS Requirements for Internal Memory on the DSP56852

	Program Memory (words)	Data ROM (words)	Software Stack Per Channel (words)	Data RAM Per Channel (words)	MIPS at 8000Hz Sampling Rate (echo tail (msec))
Library allocates memory [includes <i>g168Create()</i> and <i>g168Destroy()</i>]	2211	234	20	282 + 3 * Echo-Span (number of taps)	5.2429 (8) 6.0077(16) 7.2685 (24) 8.5613 (32) 9.8541 (40) 11.1469 (48) 12.4397 (56) 13.7325 (64)
User application allocates memory [does not include <i>g168Create()</i> and <i>g168Destroy()</i>]	1716	234	20	282 + 3 * Echo-Span (number of taps)	5.2429 (8) 6.0077(16) 7.2685 (24) 8.5613 (32) 9.8541 (40) 11.1469 (48) 12.4397 (56) 13.7325 (64)

Notes:

- Data RAM figures in [Table 7-37](#) do not include the gaps (unused memory) that will be introduced due to circular buffers. These buffers take advantage of the modulo buffer indexing feature of Motorola's DSP architecture, which has strict memory alignment requirements. Data ROM has a circular buffer of length 4. Data RAM has circular buffers of length 3, 3, 3, 3, 6, 6, EchoSpan (number of filter taps at an 8KHz sampling rate).
- MIPS in [Table 7-37](#) are for **80 input samples per call**.

This library is described in more detail in the document **G.168 Library**.

7.4.11 G.711 Library

G.711-based log-PCM conforms to the ITU-T Recommendation G.711, previously "CCITT Recommendation".

The characteristics mentioned in the standard are recommended for encoding (A-law or μ law) voice frequency signals to yield a bit rate of 64 kbits/s at an 8000Hz sampling rate. G.711 is a PCM Encoding and Decoding (log-PCM) Library.

7.4.11.1 Performance

[Table 7-38](#) outlines basic performance of the G.711 library for internal memory on the DSP56852 for Small Memory Model.

Table 7-38. G.711 Memory and MIPS Requirements for Internal Memory on the DSP56852 for Small Memory Model

Library Functions	Program Memory (words)	Data ROM (words)	Software Stack Per Channel (words)	Data RAM Per Channel (words)	MIPS at 8000Hz Sampling Rate
Linear2alaw	43	0	16	N/A	0.256
Linear2ulaw	41	0	16	N/A	0.232
alaw2linear	42	0	10	N/A	0.24
ulaw2linear	38	0	8	N/A	0.216
alaw2ulaw	30	64	6	N/A	0.12
ulaw2alaw	31	64	6	N/A	0.128

[Table 7-39](#) outlines basic performance of the G.711 library for internal memory on the DSP56852 for Large Memory Model.

Table 7-39. G.711 Memory and MIPS Requirements for Internal Memory on the DSP56852 for Large Memory Model

Library Functions	Program Memory (words)	Data ROM (words)	Software Stack Per Channel (words)	Data RAM Per Channel (words)	MIPS at 8000Hz Sampling Rate
Linear2alaw	43	0	16	N/A	0.256
Linear2ulaw	41	0	16	N/A	0.232
alaw2linear	42	0	10	N/A	0.24
ulaw2linear	38	0	8	N/A	0.216
alaw2ulaw	30	64	6	N/A	0.12
ulaw2alaw	31	64	6	N/A	0.128

Notes:

- [Table 7-38](#) and [Table 7-39](#) does not include memory requirements for the input and output buffers.
- Data RAM figures in [Table 7-38](#) and [Table 7-39](#) do not include the gaps (unused memory) that will be introduced due to circular buffers. These buffers take advantage of the modulo buffer indexing feature of Motorola's DSP architecture, which has strict memory alignment requirements.

A test that verifies the operation of the G.711 library is described in [Chapter 8](#).

This library is described in more detail in the document [G.711 Library](#).

7.4.12 G.726 Library

The ITU-T Recommendation G.726 Adaptive Differential Pulse Code Modulation Speech Codec converts a 64 Kbits/s A-Law or μ Law pulse code modulation (PCM) channel to and from 40, 32, 24, 16 Kbits/s PCM stream using an ADPCM transcoding technique.

7.4.12.1 Performance

Memory requirements for the G.726 ADPCM Codec library for internal memory on the DSP56852 are shown in [Table 7-40](#).

Table 7-40. G.726 Memory and MIPS Requirements for Internal Memory on the DSP56852

		Program Memory (words)	Data ROM (words)	Software Stack Per Channel (words)	Data RAM Per Channel (words)	MIPS at 8000Hz Sampling Rate
Library allocates memory [includes <i>G726EncCreate()</i> <i>G726EncDestroy()</i> <i>G726DecCreate()</i> and <i>G726DecDestroy()</i>]	Encoder	1060	220	16	73	7.272 @40kbps 7.272@32kbps 7.208@24kbps 7.176@16kbps
	Decoder	1216	220	16	73	8.304 @40kbps 8.304@32kbps 8.24@24kbps 8.208@16kbps
	Encoder + Decoder	1311	220	16	146	15.576@40kbps 15.576@32kbps 15.448@24kbps 15.384@16kbps
User application allocates memory [does not include <i>G726EncCreate()</i> <i>G726EncDestroy()</i> <i>G726DecCreate()</i> and <i>G726DecDestroy()</i>]	Encoder	1110	220	16	73	7.272 @40kbps 7.272@32kbps 7.208@24kbps 7.176@16kbps
	Decoder	1141	220	16	73	8.304 @40kbps 8.304@32kbps 8.24@24kbps 8.208@16kbps
	Encoder + Decoder	1205	220	16	146	15.576@40kbps 15.576@32kbps 15.448@24kbps 15.384@16kbps

Note: Data RAM figures in [Table 7-40](#) do not include the gaps (unused memory) that will be introduced due to circular buffers. These buffers take advantage of the modulo buffer indexing feature of Motorola's DSP architecture, which has strict memory alignment requirements.

A test that verifies the operation of the G.726 library is described in [Chapter 8](#).

This library is described in more detail in the document **G.726 Library**.

7.4.13 MFCR2 Detection Library

The MFCR2 signal consists of two frequencies, taken from either the set of six forward frequencies or from the set of six backward frequencies. The MFCR2 receiver detects any valid MFCR2 signal and outputs the numerical value corresponding to that signal.

7.4.13.1 Performance

The MFCR2 Detection memory and MIPS requirements for internal memory on the DSP56852 are shown in [Table 7-41](#).

Table 7-41. MFCR2 Detection Memory and MIPS for Internal Memory on the DSP56852

	Program Memory (words)	Data ROM (words)	Software Stack Per Channel (words)	Data RAM Per Channel (words)	MIPS at 8000Hz Sampling Rate
Library allocates memory [includes <i>MFCR2DetectCreate()</i> and <i>MFCR2DetectDestroy()</i>]	986	250	5	216	4.1
User application allocates memory [does not include <i>MFCR2DetectCreate()</i> and <i>MFCR2DetectDestroy()</i>]	816	250	5	216	4.1

Notes:

- Data memory figures in [Table 7-41](#) do not include the gaps (unused memory) that will be introduced due to circular buffers. These buffers take advantage of the modulo buffer indexing feature of the Motorola DSP architecture, which has strict memory alignment requirements.
- MIPS in [Table 7-41](#) is for an input buffer length of 32 samples.

A test that verifies the operation of the MFCR2 Detection library is described in [Chapter 8](#).

This library is described in more detail in the document **MFCR2 Detection Library**.

7.4.14 Noise Suppression Library

[Table 7-42](#) details basic performance of Noise Suppression (NS) on the DSP56852.

Table 7-42. NS Memory and MIPS Requirements for Internal Memory on the DSP56852

	Program Memory (words)	Data ROM (words)	Software Stack Per Channel (words)	Data RAM Per Channel (words)	MIPS at 8000Hz Sampling Rate
Library allocates memory [includes <i>nsCreate()</i> and <i>nsDestroy()</i>]	1724	357	40	653	3.063
User application allocates memory [does not include <i>nsCreate()</i> and <i>nsDestroy()</i>]	1294	357	40	653	3.063

Notes:

- Data RAM figures in [Table 7-42](#) do not include the gaps (unused memory) that will be introduced due to circular buffers. These buffers take advantage of the modulo buffer indexing feature of Motorola's DSP architecture, which has strict memory alignment requirements.
- The MIPS numbers quoted in [Table 7-42](#) are for **80 input samples per call**

This library is described in more detail in the document **Noise Suppression Library**.

7.4.15 Voice Activity Detector Library

Voice Activity Detection (VAD) saves bandwidth by sending packets only when speech is present. The effectiveness of the voice activity detector can be determined by measuring these factors:

- **Front-end clipping**, which is the amount of time it takes a voice activity detector to detect speech and begin transmitting audio
- **Holdover time**, which is the amount of time needed to determine that speech is no longer present and to stop transmitting background audio

The VAD detects voice activity and activates or deactivates the transmission of packets to optimize bandwidth. When no activity is detected, the encoder output will not be transported across the network. Idle noise is reproduced by the remote end when there is no voice activity, so the remote user will not believe that the line has gone dead.

7.4.15.1 Performance

[Table 7-43](#) details the basic performance of VAD for internal memory on the DSP56852 for Small Memory Model.

Table 7-43. VAD Memory and MIPS Requirements for Internal Memory on the DSP56852 for Small Memory Model

	Program Memory (words)	Data ROM (words)	Software Stack Per Channel (words)	Data RAM Per Channel (words)	MIPS at 8000Hz Sampling Rate
Library allocates memory [includes <i>vadCreate()</i> and <i>vadDestroy()</i>]	793	0	38	84	1.645
User application allocates memory (does not include <i>vadCreate()</i> and <i>vadDestroy()</i>)	708	0	38	84	1.645

Table 7-44 details the basic performance of VAD for internal memory on the DSP56852 for Large Memory Model.

Table 7-44. VAD Memory and MIPS Requirements for Internal Memory on the DSP56852 for Large Memory Model

	Program Memory (words)	Data ROM (words)	Software Stack Per Channel (words)	Data RAM Per Channel (words)	MIPS at 8000Hz Sampling Rate
Library allocates memory [includes <i>vadCreate()</i> and <i>vadDestroy()</i>]	793	0	38	88	1.645
User application allocates memory (does not include <i>vadCreate()</i> and <i>vadDestroy()</i>)	708	0	38	88	1.645

Notes:

- **Table 7-43** and **Table 7-44** does not include memory requirement for the input and output buffers, nor for the structures which must be passed to the calling modules.
- Data RAM figures in **Table 7-43** and **Table 7-44** do not include the gaps (unused memory) that will be introduced due to circular buffers. These buffers take advantage of the modulo buffer indexing feature of the Motorola DSP architecture, which has strict memory alignment requirements.

This library is described in more detail in the document **Voice Activity Detector Library**.

7.5 Speech Libraries

Speech libraries include algorithms that are used speech recognition and generation.

7.5.1 VRLite-1 Library

Many products, such as a mobile phone, require a voice recognition system to operate the phone through voice. VRLite-1 provides a solution for this requirement. VRLite-1 is a memory-optimized, isolated-word, speaker-dependent speech recognition system. This means that the system must be trained to the voice of a particular user and that it can recognize only isolated words. For example, if the user trained the words “call” and “Bob”, the algorithm can recognize both “call” and “Bob”, but not the phrase “call Bob”.

7.5.1.1 Performance

Table 7-45 shows VRLite-1 Memory and MIPS figures for internal memory on the DSP56852.

Table 7-45. VRLite-1 Memory and MIPS Requirements for Internal Memory on the DSP56852

	Program Memory (words)	Data ROM (words)	Software Stack Per Channel (words)	Data RAM Per Channel (words)	MIPS
Library allocates memory [includes <i>vrlite1Create()</i> and <i>vrlite1Destroy()</i>]	6958	540	51	4670 (only one channel is possible)	13.5 (front end only, which is real time)
User application allocates memory [does not include <i>vrlite1Create()</i> and <i>vrlite1Destroy()</i>]	6859	540	51	4670 (only one channel is possible)	13.5 (front end only, which is real time)

Table 7-46 shows the Instruction Cycle Count figures for internal memory on the DSP56852 (which equals Oscillator Cycle Count) for Training, Rejection Analysis, and Recognition, non-real-time modules.

Table 7-46. Instruction Cycle Counts for Internal Memory on the DSP56852

Module	Instruction Cycle Count
Training of a word	1000336
Rejection Analysis per trained word in the vocabulary	318114
Recognition per trained word in the vocabulary	442708

Notes:

- Memory resources in **Table 7-46** include the gaps (unused memory) that are introduced due to circular buffers. These buffers take advantage of the modulo buffer indexing feature of the Motorola DSP architecture, which has strict memory alignment requirements.
- Table 7-46** does not include memory requirements for the input and output buffers and the structures which must be passed to the calling modules.

This library is described in more detail in the document **VRLite-1 Library**.

7.6 Feature Phone Libraries

The Feature Phone Libraries include:

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

When combined with 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. By using some or all of these modules, several other types of telephony applications are also possible. Each module may also be used independently.

7.6.1 Generic Echo Cancellor Library

The Generic Echo Cancellor (GEC) Library implements the algorithm for a generic echo canceller in Customer Premises Equipment (CPE). The GEC, along with other modules, provides for a complete set-up for telephony applications and is used for Type 2 Caller ID and for speakerphone applications, as well as for other applications that may require echo cancellation. It works with the Full Duplex Speakerphone Library as a line echo canceller for achieving all functionalities of a natural-sounding speakerphone and can be used for echo cancellation in any other situation

7.6.1.1 Performance

Table 7-47 details the memory and MIPS requirements for the GEC Library. There are two versions of code, depending on whether the data structure and the modulo buffer used by the library are placed in internal memory or external memory, and the MIPS changes between the two versions. The MIPS also depends on the tail length chosen for the GEC. For every extra 8ms of tail length, approximately 0.768 MIPS if in Internal Memory, or approximately 1.28 MIPS if in External Memory, should be added to the MIPS displayed in **Table 7-47**. The MIPS in this table have been estimated for a tail length of 8ms for the GEC. Depending on the tail length, the Data RAM per instance changes. The buffer size shown in Table 3-1 of the **Generic Echo Cancellor Library** manual should be added to the memory requirement stated in **Table 7-47**.

Table 7-47. Generic Echo Cancellor Memory and MIPS Requirements

	Program Memory (ROM) (16 bit words)	Data Memory (ROM) (16 bit words)	Data RAM (16 bit words)	Data RAM Per Instance (16 bit words)	MIPS
GEC Library (Internal Memory)	680	NA	NA	580 + buffer ¹	10.88
GEC Library (External Memory)	680	NA	NA	580 + buffer ¹	13.00

1. See Section 1.3.2 in the **Generic Echo Cancellor Library** manual for details

7.6.2 Type 1 and 2 Telephony Parser Library

The Type 1 and 2 Telephony Parser Library is an integral part of the processing of received on-hook Caller ID and related information reception services, known as Type 1 services, as well as off-hook Caller ID and related services, known as Type 2 services.

The library extracts protocol data and stores the data into different service fields and checks error messages. For example, the Type 1 and 2 Telephony Parser Library takes a demodulated Caller ID message from the FSK processor and places the appropriate data into the date, time, and calling number fields. It also calculates the checksum, checks if there are any errors in the data, and reports any errors to the user application. Other Feature Phone Library functions must work with the Type 1 and 2 Telephony Parser Library for a user application to correctly handle an incoming Caller ID, Visual Message Waiting Indicator (VMWI), or other Type 1 or 2 telephony service messages.

7.6.2.1 Performance

The most efficient way to use the Type 1 and 2 Telephony Parser Library is to execute the library in the user application's background, since this does not consume processor MIPS. The Parser Library is a messaging processor and does not require running in real time. If the Type 1 and 2 Telephony Parser Library must run in the application's foreground, it will consume approximately 6.5 MIPS. [Table 7-48](#) details the memory and MIPS requirements for the Type 1 and 2 Telephony Parser Library.

Table 7-48. Type 1 and 2 Telephony Parser Library Memory and MIPS Requirements

	Program Memory (ROM) (16 bit words)	Data ROM (words)	Data RAM - Algorithm (16 bit words)	Data RAM Per Instance (words)	MIPS
Type 1 and 2 Telephony Parser Implementation	1.4 K	NA	550	NA	NA

7.6.3 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.

7.6.3.1 Performance

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. [Table 7-49](#) details the memory and MIPS requirements for the Full Duplex Speakerphone Library. The MIPS changes between the two versions of code. The MIPS also depends on the tail length chosen for the acoustic echo canceller. For every 8ms of tail length, approximately 0.768 MIPS if in Internal Memory, or approximately 1.28 MIPS if in External Memory, should be added to the MIPS shown in [Table 7-49](#). Depending on the tail length, the Data RAM

per instance changes. The buffer size shown in Table 3-7 of the **Full Duplex Speakerphone Library** manual should also be added to the memory requirement stated in [Table 7-49](#). The MIPS shown in [Table 7-49](#) is for a tail length of 24ms for the acoustic echo canceller.

Table 7-49. Full Duplex Speakerphone Memory and MIPS Requirements

	Program Memory (ROM) (16 bit words)	Data Memory (ROM) (16 bit words)	Data RAM (16 bit words)	Data RAM Per Instance (16 bit words)	MIPS
Full Duplex Speakerphone Library (Internal Memory)	2203	NA	NA	603 + Buffer ¹	12.96
Full Duplex Speakerphone Library (External Memory)	2203	NA	NA	603 + Buffer ¹	16.80

1. See Section 1.3.2 in the **Full Duplex Speakerphone Library** manual for details

7.6.4 Type 1 Telephony Features Library

Customer Premises Equipment (CPE) using the Type 1 Telephony Features Library has the features to support existing on-hook Caller ID services, such as Calling Number Delivery and Calling Name Delivery; other existing on-hook services, such as Visual Message Waiting Indicator (VMWI) or Call Qualifier (CQ); as well as future services. Type 1 CPE supports on-hook signaling with or without power ringing and can decode data frames packaged in Single Data Message Format (SDMF) or Multiple Data Message Format (MDMF), depending on the needs of the specific telephony service supported. While this module is primarily intended to be used for Caller ID reception, other convenient features have been added to allow for simple integration into a Caller ID telephone.

7.6.4.1 Performance

[Table 7-50](#) details the memory and MIPS requirements for the Type 1 Telephony Features Library.

Table 7-50. Type 1 Telephony Features Library Memory and MIPS Requirements

	Program Memory (ROM) (16 bit words)	Data ROM (16 bit words)	Data RAM 1st Instance (16 bit words)	Data RAM Additional Instance (16 bit words)	MIPS (When all program and data is internal memory)
Type 1 Telephony Features Library	1.7 K	20	257	192	9.6



7.6.5 Type 1 and 2 Telephony Features Library

Type 2 CPE has all the features of Type 1, plus the feature of supporting off-hook Caller ID. In other words, Type 2 CPE can also process Caller ID during a call. Unlike Type 1, Type 2 CPE can receive multiple Caller IDs one at a time during a call. Type 2 CPE, therefore, has the features to support existing off-hook Caller ID services, such as Calling Identity Delivery on Call Waiting (CIDCW) and Call Waiting Deluxe (sometimes called Call Waiting with Call Disposition).

7.6.5.1 Performance

Table 7-51 details the memory and MIPS requirements for the Type 1 and 2 Telephony Features Library.

Table 7-51. Type 1 and 2 Telephony Features Memory and MIPS Requirements

	Program Memory (ROM) (16 bit words)	Data ROM (16 bit words)	Data RAM 1st Instance (16 bit words)	Data RAM Additional Instance (16 bit words)	MIPS (When all program and data is internal memory)
Type 1 and 2 Telephony Features Library	2.8 K	20	257	192	9.6

Freescale Semiconductor, Inc.

ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

Because of an order from the United States International Trade Commission, BGA-packaged product lines and part numbers indicated here currently are not available from Freescale for import or sale in the United States prior to September 2010: DSP56852VFE

Chapter 8

Library Tests

The SDK for the DSP56852 platform comes with several tests. These tests are used to verify that the libraries in the SDK are functioning correctly. The following sections describe these tests in more detail.

8.1 System Library

The system test, used to verify the system library, is found in:

...\sdk\src\dsp56852evm\nos\sys\Debug\sys.lib

8.1.1 System Test

The system test verifies the correct operation of the intrinsic functions; SDK handling of interrupts; and the SDK memory management functions.

8.1.1.1 Set-up for System Test

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**.

8.1.1.2 Procedure for System Test

- Using CodeWarrior, open:
...\nos\sys\test\testsys.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- The test results will be displayed on CodeWarrior's console

8.2 Tools Library

The tools tests, used to verify the tools library, are found in:

...\sdk\src\dsp56852evm\nos\tools\Debug\tools.lib

The tools library includes the FIFO, stack check, and debug functions of the SDK.

8.2.1 FIFO Test

The FIFO test verifies the correct operation of the FIFO functions provided by the SDK.

8.2.1.1 Set-up for FIFO Test

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**.

8.2.1.2 Procedure for FIFO Test

- Using CodeWarrior, open:
...\nos\tools\test\fifo.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- The test results will be displayed on CodeWarrior's console

8.3 DSP Functions Library

The DSP function tests are used to verify the DSP functions library, found in:

...\sdk\src\dsp56852evm\nos\signal\dspfunc\Debug\dspfunc.lib

The DSP Functions library includes the Math, Filters, RFFT, and CFFT functions of the SDK.

8.3.1 Math Tests

The math tests verify the correct operation of the math functions provided by the SDK. There are separate tests available to test 16-bit math operations and 32-bit math operations.

8.3.1.1 Set-up for Math Test

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**.

8.3.1.2 Procedures for Math Tests

To test 16-bit math operations:

- Using CodeWarrior, open:
 `...\nos\signal\dspfunc\test\math\math.mcp`
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- The test results will be displayed on CodeWarrior's console

To test 32-bit math operations:

- Using CodeWarrior, open:
 `...\nos\signal\dspfunc\test\math2\math2.mcp`
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- The test results will be displayed on CodeWarrior's console

8.3.2 Filter Test

The filter test verifies the correct operation of the filter functions provided by the SDK.

8.3.2.1 Set-up for Filter Test

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**.

8.3.2.2 Procedure for Filter Test

- Using CodeWarrior, open:
 `...\nos\signal\dspfunc\test\filters\filters.mcp`
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- The test results will be displayed on CodeWarrior's console

8.3.3 RFFT Test

The RFFT test verifies the correct operation of the RFFT functions provided by the SDK.

8.3.3.1 Set-up for RFFT Test

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**.

8.3.3.2 Procedure for RFFT Test

- Using CodeWarrior, open:
...\\nos\\signal\\dspfunc\\test\\rfft\\testrfft.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- The test results will be displayed on CodeWarrior's console

8.3.4 CFFT Test

The RFFT test verifies correct operation of the CFFT functions provided by the SDK.

8.3.4.1 Set-up for CFFT Test

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**.

8.3.4.2 Procedure for CFFT Test

- Using CodeWarrior, open:
...\\nos\\signal\\dspfunc\\test\\cfft\\testcfft.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- The test results will be displayed on CodeWarrior's console

8.4 Security Libraries

8.4.1 DES Loopback Test

The loopback test is intended to confirm the accuracy of encryption and decryption algorithms. DES can be operated in four modes:

- Cipher Feedback mode (CFB)
- Cipher Block Chaining mode (CBC)
- Output Feedback mode (OFB)
- Electronic Code Book mode (ECB)

At this time, the DES library supports only the CFB and CBC modes.

In loopback test, a certain length of plaintext is supplied to the encryption algorithm, which generates ciphertext. The ciphertext is supplied to the decryption algorithm to retrieve the decrypted text. For details about DES and its modes, refer to FIPS PUB 46-2.

8.4.1.1 Set-up for DES Loopback Test

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**.

8.4.1.2 Procedure for DES Loopback Test

- Using CodeWarrior, open:
...\\nos\\security\\des\\test\\test_des.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- The test results will be displayed on CodeWarrior's console

8.4.1.3 Triple DES Loopback Test

The Triple DES loopback test demonstrates the functionality of encryption and decryption algorithms.

In the loopback test, a length of plaintext is supplied to the encryption algorithm, which returns ciphertext. This ciphertext is supplied to the decryption algorithm to retrieve the decrypted text, which should be identical to the plaintext.

8.4.1.4 Set-up for Triple DES Loopback Test

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**.

8.4.1.5 Procedure for Triple DES Loopback Test

- Using CodeWarrior, open:
...\\nos\\security\\3des\\test_3des\\test_3des.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- The test results will be displayed on CodeWarrior's console

8.4.1.6 RSA Loopback Test

The loopback test demonstrates the functionality of encryption and decryption algorithms.

In the loopback test, a message of a certain length is supplied to the encryption algorithm, which gives the signature. This signature is supplied to the decryption algorithm to retrieve the decrypted message, which should be the same as the message.

8.4.1.7 Set-up for RSA Loopback Test

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**.

8.4.1.8 Procedure for RSA Loopback Test

- Using CodeWarrior, open:
...\\nos\\security\\rsa\\test\\test_rsa.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- The test results will be displayed on CodeWarrior's console

8.5 Telephony Libraries

8.5.1 AEC Test

The AEC test demonstrates the accuracy of the acoustic echo cancellation algorithms.

In the test for the AEC, the input signals are speech samples, sampled at 8KHz. The far-end speech samples are used by the AEC to estimate the echo and subsequently used for cancelling the echo present in the near-end speech samples.

8.5.1.1 Set-up for AEC Test

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**.

File I/O:

This test is File I/O based. Connect the serial cable between the EVM and the COM1 port of the computer.

8.5.1.2 Procedure for AEC Test

- Invoke the PC File I/O application by executing the following file:
...\\Embedded SDK\\src\\x86\\win32\\applications\\fileio\\fileio.exe
- Modify the settings to configure the File I/O application to run at 56000bps
- Using CodeWarrior, open:
...\\nos\\telephony\\aec\\test\\test_aec.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- The test results will be displayed on CodeWarrior's console
- An **interlaced** speech sample (far- and near-end speech) is read from an input file (*Inaec.in*, located in: ...\\nos\\telephony\\aec\\test\\io\\) into a temporary buffer.
 - The interlacing of samples is due to the limitation of File I/O capabilities.
 - Far- and near-end samples are then separated in the test code; for more details, see the test file.
- The echo-cancelled output is stored in the output file, *Outaec.out*, found in:
...\\nos\\telephony\\aec\\test\\io\\

- The output file is converted into *au* format via the utility *Conv2au.exe*, located in:
...\\nos\\telephony\\aec\\test\\io
- The files *far.au*, (far-end reference signal), *near.au*, (near-end input signal), and *Outaec.au* are played on speakers using Windows Media Player. All three files are found in:
...\\nos\\telephony\\aec\\test\\io
- File *far.au* has only the far-end speech signal. File *near.au* has near-end speech plus echo of far-end speech. File *Outaec.au* has the echo-cancelled near-end speech signal.

8.5.2 Automatic Gain Control (AGC) Test

The AGC test is intended to demonstrate the accuracy of the algorithm.

The input to AGC is a set of signals, (unit step, sinusoidal and speech), which is processed on a sample-by-sample basis. The input files (*sine_agc.in*, *speech_agc.in* and *unitstep_agc.in*) for the AGC are located in ...\\nos\\telephony\\agc\\test\\io. The corresponding output files with extension, ".out", will be generated by the test and stored in the same directory. These output files should be identical to the corresponding reference files provided in the same directory.

8.5.2.1 Set-up for AGC Test

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**.

File I/O:

This test is File I/O based. Connect the serial cable between the EVM and the COM1 port of the computer.

8.5.2.2 Procedure for AGC Test

- Invoke the PC File I/O application by executing the following file:
...\\Embedded SDK\\src\\x86\\win32\\applications\\fileio\\fileio.exe
- Modify the settings to configure the File I/O application to run at 56000bps
- Using CodeWarrior, open:
...\\nos\\telephony\\agc\\test\\test_agc.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- When the Automatic Gain Control test is done on the input files (*sine_agc.in*, *speech_agc.in* and *unitstep_agc.in*), the following message will appear on the CodeWarrior console: "All Output files created, manually diff them with reference files"
- The output files generated are : *sine_agc.out*, *speech_agc.out* and *unitstep_agc.out*.
- These files should be identical to the corresponding reference file (*sine_agc.ref*, *speech_agc.ref* and *unitstep_agc.ref*)

8.5.3 Caller ID Detection Test

This test is intended to verify the accuracy of Caller ID algorithms. Caller ID can be operated in two modes:

- On-hook
- Off-hook

In this test, CPFSK samples generated through MATLAB are supplied to the Caller ID Detection algorithm.

The Caller ID Software is tested for these aspects:

1. Different signal conditions
2. On-hook and Off-hook transmission

Test cases are generated for different signal conditions for both On-hook and Off-hook modes. These test cases are kept in the files *test1.in*, *test2.in*, and so on; descriptions follow:

Case 1:

Plain data, No noise, No channel shaping

This case provides the Caller ID with plain data sampled at 8KHz.

Test Cases:

On-hook = *test1.in*
Off-hook = *test2.in*

Case 2:

Data + Noise

Noise is calculated as (Signal Amplitude)/(sqrt[Noise var]).

Test Cases:

On-hook = *test3.in*
Off-hook = *test4.in*

Case 3:

Data + Noise + Channel Shaping

Channel attenuation is linearly approximated with gain at DC to be 1, and gain at 4KHz as a control parameter. The FFT of the whole data is multiplied with this straight line approximation of the channel and the IFFT gives the channel shaped data in time domain.

Test Cases:

On-hook = *test5.in*
Off-hook = *test6.in*

Case 4:

Data + Noise + Channel shaping + Initial phase offset

The initial phase shift can be between 0 and 2PI

Test Cases:

On-hook = *test7.in*

Off-hook = *test8.in*

Case 5:

Data + Noise + Channel shaping + Initial phase offset + freq offset

The frequency variation can be in the range of 10% of the carrier frequency

Test Cases:

On-hook = *test9.in*

Off-hook = *test10.in*

Case 6:

Data + Noise + Channel shaping + Initial phase offset + freq offset + N samples delay

N refers to the delay as the number of samples for the whole data

Test Cases:

On-hook = *test11.in*

Off-hook = *test12.in*

Case 7:

Data + Noise + Channel shaping + Initial phase offset + freq offset + N samples delay + Attenuation

The signal attenuation can be as high as 38dbm.

Test Cases:

On-hook = *test13.in*

Off-hook = *test14.in*

Case 8:

Data + Noise + Channel shaping + Initial phase offset + freq offset + N samples delay + Attenuation + Baud offset

The transmission baud rate can be 1200 $\pm$ 12bps

Test Cases:

On-hook = *test15.in*

Off-hook = *test16.in*

8.5.3.1 Set-up for Caller ID Detection Test

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**.

File I/O:

This test is File I/O based. Connect the serial cable between the EVM and the COM1 port of the computer.

8.5.3.2 Procedure for Caller ID Detection Test

- Invoke the PC File I/O application by executing the following file:
`...\Embedded SDK\src\x86\win32\applications\fileio\fileio.exe`
- Configure the File I/O application to run at 56000bps by modifying the settings
- Using CodeWarrior, open:
`...\nos\telephony\caller_id\test\test_callerid.mcp`
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- The test results will be displayed on CodeWarrior's console

8.5.4 CAS Detection Test

The CAS Detection test demonstrates the accuracy of the CAS Detect algorithm.

In the test for CAS Detect, the input signals are sampled at 8KHz. When a valid CAS tone is detected, further processing of the input samples halts.

8.5.4.1 Set-up for CAS Detection Test

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**.

File I/O:

This test is File I/O based. Connect the serial cable between the EVM and the COM1 port of the computer.

8.5.4.2 Procedure for CAS Detection Test

- Invoke the PC File I/O application by executing the following file:
`...\Embedded SDK\src\x86\win32\applications\fileio\fileio.exe`
- Configure the File I/O application to run at 56000bps by modifying the settings
- Using CodeWarrior, open:
`...\nos\telephony\cas_detect\test_casdetect\test_casdetect.mcp`
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- The test results will be displayed on CodeWarrior's console

8.5.5 CPT Detection Test

The CPT Detection test is intended to confirm the accuracy of the CPT Detection algorithm. The input to the algorithm is through a file called *test1x.in*; the input files for *test1x.in*, (where $x = \text{busy, dial, msg\_wait, recall, reorder and ringing}$), for running this test are taken from the directory:

```
...\nos\telephony\cpt\test_cpt\C_Sources\io
```

In this test, the input signals are sampled at 8KHz.

8.5.5.1 Set-up for CPT Detection Test

EVM Jumper Settings:

Use default settings as shown in the **DSP56852 Evaluation Module User's Manual**.

File I/O:

This test is File I/O based. Connect the serial cable between the EVM and the COM1 port of the computer.

8.5.5.2 Procedure for CPT Detection Test

- Invoke the PC File I/O application by executing the following file:
`...\Embedded SDK\src\x86\win32\applications\fileio\fileio.exe`
- Modify the settings to configure the File I/O application to run at 56000bps
- Using CodeWarrior, open:
`...\nos\telephony\cpt\test_cpt\test_cpt.mcp`
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- The test results will be displayed on CodeWarrior's console

8.5.6 Common Tone Generation (CTG) Test

The Common Tone Generation test is intended to confirm the accuracy of the CTG algorithm.

In this test for CTG, the instance variables are initialized with predefined values. The output samples are compared with values in a reference output file. The Metrowerks (MW) console displays test results: if the values match, the test was successful; otherwise, the test failed.

The reference output file is present in the folder:

```
...\nos\telephony\c\test\ref_outputs
```

In this test, the input signals are sampled at 8KHz.

8.5.6.1 Set-up for Common Tone Generation Test

EVM Jumper Settings:

Use default settings shown in the **DSP56852 Evaluation Module User's Manual**.

File I/O:

This test is File I/O based. Connect the serial cable between the EVM and the COM1 port of the computer.

8.5.6.2 Procedure for Common Tone Generation Test

- Invoke the PC File I/O application by executing the following file:
...\\Embedded SDK\\src\\x86\\win32\\applications\\fileio\\fileio.exe
- Modify the settings to configure the File I/O application to run at 56000bps
- Using CodeWarrior, open:
...\\nos\\telephony\\ctg\\test\\test_ctg.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- The test results will be displayed on CodeWarrior's console

8.5.7 DTMF Detection Test

This test is intended to confirm the accuracy of DTMF Detection algorithms.

This test utilizes the CodeWarrior's File I/O. Input files *testx.in* (x = 1,2,3) for running this test are taken from the directory:

...\\nos\\telephony\\dtmf_det\\test\\c_sources

- *test1.in* tests the DTMF detection algorithm for Nominal keys detection
- *test2.in* tests the DTMF detection algorithm for Twist
- *test3.in* tests the DTMF detection algorithm for Dynamic range

The testing is performed for the following configuration:

- SampleRate = 8000 /* The algorithm performs detection for 8K sampling frequency only */
- OnDuration = 40ms minimum
- OffDuration = 40ms minimum

The output is stored in a buffer called *det_keys[]* and is compared with the standard output buffers *keybuf12[]* and *keybuf3[]* defined in the *testdtmfdet.c* file

- The standard output buffer for test1 and test2 is *keybuf12[]*
- The standard output buffer for test3 is *keybuf3[]*

8.5.7.1 Set-up for DTMF Detection Test

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**.

File I/O:

This test is File I/O based. Connect the serial cable between the EVM and the COM1 port of the computer.

8.5.7.2 Procedure for DTMF Detection Test

- Invoke the PC File I/O application by executing the following file:
...\\Embedded SDK\\src\\x86\\win32\\applications\\fileio\\fileio.exe
- Modify the settings to configure the File I/O application to run at 56000bps
- Using CodeWarrior, open:
...\\nos\\telephony\\dtmf_det\\test\\test_dtmf_det.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- The test results will be displayed on CodeWarrior's console window

8.5.8 DTMF Generation Test

The DTMF Generation test, which runs on the DSP56852 EVM, is intended to confirm the accuracy of DTMF Generation algorithms provided by the SDK.

This test utilizes the CodeWarrior's File I/O. The input samples are stored in the file:

...\\nos\\telephony\\dtmf_gen\\test_dtmfg\\test_in.io

The output samples are stored in file:

...\\nos\\telephony\\dtmf_gen\\test_dtmfg\\test_std.io

Testing is performed with the following configuration:

SampleRate	= 7200	The algorithm performs generation for either 8K or 7.2K sampling frequency
OnDuration	= 360	The number of samples corresponding to 50ms
OffDuration	= 360	The number of samples corresponding to 50ms
Amp	= 0.3	Amplitude should be less than 0.5

The output of the execution is compared to the contents of file *test_std.io* and frequency contents are verified in MATLAB.

8.5.8.1 Set-up for DTMF Generation Test

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**.

File I/O:

This test is File I/O based. Connect the serial cable between the EVM and the COM1 port of the computer.

8.5.8.2 Procedure for DTMF Generation Test

- Invoke the PC File I/O application by executing the following file:
...\\Embedded SDK\\src\\x86\\win32\\applications\\fileio\\fileio.exe

- Modify the settings to configure the File I/O application to run at 56000bps
- Using CodeWarrior, open:
`...\nos\telephony\dtmf_gen\test_dtmfg\test_dtmf_gen.mcp`
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select Run in the debugger to continue executing the test
- The test results will be displayed on CodeWarrior's console window

8.5.9 G.165 Test

The G.165 test is intended to confirm the accuracy of G.165 echo cancellation, hold release logic and tone disabler logic algorithms.

In the test for echo canceller, both the reference signal and the far-end signal are band-limited white noise. The echo of the reference signal is added to the far-end band-limited white noise signal. The echo-cancelled output signal is compared with the standard output.

In testing tone disabler and hold release logic, one of the signals should be a tone of $2100 \pm 21\text{Hz}$, with at least two phase reversals in one second.

Interlaced band-limited white noise data (reference and far-end speech) is read from an input file into a temporary buffer.

The data is then separated into two buffers: RinBuffer and SinBuffer.

The generated output is compared for bit exactness with the standard output, which is stored in the directory *test_g165* located at:

`...\dsp56852evm\nos\telephony\g165\.`

8.5.9.1 Set-up for G.165 Test

EVM Jumper Settings:

Use default settings as shown in the **DSP56852 Evaluation Module User's Manual**.

File I/O:

This test is File I/O based. Connect the serial cable between the EVM and the COM1 port of the computer.

8.5.9.2 Procedure for G.165 Test

- Invoke the PC File I/O application by executing the following file:
`...\Embedded SDK\src\x86\win32\applications\fileio\fileio.exe`
- Configure the File I/O application to run at 56000bps by modifying the settings
- Using CodeWarrior, open:
`...\nos\telephony\g165\test_g165\test_g165.mcp`
- Build and download the project; when the debugger reaches *main()*, it will stop.
- Select *Run* in the debugger to continue executing the test
- The test results will be displayed on CodeWarrior's console

8.5.10 G.168 Test

The G.168 test is intended to confirm the accuracy of G.168 echo cancellation, hold release logic and tone disabler logic algorithms.

In the test for echo canceller, both the reference signal and the far-end signal are band-limited composite source signals. The echo of the reference signal is added to the far-end band-limited composite source signal. The echo-cancelled output signal is compared with the standard output.

In testing tone disabler and hold release logic, one of the signals should be a tone of 2100 ± 21 Hz, with at least two phase reversals in one second.

Interlaced band-limited composite source signal data (reference and far-end speech) is read from an input file into a temporary buffer.

The data is then separated into two buffers: RinBuffer and SinBuffer.

The generated output is compared for bit exactness with the standard output, which is stored in the directory *reference* located at:

...\dsp56852evm\nos\telephony\g168\test_g168\.

8.5.10.1 Set-up for G.168 Test

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**.

File I/O:

This test is File I/O based. Connect the serial cable between the EVM and the COM1 port of the computer.

8.5.10.2 Procedure for G.168 Test

- Invoke the PC File I/O application by executing the following file:
...\Embedded SDK\src\x86\win32\applications\fileio\fileio.exe
- Modifying the settings to configure the File I/O application to run at 56000bps
- Using CodeWarrior, open:
...\nos\telephony\g168\test_g168\test_g168.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- The test results will be displayed on CodeWarrior's console

8.5.11 G.711 Test

The G.711 test is intended to confirm the accuracy of G.711 Log-PCM Encoding and Decoding provided by the SDK.

For testing the encoder, 8192 PCM input samples are generated with values from 0 to 65528, in increments of eight. For testing the decoder, the log PCM data is an array of numbers from 0 to 255, in increments of one.

The generated outputs are compared for bit exactness with the standard output files, then stored in the directory:

...dsp56852evm\nos\telephony\g711\test_g711\

8.5.11.1 Set-up for G.711 Test

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**.

File I/O:

This test is File I/O based. Connect the serial cable between the EVM and the COM1 port of the computer.

8.5.11.2 Procedure for G.711 Test

- Invoke the PC File I/O application by executing the following file:
...\\Embedded SDK\\src\\x86\\win32\\applications\\fileio\\fileio.exe
- Modify the settings to configure the File I/O application to run at 56000bps
- Using CodeWarrior, open:
...\\nos\\telephony\\g711\\test_g711\\test_g711.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- The test results will be displayed on CodeWarrior's console

8.5.12 G.726 Test

This test is intended to confirm the accuracy of G.726 algorithms provided by the SDK.

Input to the application is passed through a file, *files.in*, with four files for Encoder and 16 files for Decoder. All the input files for running this test are read from these directories:

...\\nos\\telephony\\g726\\test\\test_enc\\ (Encoder)

...\\nos\\telephony\\g726\\test\\test_dec\\ (Decoder)

The testing is performed with this configuration :

- SampleRate = 8000 /* The algorithm performs detection for 8k Sampling Freq. only */

The input is assumed to be A-Law or Mu-Law coded for Encoder and any of the following rates for Decoder: 40, 32, 24, or 16kbps.

The test vectors test the G.726 Encoder or Decoder for normal and overload conditions for both A-Law and Mu-Law and for the four 40, 32, 24, 16kbps data rates. The standard output test vectors are available in the folder ...\\nos\\telephony\\g726\\test\\test_dec\\ in *files.out*, where *files* refer to "rn16fa", "rn16fm", etc.

The output is stored in a buffer called *output[]* and compared with the standard output buffers provided in *files.out*.

From these standard vectors, 64 values at a time are taken for comparison. This step is repeated until the end of the vector.

8.5.12.1 Set-up for G.726 Test

EVM Jumper Settings:

Use default settings as shown in the **DSP56852 Evaluation Module User's Manual**.

File I/O:

This test is File I/O based. Connect the serial cable between the EVM and the COM1 port of the computer.

8.5.12.2 Procedure for G.726 Test

- Invoke the PC File I/O application by executing the following file:
`...\Embedded SDK\src\x86\win32\applications\fileio\fileio.exe`
- Modify the settings to configure the File I/O application to run at 56000bps
- Using CodeWarrior, open:
`...\nos\telephony\g726\test\test_enc\test_g726_enc.mcp` (Encoder) or
`...\nos\telephony\g726\test\test_dec\test_g726_dec.mcp` (Decoder)
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- The test results will be displayed on CodeWarrior's console

8.5.13 MFCR2 Detection Test

This test is intended to confirm the accuracy of MFCR2 detection algorithms.

This test utilizes the CodeWarrior's File I/O. Input files for the test are taken from the directory
`...\nos\telephony\MFCR2\test\inputs`

8.5.13.1 Set-up for MFCR2 Detection Test

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**.

File I/O:

This test is File I/O based. Connect the serial cable between the EVM and the COM1 port of the computer.

8.5.13.2 Procedure for MFCR2 Detection Test

- Invoke the PC File I/O application by executing the following file:
`...\Embedded SDK\src\x86\win32\applications\fileio\fileio.exe`
- Modify the settings to configure the File I/O application to run at 56000 bps

- Using CodeWarrior, open:
...\\nos\\telephony\\MFCR2\\test\\test_MFCR2.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- The test results will be displayed on CodeWarrior's console window. Note that the test cases test for invalid MFCR2 signals also. So, when the console displays that it is testing for error signals, the expected output is that there is no valid MFCR2 signal present.

8.5.14 Noise Suppression (NS) Test

The Noise Suppression test is intended to demonstrate the accuracy of the algorithm.

The input to NS is a speech signal corrupted with noise, which is processed on a frame basis (80 samples per frame). The input file, *ns.in*, is located in ...\\nos\\telephony\\ns\\test\\io. The file, *ns.out*, will be generated by the test and stored in the same directory. This should be identical to *ns.ref*. Using *Conv2au.exe* in the same directory, you can convert the *ns.in* and *ns.out* to audio files (*.au) and compare the audio of each file. You should notice a decrease in noise for the audio file for *ns.out*.

8.5.14.1 Set-up for NS Test

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**.

File I/O:

This test is File I/O based. Connect the serial cable between the EVM and the COM1 port of the computer.

8.5.14.2 Procedure for NS Test

- Invoke the PC File I/O application by executing the following file:
...\\Embedded SDK\\src\\x86\\win32\\applications\\fileio\\fileio.exe
- Modify the settings to configure the File I/O application to run at 56000bps
- Using CodeWarrior, open:
...\\nos\\telephony\\ns\\test\\test_ns.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop.
- Select *Run* in the debugger to continue executing the test
- When the Noise Suppression test is completed, the message "Output file generated; Play the files" will appear on the CodeWarrior's console
 - The file *ns.out* should have been generated and should be identical to *ns.ref*
- Using *Conv2au.exe*, convert *ns.in* and *ns.out* to audio files (*.au)
 - The audio file for *ns.out* should not be as noisy as the audio file for *ns.in*

8.5.15 VAD Test

The VAD test is intended to demonstrate the accuracy of the algorithm.

The input to the VAD is a speech signal, which is processed on a frame basis (64 samples per frame). The decision of 1 is made for the speech frame, and 0 for the silence frame. The results are compared with the reference output to confirm the accuracy of the algorithm.

Speech data is read from the input file *speech.in* into a temporary buffer. To locate *speech.in*, see:

...\dsp56852evm\nos\telephony\vad\test_vad\inputs\

The generated output is compared with the standard output for bit exactness, then stored in the output file *vad.ref*, found in:

...\dsp56852evm\nos\telephony\vad\test_vad\outputs\

8.5.15.1 Set-up for VAD Test

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**.

File I/O:

This test is File I/O based. Connect the serial cable between the EVM and the COM1 port of the computer.

8.5.15.2 Procedure for VAD Test

- Invoke the PC File I/O application by executing the following file:
...\Embedded SDK\src\x86\win32\applications\fileio\fileio.exe
- Modifying the settings to configure the File I/O application to run at 56000bps
- Using CodeWarrior, open:
...\nos\telephony\vad\test_vad\test_vad.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- The test results will be displayed on CodeWarrior's console

8.6 Modem Libraries

8.6.1 V.21 Loopback Test

The V.21 test demonstrates digital and analog loopback tests.

In the test for V.21, data bytes to be transmitted are input to the V.21 Transmitter. The samples generated (at a 7.2KHz rate) for transmission are given to the user for transmission. The received signal is sampled at a 7.2KHz sampling rate (by the user), and the samples are input to the V.21 Receiver. Information about whether the test passed or failed is output to and displayed on the CodeWarrior console.

8.6.1.1 Set-up for V.21 Loopback Test

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**.

To set up the test for V.21:

- Connect Line Out to Line In using an audio cable, as shown in the set-up in [Figure 8-1](#)
- Line Out serves as the V.21's transmitter
Line In serves as the V.21's receiver
- The connection from Line Out to Line In is not required for the Digital Loopback Test
- The connection from Line Out to Line In is required for the Analog Loopback Test; if this connection is not made, the test will fail.
 - If the connection is removed after the Analog Loopback Test starts, the message "Carrier Lost" will be displayed on the CodeWarrior console.

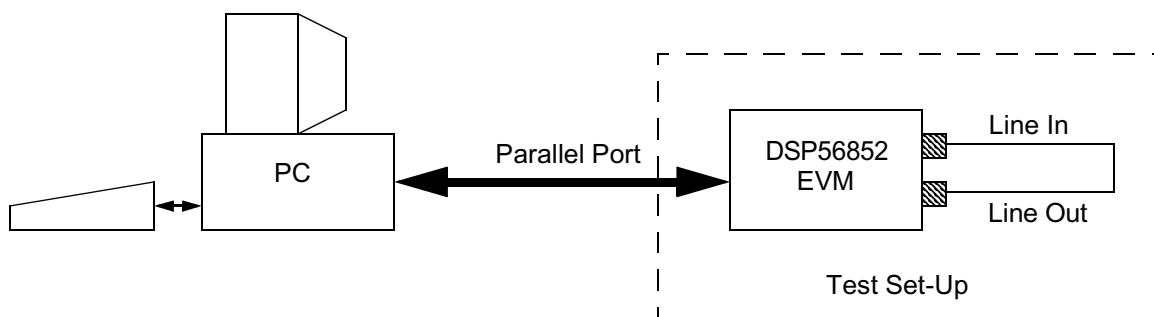


Figure 8-1. Test Set-up for V.21

8.6.1.2 Procedure for V.21 Loopback Test

- Using CodeWarrior, open:
...\\nos\\modem\\v21\\test\\test_v21.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- The test results will be displayed on CodeWarrior's console
- Once testing is complete, a "Pass" or "Fail" message appears on CodeWarrior's console
- With this process, both digital and analog loopback tests are verified

8.6.2 V.22bis Loopback Test

The V.22bis library implements only the data pump. The V.22bis library consists of a transmitter which modulates the incoming bit-stream and generates linear PCM samples to be transmitted on the telephone channel via the codec. The receiver demodulates the samples received through the codec and decodes the bits. The sampling rate for both transmission and reception is 7200Hz.

The V.22bis data pump works in two modes:

- **Calling modem:** The V.22bis soft modem initiates the call to the remote modem. The transmitter and the receiver are named as calling transmitter and calling receiver.

- **Answering modem:** The remote modem initiates the call to the V.22bis soft modem. Both the transmitter and the receiver are named as answering transmitter and answering receiver.

For the loopback test, if the transmitter is configured as calling, the receiver must be configured as answering and vice versa. In this loopback test, the transmitter is configured as Call modem and the receiver is configured as Answer modem. For detailed implementation of the loopback test, refer to the C code *testLoopback.c* in the directory **...modem\v22bis\test**

The Loopback test can be performed in two modes:

- In the **Digital Loopback** test, samples generated by the transmitter are input to the receiver without going through the codec, so the conversion of signals from digital to analog and analog to digital is avoided. This tests the functionality of the V.22bis transmitter and receiver.
- In the **Analog Loopback** test, the samples generated by the transmitter are sent to the codec at 7200 samples/second. The “codec in” and “codec out” pins on the EVM are shorted (closed) through a stereo cable, so it receives the samples from the same transmitter.

8.6.2.1 Set-up for V.22bis Loopback Test

EVM Jumper Settings:

For both digital and analog loopback tests, jumper JG4 on the DSP56852EVM should be closed (it is open by default). All other jumpers use the default settings shown in the **DSP56852 Evaluation Module User’s Manual**.

- For the digital loopback test, the “codec in” and “codec out” pins need not be shorted through cable. For the analog loopback test, the pins should be connected through stereo cable.
- Since there is only one test file which tests both digital and analog loopback, keep the “codec in” and “codec out” on the EVM connected through the stereo cable as shown in **Figure 8-2**.

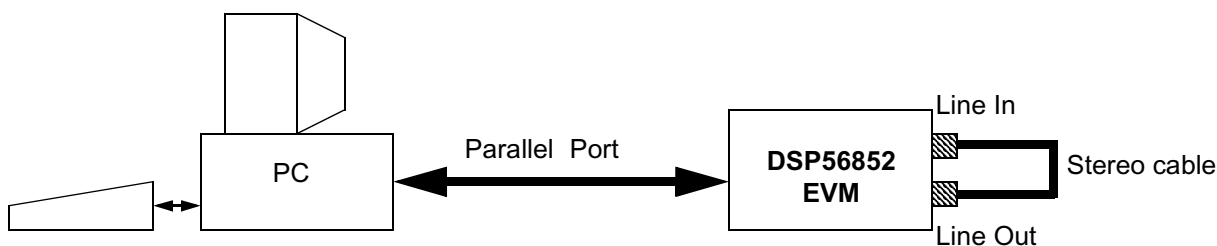


Figure 8-2. V.22bis Set-up for Digital and Analog Loopback Tests

8.6.2.2 Procedure for V.22bis Loopback Test

- Connect the stereo cable between “codec in” and “code out” pins on the EVM
- Using CodeWarrior, open:
...\\nos\\modem\\v22bis\\test\\loopback_test.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test

- The status of digital and analog loopback tests will be displayed on CodeWarrior's console; in this case, both the tests should display "PASS"
- Close the debugger window
- Disconnect the stereo cable between the "codec in" and "codec out" on the EVM
- Using CodeWarrior, open:
`...\nos\modem\v22bis\test\loopback_test.mcp`
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- In this case, the digital loopback test will "PASS" and the analog loopback will "FAIL", displaying the message "Connection Lost".

8.6.3 V.8bis Test

The V.8bis library implements all thirteen transactions specified by the V.8bis Standard, Revision 1998.

The V.8bis library can work both as an initiating station and as a responding station, depending on the flag supplied by the user. To learn more about flag setting, see the code in files:

`...\modem\v8bis\test_v8bis\c_sources\test_v8bisIS.c` (Initiating Station) or
`...\modem\v8bis\test_v8bis\c_sources\test_v8bisRS.c` (Responding Station)

The test input is a buffer containing host-config word; local/remote capabilities; priorities list and transmission gain; see **Appendix A** in the document **V.8bis Library** for details. Two tests with different capabilities have been provided, and each test has all 13 transactions, 16 test cases per test. The expected output is stored in another buffer. Once the transaction is complete, the output obtained is compared with the expected output, and the test outcome is displayed on the CodeWarrior console.

8.6.3.1 Set-up for V.8bis Test

EVM Jumper Settings:

For this test, jumper JG4 on the DSP56852EVM should be closed (it is open by default). All other jumpers use the default settings shown in the **DSP56852 Evaluation Module User's Manual**.

Set up the test circuit and connect Line In to Line Out and vice-versa as shown in [Figure 8-3](#).

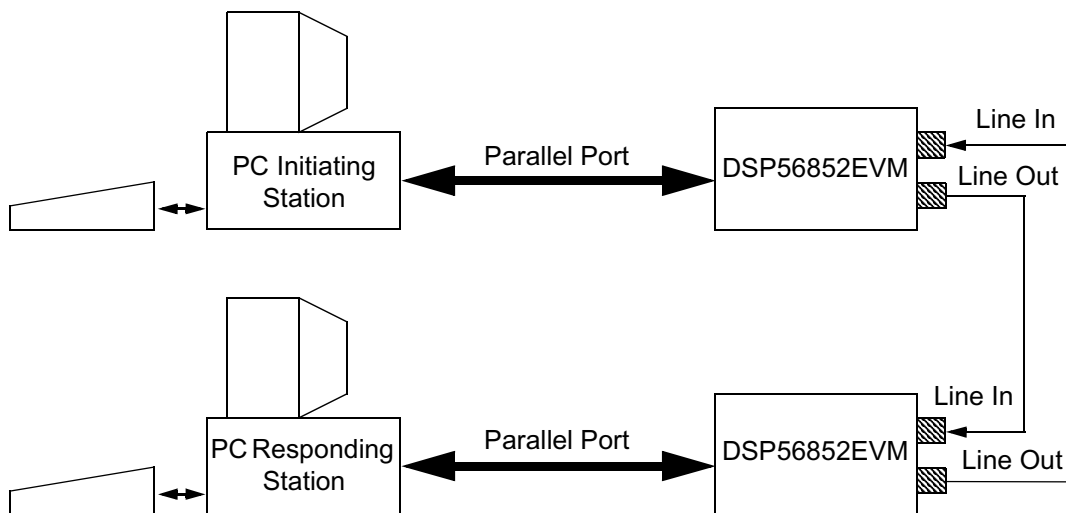


Figure 8-3. V.8bis Test Set-up

8.6.3.2 Procedure for V.8bis Test

- Using CodeWarrior on one PC, open:
...\\nos\\modem\\v8bis\\test_v8bis\\test_v8bis_IS.mcp
- Using CodeWarrior on the other PC, open:
...\\nos\\modem\\v8bis\\test_v8bis\\test_v8bis_RS.mcp
- Build and download each project; when the debugger reaches *main()*, it will stop
- The responding station must begin running before the initiating station.
- Select Run in the debugger of the responding station
- Select Run in the debugger of the initiating station.
- Test results will be displayed on CodeWarrior's console
- Note: The transactions can be made audible by connecting PC audio speakers in the test circuit, as shown in [Figure 8-4](#). The speakers' volume should be relatively high, approximately half of the entire volume scale, to meet the threshold signal level of the codec on the 56852 EVM board. Use the ACS90 Computer Speaker System or its equivalent for best results.

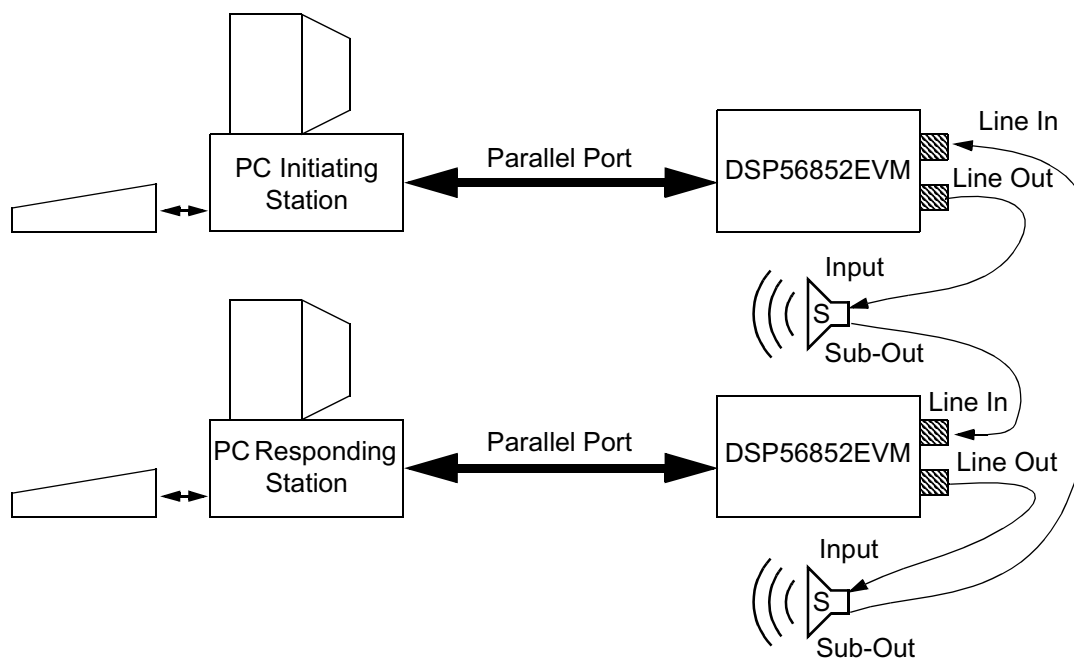


Figure 8-4. V.8bis Test Set-up with Speakers

8.7 Speech Libraries

8.7.1 VRLite-1 Test

The VRLite-1 test demonstrates the accuracy of the Voice Recognition algorithms.

In the test for VRLite-1, the input is speech samples, sampled at 8KH, then stored in files. Information about whether the test passed or failed is output to and displayed on the CodeWarrior console.

- For **Training with Rejection Analysis**, the two utterances from a speaker are stored in files *utt1.in* and *utt2.in*:
`...\nos\speech\vr1ite1\test\io\`
 - The *utt1.in* and *utt2.in* files contain samples corresponding to approximately 5 seconds. The previously-trained models are stored in the file *prev_models.in* at:
`...\nos\speech\vr1ite1\test\io\`
 - The global noise mean values are directly entered in the GlobalStats[] buffer of the *pConfig* structure, in the test file *test_vr1ite1_training.c*:
`...\nos\speech\vr1ite1\test\c_sources`
- For **Recognition**, one utterance from a speaker is stored in the file *utt_recog.in*:
`...\nos\speech\vr1ite1\test\io\`
 - The *utt_recog.in* file contains samples corresponding to approximately 5 seconds. The previously-trained models are used from the file *prev_models.in*:
`...\nos\speech\vr1ite1\test\io\`
 - The global noise mean values are directly entered in the GlobalStats[] buffer of the *pConfig*

structure, where they are stored in the test file *test_vrlite1_training.c*:
 ...\\nos\\speech\\vrlite1\\test\\c_sources

8.7.1.1 Set-up for VRLite-1 Test

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**.

File I/O:

This test is File I/O based. Connect the serial cable between the EVM and the COM1 port of the computer.

8.7.1.2 Procedure for VRLite-1 Test

- Invoke the PC File I/O application by executing the following file:
 ...\\Embedded SDK\\src\\x86\\win32\\applications\\fileio\\fileio.exe
- Modify the settings to configure the File I/O application to run at 56000bps
- Using CodeWarrior, open:
 ...\\nos\\speech\\vrlite1\\test\\test_vrlite1.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- The test results will be displayed on CodeWarrior's console
- After the Rejection Analysis is complete, a message indicating whether the current model is accepted or not is flagged out on CodeWarrior's console
- After the utterance is recognized in file *utt_recog.in*, a pass or fail message is flagged out on CodeWarrior's console

8.8 Feature Phone Libraries

8.8.1 Generic Echo Cancellor Library Application

Two example applications are discussed in this section; a speakerphone application and a test application.

8.8.1.1 Speakerphone Application

A full duplex Speakerphone application has been provided for use on an DSP5685xEVM board using the *gec.lib* library. Modifications can be made to suit the application. The application uses codec drivers and serial port drivers that are part of the SDK and is located in the */nos/applications/fdspk* directory.

8.8.1.2 Test Application

A test application has also been provided to verify the functionality of the module. To run it, the user can open the *testgec.mcp* project in the *nos/telephony/gec/test* directory. Example code for this application is given in [Code Example 8-1](#).

The filter *gecLengthIndex* is set to 0 and the *correctCoeffs* is equated to *gecCoeffs0.h*. The user can change both the *gecLengthIndex* and *correctCoeffs* to the same number to test for different filter lengths; e.g., *gecLengthIndex* = 4 and *correctCoeffs* = *#include gecCoeffs4.h*. The application is meant to run the module for 800 samples worth of white noise and return a “PASS” or “FAIL” result, depending on the run. The result is printed on the console screen. This application runs digitally, implying it requires no other modules.

Code Example 8-1. GEC testapp.c

```

/* File: main.c */
#include <stdio.h>
#include "port.h"
#include "teldefs.h"
#include "gec.h"

int lineSamples[800]={
#include "gecLine.h"
};
int audioSamples[800]={
#include "gecCoeffs0.h"
};

int compare;
int done;
int cntr;
int index;

/* module related declarations */
teldefs_sControl      line1Control;
teldefs_sSamples      line1Samples;
gec_sData*            pgec1Data;

int main(void)
{
    int i;
    int fail = 0;

    printf("Generic Echo Canceller Test...\n");
    cntr = 0;
    compare = 0;
    done = 0;
    index = 0;

    line1Control.gecLengthIndex = 0;          // choosing filter length
    pgec1Data = gecEchoCancellerCreate(&line1Control);

    line1Control.hookSwitch = 1;
    line1Control.handsFreeLayer1 = 0;

    // All for test mode. Do not use in regular mode.
    line1Control.disableAnalysis = 1;
    line1Control.pre2lecPowMsw = 1000;
    line1Control.dtHang = -1;
    line1Control.nesPresent = 1;
    line1Control.trainLec = 1;

```

```

while(!done) {
    for(i=0; i<5; i++){
        line1Samples.line[i] = lineSamples[index];
        line1Samples.audio[i] = audioSamples[index];
        index++;
    }

    for(i=0; i<5; i++) {
        line1Control.lecDelaySample = line1Samples.line[i];
        if (line1Control.hookSwitch == 1) {
            gecEchoCanceller(pgec1Data,&line1Control,&line1Samples);
            if(line1Control.trainLec == 0)
                asm(debughlt);
        }
    }

    cntr++;

    if(cntr >= 160){
        compare = 1;
        cntr = 0;
    }

    if(compare == 1){
        for(i=0, i<64, i++){
            if(correctCoeffs[i] != pgec1Data -> lecSpkcoefs[i]){
                fail = 1;
            }
        }
        done = 1;
    }
}
gecEchoCancellerDestroy(pgec1Data, &line1Control);
if (fail == 0) printf("PASS \n");
else printf("FAIL \n");

return 0;
}

```

Should the application print “FAIL”, this would indicate that the Generic Echo Canceller Library is not operating correctly. This is not expected to happen for the test application; however, if it does, please report the failure to Motorola for resolution.

8.8.2 Type 1 and 2 Telephony Parser Library

Two example applications are provided in this section. The first example is a self-verification test. It verifies the functionality of the Type 1 and 2 Telephony Parser Library. The second example is provided to illustrate how to use the Type 1 and 2 Telephony Parser Library and how to integrate it with the Type 1 Telephony Features Library. This example is for illustrative purposes only, and source code is not included in the SDK.

8.8.2.1 Type 1 and 2 Telephony Parser Verification Test

To verify the functionality of the Type 1 and 2 Telephony Parser Library, an individual test application is provided. The test performs parsing of the samples that are precaptured from a demodulated Caller ID in Multiple Data Message Format (MDMF). The test application is located in the ...\\telephony\\cidparse\\test directory. The name of the test project is *test.mcp*.

8.8.2.1.1 Test Set-up and Procedure

The test application runs in simulation mode and does not require external equipment. Before executing the test application, the *Target Setting Protocol* must be selected as *Simulator*. It can be set in the *Protocol* option of the *M56800E Target Settings*.

The test project for the Type 1 and 2 Telephony Parser Library is *test.mcp*. Before loading the executable project, the project must be compiled without errors. To compile, choose the *Make* option in the project menu or use function key [F7]. To load the executable project, choose the *Debug* option in the project menu or use function key [F5].

To run the test application, use the mouse to choose the green arrow button in the compile window or manually by using the function key [F5]. After the application starts running, the application performs initialization and automatically parses the pre-captured samples. The test will compare the parsed message to the expected result. If the Type 1 and 2 Telephony Parser Library functionality is verified, the message "No Parser Error" will be printed on the PC console window.

Be aware that the precaptured samples to be processed in the Type 1 and 2 Telephony Parser Library are stored in header file *mdmf.h* and the resultant parsed message used for comparison is stored in header file *message.h*. It is recommended that the user does not change the content in these header files.

No malfunction is expected to occur with this test application. However, if any response other than the above message occurs, please report the failure and test conditions to Motorola for resolution.

8.8.2.2 Type 1 and 2 Telephony Parser Library Example Application

An example using the Type 1 and 2 Telephony Parser Library to illustrate how to set the library variables for *Type1CID* in the Control structure of *teldefs.h* is shown in [Code Example 8-2](#). This code does not exist in the SDK.

Code Example 8-2. Use of *CIDMessageParser* Interface

```
#include <stdio.h>
#include <stdlib.h>
#include "teldefs.h"
#include "cid1.h"
#include "mem.h"
#ifdef USEPARSER
#include "cidparser.h"
#endif
```




```

struct teldefs_tsSamples Line1Samples;
struct cid_tsData* pcid1Data;
struct teldefs_tsControl Line1Control;
#ifdef USEPARSER
teldefs_sParser ParserControl;
#endif

int main(void)
{
    // Initialize some necessary structure elements and call create function

    Line1Control.messageDone=0;
    Line1Control.cidByteReady = 0;
    Line1Control.ExtUseCheck=0;
    Line1Control.NoExtFound=1;
    Line1Control.FrameErrors=0;
    Line1Control.dtmfRequest=0;
    Line1Control.dtmfComplete=0;
    Line1Control.hookSwitch = 0;
    Line1Control.flashCommand = 0;
    Line1Control.cwdCommand = 0;
    pcid1Data = Type1CIDcreate(&Line1Control);

#ifdef USEPARSER
    ParserControl.FskMessageIndex=0;
    ParserControl.FskParserLength=0;
#endif

    while(1) {
        /* Wait for 5 8KHz samples to be received and transmitted by */
        /* the Codec Interrupt Service Routine. */

        while(!SamplesReady){
            ; /* do nothing */
        }
        CalleridAppMain();
    }

    Type1CIDDestroy(pcid1Data,&Line1Control);

    return 0;
}

void CalleridAppMain(){
    int i;

    /* copy samples from Codec buffers */
    for( i = 0; i < 5 ; i++){
        codecBufferLeftout[i] = Line1Samples.audio[i];
    }
}

```

```

        codecBufferRightout[i] = Line1Samples.line[i];
        Line1Samples.line[i] = codecBufferLeftin[i];
        Line1Samples.audio[i] = codecBufferRightin[i];
    }

    // Get Ring Signal from DAA

    if( /* Poll Ring Detect Bit Here */ ){
        Line1Control.cidRingPolarity = 1;        // Ring present
    }
    else{
        Line1Control.cidRingPolarity = 0;        // Ring not present
    }

    // Call Type 1 Telephony Features Library
    Type1CID(pcid1Data,&Line1Control,&Line1Samples);

    // Process command for line flash
    if(Line1Control.flashCommand){
        Line1Control.flashCommand = 0;
        if(Line1Control.flashPolarity == 1) go_offhook();
        else go_onhook();
    }

#ifdef USEPARSER

    // Use Type 1 and 2 Telephony Parser Library
    CIDMessageParser(&ParserControl, &Line1Control);

    if(ParserControl.FskParserLength != 0){
        /* Parsed Message Ready. Send to Output Device */
        if(ParserControl.ErrorType == 0){
            for( i = 0 ; i < ParserControl.FskParserLength ; i++)
                printf("%c",ParserControl.FskParserBuffer[i]);
        }
        ParserControl.FskParserLength=0;
    }
#elseif

    // Use Custom Parser

    if(Line1Control.cidByteReady){
        /* Buffer cid bytes here*/
        cid_message_buffer[cid_message_index++] = Line1Control.cidByte;
    }
    if (Line1Control.messageDone){
        if(Line1Control.FrameErrors == 0){
            /* Call custom Parser here */
        }
        else Line1Control.FrameErrors = 0;
    }
}

```

```
#endif
```

```
}
```

Code Example 8-2 shows a basic application using the *test.mcp* module. This module is designed to interact with the Type 1 and 2 Telephony Parser Library (*cidparser.lib*). However, independent use of the *test.mcp* module is supported. When implementing a full duplex speakerphone, the *test.mcp* module is intended to be used with the *fdspk.lib* and *gec.lib* libraries; however, custom line echo canceller and full duplex speakerphone software can also interface with the *test.mcp* module.

This example shows how to manage Caller ID information using either the Type 1 and 2 Telephony Parser Library or a custom parser. The definition *USEPARSER* is undefined when a custom parser is desired. When using a custom parser, the application must transfer the input bytes into a buffer, then process them. This is done automatically when using the Type 1 and 2 Telephony Parser Library.

While this module is primarily intended to be used for Caller ID reception, other convenient features are added to allow for simple integration into a Caller ID telephone. While the module is in an on-hook state (*Line1Control.hookSwitch* = 0), the *line[]* samples buffer will be filled with ring generator samples during power ringing. Otherwise, the *line[]* samples will be filled with zeros. These samples may act as an output to an audio channel (for example, to be played to a speaker).

While the module is in an off-hook state (*Line1Control.hookSwitch* = 1), the *audio[]* samples buffer will be filled with DTMF samples when a DTMF digit is requested or will otherwise remain unchanged. The *line[]* samples remain unchanged during the off-hook state unless *Line1Control.handsFreeLayer1* is set (a condition which is not shown in this example). In this case, the *gec[]* samples will be copied to the *line[]* samples structure.

The *gec[]* samples can be obtained by using the *gec.lib* library, or by using a separate custom echo canceller. The *audio[]* samples will be modified by the *fdspk.lib* library (or custom full duplex speakerphone software) in this case. See the description of these libraries and the Feature Phone Application for more details.

In this example, there are two GPIO pins: one output pin, which places the telephone interface on-hook or off-hook, and one input pin, which indicates the ring detect from the telephone interface. The ring detect signal is polled in the application and its polarity is placed in the variable *Line1Control.cidRingPolarity*. This information is necessary for the Type 1 and 2 Telephony Parser Library, even when *disableRinger* is set (disabling the ring tone generator) and should be polled at a rate of 1600/second as shown in the application. However, slower polling rates are allowable down to the byte rate of Caller ID, which is 120/second.

It is assumed that the functions *go_onhook()* and *go_offhook()* which are not shown in **Code Example 8-2**, will simply set or clear the output pin connected to the telephone interface. However, note that these are not the same functions set or cleared when the application actually puts the module into an on-hook or off-hook state. In this situation, the application must also set or clear the *Line1Control.hookSwitch* variable, then call *Type1CIDinit()*.

The codec Interrupt Service Routine is also not shown in this example. This routine is assumed to transmit and receive 8KHz samples from a codec and to read/write to the buffers *codecBufferLeftin[]* and *codecBufferLeftout[]*. This codec is connected to the telephone line interface. This routine is also assumed

to transmit and receive 8KHz samples from a codec and read/write to the buffers *codecBufferRightin[]* and *codecBufferRightout[]*. This codec is connected to an audio interface. When the ISR has received/transmitted five samples, the buffers are full/empty and the *SamplesReady* flag is set, indicating to the application that it should call *CalleridAppMain()*. The calling rate of this routine should be 1600 calls/second.

In the beginning of the *CalleridAppMain()* routine, the codec samples are copied into the module's sample structure. Note that for the output samples, the *line[]* samples are copied into the right (audio) codec channel, and the *audio[]* samples are copied into the left (line) codec channel. This sample "criss-cross" is indicative of the natural signal flow for a telephone.

8.8.3 Full Duplex Speakerphone Library

Three example applications are discussed in this section: a speakerphone application, a test application, and a diagnostics application. Example code for these are found in the **Full Duplex Speakerphone Library** manual, Chapter 3.

8.8.3.1 Full Duplex Speakerphone Application

The Full Duplex Speakerphone application has been provided for use on a DSP5685xEVM board using the *fdspk.lib* library. Modifications can be made to suit the application. The application uses codec drivers and serial port drivers that are part of the SDK. It is located in the */nos/applications/fdspk* directory. Example code for this application is given in [Code Example 8-3](#).

8.8.3.2 Test Application

A test application has been provided to verify the functionality of the module. The user can open the *testfdspk.mcp* project in the *nos/telephony/fdspk/test* directory to run it. Example code for this application is found in [Code Example 8-3](#).

The test header file for coefficients must be changed for different length indices. The *aecFilterIndex* is set to 2 and the *correctCoeffs* is equated to *fdspkCoeffs2.h*. The user can change both the *aecFilterIndex* and *correctCoeffs* to the same number in order to test for different filter lengths; e.g., *aecFilterIndex = 4* and *correctCoeffs = #include fdspkCoeffs4.h*. The application is meant to run the module for 800 samples worth of white noise and return a "PASS" or "FAIL" result, depending on the run. The result is printed on the console screen. This application runs digitally, implying that it does not require any codec or serial port drivers to run.

Code Example 8-3. Speakerphone testapp.c

```
#include <stdio.h>
#include "port.h"
#include "teldefs.h"
#include "fdspk.h"
```



```

int lineSamples[800]={
#include "fdspkLine.h"
};
int audioSamples[800]={
#include "fdspkAudio.h"
};
int correctCoeffs[]={
#include "fdspkCoeffs2.h"
};

int compare;
int done;
int cntr;
int index;

teldefs_sControl    line1Control;
teldefs_sSamples    line1Samples;
fdspk_sData*        pfdspk1Data;

int main(void)
{
    int i;
    int fail = 0;

    printf("Full Duplex Speakerphone Test ...\n");
    cntr = 0;
    compare = 0;
    done = 0;
    index = 0;

    line1Control.aecLengthIndex = 2;
    pfdspk1Data = fdspkCreate(&line1Control);

    Line1Control.digitalSwitch1 = 1;
    Line1Control.digitalSwitch2 = 0;
    Line1Control.digitalSwitch3 = 0;
    Line1Control.digitalSwitch4 = 0;
    Line1Control.digitalSwitch5 = 1;
    Line1Control.digitalSwitch6 = 0;

    line1Control.disableBothEc = 0;

    Line1Control.hookSwitch = 1;
    Line1Control.handsFreeLayer1 = 1;
    Line1Control.volumeGaindB = 18;
    Line1Control.silenceOffset = 363;
    Line1Control.erlFactor = 5574;

    // All for test mode. Do not use in regular mode.
    Line1Control.disableAnalysis = 1;
    Line1Control.prelecPowMsw = 1000;
    Line1Control.aecdHang = -1;
    Line1Control.nesPresent = 0;
    Line1Control.trainAec = 1;
    pfdspk1Data->doubleTalkFlag = 0;

```

```

while(!done){
    for( i = 0; i < 5 ; i++){
        Line1Samples.line[i] = lineSamples[index];
        Line1Samples.audio[i] = audioSamples[index];
        index++;
    }

    for(i=0;i<5;i++){
        Line1Control.lecDelaySample = Line1Samples.line[i];
        if (Line1Control.hookSwitch == 1){
            if (Line1Control.handsFreeLayer1 == 1)
                fdspkState(&fdspk1Data,&line1Control,&line1Samples);
        }
        line1Control.sampleIndex += 1;
        if (line1Control.sampleIndex == 5)
            line1Control.sampleIndex = 0;
    }

    cntr++;
    if(cntr >= 160){
        compare = 1;
        cntr = 0;
    }

    if(compare == 1){
        for( i = 0 ; i < 192 ; i++){
            if(correctCoeffs[i] != pfdspk1Data->aecSpkCoefs[i]){
                fail = 1;
            }
        }
        compare = 0;
        done = 1;
    }

}
if (fail == 0) printf("PASS \n");
else printf("FAIL \n");

return 0;
}

```

8.8.3.3 Diagnostics Application

The diagnostics application discussed in detail in Chapter 3 of the **Full Duplex Speakerphone Library** manual uses the diagnostic procedures of the module and can be used on a DSP5685xEVM board. It is located in the *nos/telephony/fdspk/diagnostics* folder and is used to obtain key values required to tune the speakerphone. Example code for this application is given in Chapter 3 of the the **Full Duplex Speakerphone Library** manual.

8.8.4 Type 1 Telephony Features Library

8.8.4.1 Type 1 Verification Test

To verify the functionality of the Type 1 Telephony Features Library, an individual test application is provided. The test processes the samples that are precaptured from a nominal FSK transmission. The corresponding Caller ID will be printed on screen. The test application is located in the `...telephony\cidtype1\Test` directory. The name of the test project is `cid1test.mcp`.

8.8.4.2 Test Set-up and Procedure

The test application runs in simulation and does not require external equipment. Before executing the test application, the *Target Setting Protocol* must be selected in the *Simulator*. It can be set in *Protocol* option in *M56800 Target Settings*.

The test project for Type 1 is `cid1test.mcp`. Before loading the executable project, the project must be compiled with no errors. This is done by choosing the *Make* option in the *Project* menu or by pressing [F7]. To load the executable project, choose the *Debug* option in the *Project* menu or press [F5].

To run the test application, use the mouse to choose the green arrow button or manually press [F5] on the keyboard. After the application starts running, the application initializes and automatically processes the precaptured samples.

The functionality of the Type1 Telephony Features Library will be verified and a Caller ID message will be printed on the console window. The application stops after completing the process and prints "PASS" on the console window.

Should the application print "FAIL", the Type 1 Telephony Features Library is not operating correctly. This is not expected to happen for this test application; however, if it does, please report the failure and test conditions to Motorola for resolution.

8.8.4.2.1 Example Application Using `cid1.lib`

An example using the Type 1 Telephony Features Library to illustrate how to set the library variables for `Type1CID` in the Control structure of `teldefs.h` is shown in [Code Example 8-4](#). This code does not exist in the actual `cidtype1` directory in the SDK.

Code Example 8-4. Use of `Type1CID` Interface

```
#include <stdio.h>
#include <stdlib.h>
#include "teldefs.h"
#include "cid1.h"
#include "mem.h"
#ifdef USEPARSER
#include "cidparser.h"
#endif
```



```

struct teldefs_tsSamples Line1Samples;
struct cid_tsData* pcid1Data;
struct teldefs_tsControl Line1Control;
#ifdef USEPARSER
teldefs_sParser ParserControl;
#endif

int main(void)
{

    // Initialize some necessary structure elements and call create function

    Line1Control.messageDone=0;
    Line1Control.cidByteReady = 0;
    Line1Control.ExtUseCheck=0;
    Line1Control.NoExtFound=1;
    Line1Control.FrameErrors=0;
    Line1Control.dtmfRequest=0;
    Line1Control.dtmfComplete=0;
    Line1Control.hookSwitch = 0;
    Line1Control.flashCommand = 0;
    Line1Control.cwdCommand = 0;
    pcid1Data = Type1CIDcreate(&Line1Control);

#ifdef USEPARSER
    ParserControl.FskMessageIndex=0;
    ParserControl.FskParserLength=0;
#endif

    while(1) {

        /* Wait for 5 8KHz samples to be received and transmitted by */
        /* the Codec Interrupt Service Routine. */

        while(!SamplesReady){
            ; /* do nothing */
        }
        CalleridAppMain();

    }

    Type1CIDDestroy(pcid1Data,&Line1Control);

return 0;
}

void CalleridAppMain(){
int i;

    /* copy samples from Codec buffers */
    for( i = 0; i < 5 ; i++){
        codecBufferLeftout[i] = Line1Samples.audio[i];
        codecBufferRightout[i] = Line1Samples.line[i];
    }
}

```




```

Line1Samples.line[i] = codecBufferLeftin[i];
Line1Samples.audio[i] = codecBufferRightin[i];
}

// Get Ring Signal from DAA

if( /* Poll Ring Detect Bit Here */ ){
    Line1Control.cidRingPolarity = 1;        // Ring present
}
else{
    Line1Control.cidRingPolarity = 0;        // Ring not present
}

// Call Type 1 Telephony Features Library

Type1CID(pcid1Data,&Line1Control,&Line1Samples);

// Process command for line flash

if(Line1Control.flashCommand){
    Line1Control.flashCommand = 0;
    if(Line1Control.flashPolarity == 1) go_offhook();
    else go_onhook();
}

#ifdef USEPARSER

// Use Type 1 and 2 Telephony Parser Library

CIDMessageParser(&ParserControl, &Line1Control);

if(ParserControl.FskParserLength != 0){
    /* Parsed Message Ready. Send to Output Device */
    if(ParserControl.ErrorType == 0){
        for( i = 0 ; i < ParserControl.FskParserLength ; i++)
            printf("%c",ParserControl.FskParserBuffer[i]);
    }
    ParserControl.FskParserLength=0;
}

#elseif

// Use Custom Parser

if(Line1Control.cidByteReady){
    /* Buffer cid bytes here*/
    cid_message_buffer[cid_message_index++] = Line1Control.cidByte;
}
if (Line1Control.messageDone){
    if(Line1Control.FrameErrors == 0){
        /* Call custom Parser here */
    }
    else Line1Control.FrameErrors = 0;
}

}

```

```
#endif
}
```

Code Example 8-4 shows a basic application using the *cid1.lib* module. This module is designed to interact with the Type 1 and 2 Telephony Parser Library (*cidparser.lib*); however, independent usage of the *cid1.lib* module is supported. When implementing a full duplex speakerphone, the *cid1.lib* module is intended to be used with the *fdspk.lib* and *gec.lib* modules; however, custom line echo canceller and full duplex speakerphone software can also interface with the *cid1.lib* module.

This example shows how to manage Caller ID information using either the Type 1 and 2 Telephony Parser Library or a custom parser. The definition USEPARSER is undefined when a custom parser is desired. When using a custom parser, the application must transfer the input bytes into a buffer, then process them. When using the Type 1 and 2 Telephony Parser Library, this is done automatically.

While this module is primarily intended to be used for Caller ID reception, other convenient features are added to allow for simple integration into a Caller ID telephone. While the module is in an on-hook state (*Line1Control.hookSwitch* = 0), the *line[]* samples buffer will be filled with ring generator samples during power ringing. Otherwise, the *line[]* samples will be filled with zeros. These samples may act as an output to an audio channel (for example, to be played to a speaker).

While the module is in an off-hook state (*Line1Control.hookSwitch* = 1), the *audio[]* samples buffer will be filled with DTMF samples when a DTMF digit is requested and will remain unchanged otherwise. The *line[]* samples remain unchanged during the off-hook state, unless *Line1Control.handsFreeLayer1* is set (a condition which is not shown in this example). In this case, the *gec[]* samples will be copied to the *line[]* samples structure.

The *gec[]* samples can be obtained by using the *gec.lib* module, or by using a separate custom echo canceller. The *audio[]* samples will be modified by the *fdspk.lib* module (or custom full duplex speakerphone software) in this case. See the description of these modules and FeaturePhone application for more detailed information.

In this example, there are two GPIO pins; one output pin, which places the telephone interface on-hook or off-hook, and one input pin, which indicates the ring detect from the the telephone interface. The ring detect pin is polled in the application and its polarity is placed in the variable *Line1Control.cidRingPolarity*. This information is necessary for the Type 1 Telephony Features Library, even when *disableRinger* is set (disabling the ring tone generator) and should be polled at a rate of 1600/second as shown in the application. However, slower polling rates may be tolerated.

It is assumed that the functions *go_onhook()* and *go_offhook()*, which are not shown in **Code Example 8-4**, would simply set or clear the output pin connected to the telephone interface. However, note that these are not the same as when the application actually puts the module into an on-hook or off-hook state. In this situation, the application must also set or clear the *Line1Control.hookSwitch* variable and then call *Type1CIDinit()*.

The Codec Interrupt Service Routine is also not shown in this example. This routine is assumed to transmit and receive 8KHz samples from a codec and read/write to the buffers *codecBufferLeftin[]* and *codecBufferLeftout[]*. This codec is connected to the telephone line interface. This routine is also assumed to transmit and receive 8KHz samples from a codec and read/write to the buffers *codecBufferRightin[]* and

codecBufferRightout[]. This codec is connected to an audio interface. When the ISR has received/transmitted five samples, the buffers are full/empty and the *SamplesReady* flag is set, indicating to the application that it should call *CalleridAppMain()*. The calling rate of this routine should be 1600 calls/second.

In the beginning of the *CalleridAppMain()* routine, the codec samples are copied into the module's sample structure. Note that for the output samples, the *line[]* samples are copied into the right (audio) codec channel, and the *audio[]* samples are copied into the left (line) codec channel. This sample "criss-cross" is indicative of the natural signal flow for a telephone.

8.8.5 TYPE 1 and 2 Telephony Features Library

8.8.5.1 Type 2 Verification Test

To verify the functionality of the Type 1 and 2 Telephony Features Library, an individual test application is provided. The test processes the samples that are precaptured from a nominal FSK transmission. The corresponding Caller ID will be printed on screen. The test application is located in the *...telephony\cidtype12\Test* directory. The name of the test project is *cid12test.mcp*.

8.8.5.1.1 Test Set-up and Procedure

The test application runs in the simulator mode and does not require external equipment. Before executing the test application, select *Simulator* as the *Target Setting Protocol*. It can be set in the *Protocol* option of the *M56800 Target Settings*.

The test project for Type 2 is *cid12test.mcp*. Before loading the executable project, the project must be compiled with no errors. This is done by choosing the *Make* option in the *Project* menu, or by pressing [F7]. To load the executable project, choose the *Debug* option in the *Project* menu, or press [F5].

To run the test application, use the mouse to click the green arrow button or manually press [F5] on the keyboard. After the application starts running, the application initializes and automatically processes the precaptured samples.

The functionality of the Type 1 and 2 Telephony Features Library will be verified, and a Caller ID message will be printed on the console window. The application stops after completing the process and prints "PASS" on the console window.

Should the application print "FAIL", the Generic Echo Canceller Library is not operating correctly. This is not expected to happen for this test application; however, if it does, please report the failure and test conditions to Motorola for resolution.

8.8.5.2 Example Application Using *cid12.lib*

An example using the Type 1 and 2 Telephony Features Library to illustrate how to set the library variables for *Type12CID* in the Control structure of *teldefs.h* is shown in [Code Example 8-5](#). This code does not exist in the actual *cidtype12* in the SDK.

Code Example 8-5. Use of Type12CID Interface

```

#include <stdio.h>
#include <stdlib.h>
#include "teldefs.h"
#include "cid12.h"
#include "mem.h"
#ifdef USEPARSER
#include "cidparser.h"
#endif

#ifdef USEGEC
#include "gec.h"
#endif

struct teldefs_tsSamples Line1Samples;
struct cid_tsData* pcid1Data;
struct teldefs_tsControl Line1Control;
#ifdef USEPARSER
teldefs_sParser ParserControl;
#endif

#ifdef USEGEC
struct gec_tsData* pgec1Data;
#endif

int main(void)
{
    // Initialize some necessary structure elements and call create function
    Line1Control.messageDone=0;
    Line1Control.cidByteReady = 0;
    Line1Control.ExtUseCheck=0;
    Line1Control.NoExtFound=1;
    Line1Control.FrameErrors=0;
    Line1Control.dtmfRequest=0;
    Line1Control.dtmfComplete=0;
    Line1Control.hookSwitch = 0;
    Line1Control.flashCommand = 0;
    Line1Control.cwdCommand = 0;
    pcid1Data = Type12CIDcreate(&Line1Control);

#ifdef USEPARSER
    ParserControl.FskMessageIndex=0;
    ParserControl.FskParserLength=0;
#endif

#ifdef USEGEC
    //initialize all process variables for GEC
    Line1Control.gecLengthIndex = 0;
    pgec1Data = gecEchoCancellerCreate(&Line1Control);
#endif

    while(1) {
        /* Wait for 5 8KHz samples to be received and transmitted by */
        /* the Codec Interrupt Service Routine. */

```

```

while(!SamplesReady){
    ; /* do nothing */
}
CalleridAppMain();

}

Type12CIDdestroy(pcid1Data,&Line1Control);
#ifdef USEGEC
    gecEchoCancellerDestroy(pgec1Data, &Line1Control);
#endif

return 0;
}

void CalleridAppMain(){
    int i;

    /* copy samples from Codec buffers */
    for( i = 0; i < 5 ; i++){
        codecBufferLeftout[i] = Line1Samples.audio[i];
        codecBufferRightout[i] = Line1Samples.line[i];
        Line1Samples.line[i] = codecBufferLeftin[i];
        Line1Samples.audio[i] = codecBufferRightin[i];
    }

    // Get Ring Signal from DAA

    if( /* Poll Ring Detect Bit Here */ ){
        Line1Control.cidRingPolarity = 1; /* Ring present
    }
    else{
        Line1Control.cidRingPolarity = 0; /* Ring not present
    }

#ifdef USEGEC
    // Call Generic Echo Canceller
    for(i=0;i<5;i++) gecEchoCanceller(pgec1Data,&Line1Control,&Line1Samples);
#else
    for(i=0;i<5;i++) Line1Samples.leccid[i] = Line1Samples.line[i];
#endif

    // Call Type 1 and 2 Telephony Features Library
    Type12CID(pcid1Data,&Line1Control,&Line1Samples);

    // Do extension in use check. If no detection circuit is
    // present, always set Line1Control.NoExtFound = 1

    if(Line1Control.ExtUseCheck == 1){
        // Go onhook
        go_onhook();
    }
    else if(Line1Control.ExtUseCheck == 2){
        // Detect extension in use here.
        if( /* poll extension in use check here */ ){
            Line1Control.NoExtFound = 0;
        }
        else{
            Line1Control.NoExtFound = 1;
        }
    }
}

```

```

else if(Line1Control.ExtUseCheck == 3){
    // Go offhook
    go_offhook();
    Line1Control.ExtUseCheck=0;
}

// Process command for line flash

if(Line1Control.flashCommand){
    Line1Control.flashCommand = 0;
    if(Line1Control.flashPolarity == 1) go_offhook();
    else go_onhook();
}

#ifdef USEPARSER

// Use Type 1 and 2 Telephony Parser Library

CIDMessageParser(&ParserControl, &Line1Control);

if(ParserControl.FskParserLength != 0){
    /* Parsed Message Ready. Send to Output Device */
    if(ParserControl.ErrorType == 0){
        for( i = 0 ; i < ParserControl.FskParserLength ; i++)
            printf("%c",ParserControl.FskParserBuffer[i]);
        ParserControl.FskParserLength=0;
    }
}
#elseif

// Use Custom Parser

if(Line1Control.cidByteReady){
    /* Buffer cid bytes here*/
    cid_message_buffer[cid_message_index++] = Line1Control.cidByte;
}
if (Line1Control.messageDone){
    if(Line1Control.FrameErrors == 0){
        /* Call custom Parser here */
    }
    else Line1Control.FrameErrors = 0;
}
#endif

}

```

Code Example 8-5 shows a basic application using the *cid12.lib* module. This module is designed to interact with the Type 1 and 2 Telephony Parser Library (*cidparser.lib*); however, independent use of the *cid12.lib* module is supported. When receiving Type 2 Caller ID, it is recommended that an echo canceller be incorporated to attenuate network echo, most easily accomplished by using the *gec.lib* module. When implementing a full duplex speakerphone, the *cid12.lib* module is intended to be used with the *fdspk.lib* and *gec.lib* modules; however, custom line echo canceller and full duplex speakerphone software can also interface with the *cid12.lib* module.

This example shows how to manage Caller ID information using either the Type 1 and 2 Telephony Parser Library or a custom parser. When a custom parser is desired, the definition USEPARSER is undefined. When using a custom parser, the application must transfer the input bytes into a buffer, then process them. This is done automatically when using the Type 1 and 2 Telephony Parser Library.

While this module is primarily intended to be used for Caller ID reception, other convenient features are added to allow for simple integration into a Caller ID telephone. While the module is in an on-hook state (*Line1Control.hookSwitch* = 0), the *line[]* samples buffer will be filled with ring generator samples during power ringing. Otherwise, the *line[]* samples will be filled with zeros. These samples may act as an output to an audio channel (for example, to be played to a speaker).

While the module is in an off-hook state (*Line1Control.hookSwitch* = 1), the *audio[]* samples buffer will be overwritten with DTMF samples when a DTMF digit is requested for dialing, or to acknowledge the reception of a CAS tone, and will remain unchanged otherwise. The *line[]* samples remain unchanged during the off-hook state, unless *Line1Control.handsFreeLayer1* is set (a condition which is not shown in this example). In this case, the *gec[]* samples will be copied to the *line[]* samples structure. In addition, during off-hook FSK reception, both the *line[]* and *audio[]* samples will be muted (set equal to zero), regardless of the value of *Line1Control.handsFreeLayer1*. This is required since no signal should be transmitted by the CPE onto the telephone line or played on the speaker during data reception, as specified by the CIDCW standard.

The *gec[]* samples can be obtained by using the *gec.lib* module, or by using a separate custom echo canceller. The *audio[]* samples will be modified by the *fdspk.lib* module (or custom full duplex speakerphone software) in this case.

The input samples to the Type 1 and 2 Telephony Features Library CAS detector are taken from the *leccid[]* sample buffer. When using *gec.lib*, this buffer will be filled automatically with echo cancelled line samples. If no echo canceller is used, copy the five samples from the *line[]* buffer into the *leccid[]* buffer prior to calling *Type12CID()*.

The Type 1 and 2 Telephony Features Library supports all necessary timing and logic needed for the application to easily check if an extension phone is off-hook following reception of a CAS tone, as specified in SR-3004. Upon reception of a CAS tone, if an extension phone is found to be off-hook, the Type 1 and 2 Telephony Features Library will not send back a DTMF acknowledgement; therefore, no FSK Caller ID signal will be sent. By simply monitoring *ExtUseCheck*, the application will know when to go on-hook, when to monitor line voltage to determine if an extension is found and set *NoExtFound* accordingly, and when to go back off-hook. For more information regarding monitoring line voltage to determine if an extension is off-hook, see SR-3004. When taking the phone on-hook or off-hook as a result of a *ExtUseCheck* command, the application should not change the *Line1Control.hookSwitch* variable or call *Type12CIDinit*.

In this example, there are two GPIO pins: one output pin, which places the telephone interface on-hook or off-hook, and one input pin, which indicates the ring detect from the telephone interface. The ring detect pin is polled in the application and its polarity is placed in the variable *Line1Control.cidRingPolarity*. This information is necessary for the Type 1 and 2 Telephony Features Library, even when *disableRinger* is set (disabling the ring tone generator) and should be polled at a rate of 1600/second as shown in the application. However, slower polling rates may be tolerated.

It is assumed that the functions *go_onhook()* and *go_offhook()*, which are not shown in [Code Example 8-5](#), would simply set or clear the output pin connected to the telephone interface. However, note that these are not the same as when the application actually puts the module into an on-hook or off-hook state. In this situation, the application must also set or clear the *Line1Control.hookSwitch* variable, then call *Type12CIDinit()*.



The Codec Interrupt Service Routine is also not shown in this example. This routine is assumed to transmit and receive 8KHz samples from a codec and read/write to the buffers *codecBufferLeftin[]* and *codecBufferLeftout[]*. This codec is connected to the telephone line interface. This routine is also assumed to transmit and receive 8KHz samples from a codec and to read/write to the buffers *codecBufferRightin[]* and *codecBufferRightout[]*. This codec is connected to an audio interface. When the ISR has received/transmitted five samples, the buffers are full/empty, and the *SamplesReady* flag is set, indicating to the application that it should call *CalleridAppMain()*. The calling rate of this routine should be 1600 calls/second.

In the beginning of the *CalleridAppMain()* routine, the codec samples are copied into the module's sample structure. Note that for the output samples, the *line[]* samples are copied into the right (audio) codec channel, and the *audio[]* samples are copied into the left (line) codec channel. This sample "criss-cross" is indicative of the natural signal flow for a telephone.

Chapter 9

DSP56852 Applications

The SDK for the DSP56852 platform comes with several applications. These applications serve as a reference design to illustrate how a software developer can utilize the SDK features to build his unique applications. The following sections describe these applications in more detail.

9.1 BSP Demos

9.1.1 Codec Demo

The Codec Demo demonstrates the use of the Codec driver provided by the SDK by exercising the codec hardware on the DSP56852EVM board. For details on building and executing this application, see [Section 6.2.7](#).

9.1.2 COP Demo

The COP Demo demonstrates the use of the COP driver provided by the SDK by exercising the COP on the DSP56852. For details on building and executing this application, see [Section 5.1.4](#).

9.1.3 PC Master Software Demo

The PC Master Software Demo demonstrates the use of the PC master software driver provided by the SDK. For details on building and executing this application, see [Section 6.7.6](#).

9.1.4 Quad Timer Demo

The Quad Timer demo demonstrates the use of the Quad Timer driver provided by the SDK by exercising the quad timer on the DSP56852 to generate timed interrupts that flash LEDs; for details on how to build and execute the quad timer application, see [Section 5.4.7](#).

9.1.5 SCI Demo

The SCI demo demonstrates the use of the SCI driver provided by the SDK by exercising the SCI on the DSP56852 to read and write data to a PC application using a serial port. See [Section 5.5.7](#) for more information on the SCI application.

9.1.6 SCI Echo Demo

The SCI Echo demo demonstrates the use of the SCI driver provided by the SDK by exercising the SCI on the DSP56852 to echo data to the hyperterminal using a serial port. See [Section 5.5.7.2](#) for more information on the SCI application.

9.1.7 Timer Demo

The Timer demo demonstrates the use of the Timer library provided by the SDK. For details on building and executing this application, see [Section 5.8.3](#).

For more information about the Timer library, see **Timer Services** in the **SDK Programmer's Guide**.

9.2 Security Demos

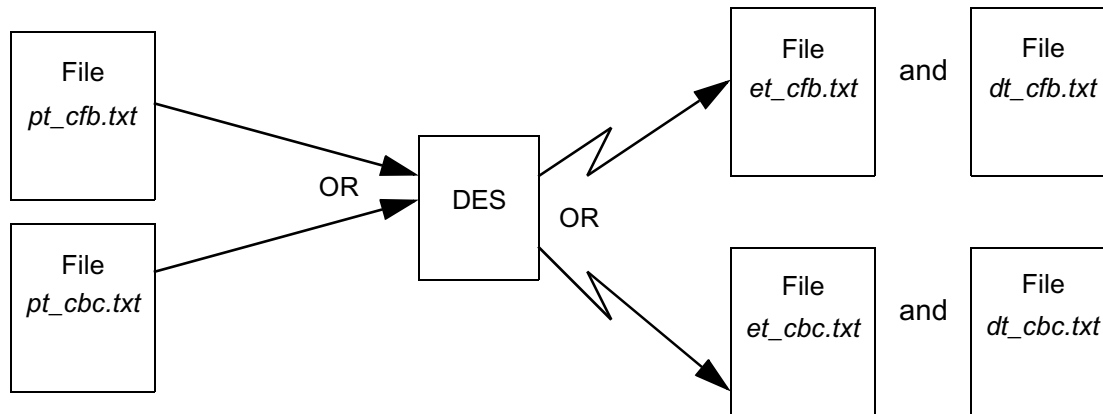
9.2.1 DES Demo

DES application is demonstrated in two ways; File I/O demo and a real-time demo.

9.2.1.1 DES File I/O demo

A demonstration of the DES application follows; for more information on DES, see the **Data Encryption Standard (DES) Library** manual.

The demo is based on file I/O. Two files, *pt_cfb.txt* and *pt_cbc.txt*, are used to store plain text for CFB and CBC modes, respectively. After encryption, the coded message is stored in *et_cfb.txt* or *et_cbc.txt*; after decryption, the decoded message is stored in *dt_cfb.txt* or *dt_cbc.txt*. The contents of *dt_cfb.txt* / *dt_cbc.txt* must be identical to that of the *et_cfb.txt* / *et_cbc.txt*, respectively. The contents of input files (i.e., the plaintext) could be text, numbers and/or special characters.


Figure 9-1. DES Demo

9.2.1.1.1 Set-up for DES File I/O Demo

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**.

File I/O:

This test is File I/O based.

- Connect the serial cable between the EVM and the COM1 port of the computer.
- Locate the files *pt_cfb.txt* and *pt_cbc.txt* in
`...\src\dsp56852evm\nos\applications\security\des\io\`
- These files may be altered if desired

9.2.1.1.2 Procedure for DES File I/O Demo

- Invoke the PC File I/O application by executing the following file:
`...\Embedded SDK\src\x86\win32\applications\fileio\fileio.exe`
- Modify the settings to configure the File I/O application to run at 56000bps
- Using CodeWarrior, open:
`...\nos\applications\security\des\des_demo.mcp`
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select Run in the debugger to continue executing the test
- The test results will be displayed on CodeWarrior's console
- After running the DES encryption and decryption services, the encrypted and decrypted files for CFB and CBC modes of operation of DES can be seen at the following location:
`...\src\dsp56852evm\nos\applications\security\des\io\`

9.2.1.2 DES Real-time demo

The DES Real-time Demo shows how to use the DES library provided by the SDK. This demo allows the user to enable and disable the DES algorithm when processing audio samples from the Codec. The processed audio samples are then sent to the Codec for the user to hear.

9.2.1.2.1 Set-up for DES Real-time Demo

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**. Toggle all three switches on the "Sample Rate" dip-switch (S4) to "Off", which will generate an 8000Hz sample rate. The demo set-up is shown in [Figure 9-2](#).

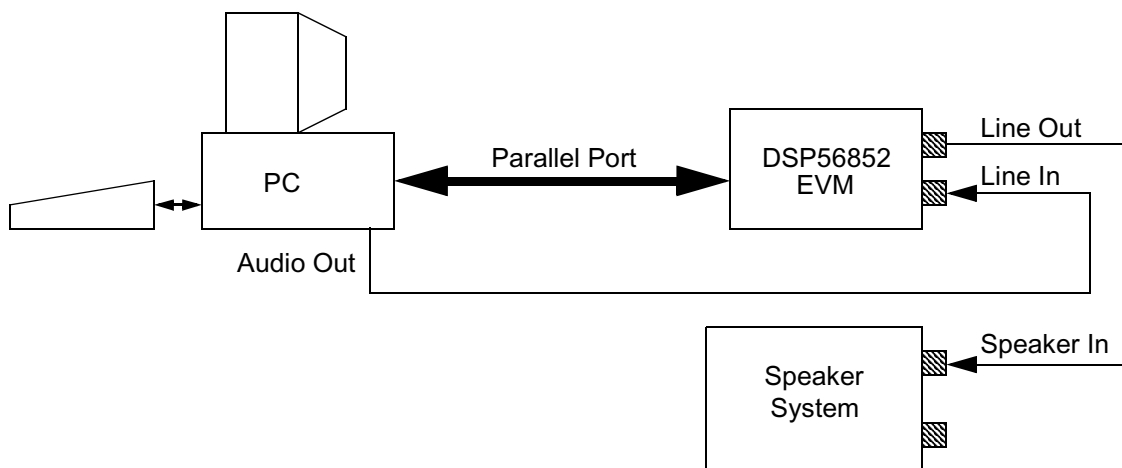


Figure 9-2. DES Real-time Demo Set-up

- Connect the parallel port of the PC to the parallel port of the EVM board
- Connect PC Audio Out to the Line In of the EVM board
- Connect Line Out of the EVM board to Speaker In of the speaker system

9.2.1.2.2 Procedure for DES Real-time Demo

- Using CodeWarrior, open:
`...\nos\applications\security\des_realtime\des_realtime_demo.mcp`
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the demo
- Once the debugger is running, send an audio input from the PC Audio Out port to the Line In of the EVM
- When the demo starts, DES Encryption and Decryption are performed on the input data
 - The Green LED should be on
 - The audio from the Codec should be intelligible
- The IRQA interrupt switch on the EVM board (IRQA Button) is used to disable and enable DES decryption
 - Press the IRQA Button to disable the decryption.
 - The input audio will appear unintelligible at the Line Out of the EVM
 - The Green LED should turn off
 - Press the IRQA Button again to enable the decryption
 - The input audio should be intelligible
 - The Green LED should be on

9.2.2 Triple DES Demo

The Triple DES (3DES) application is demonstrated in two ways: a File I/O demo and a real-time demo.

9.2.2.1 Triple DES File I/O demo

This application uses Triple Data Encryption Standard (3DES); for additional information on 3DES, see the **Triple Data Encryption Standard (3DES) Library** manual.

The following demo illustrates encryption and decryption modes of the Triple DES algorithm. The demo is based on file I/O. The file *plaintext.in* is used to store plaintext for 3DES. After encryption, the message is stored in *encr_txt.out*; after decryption, the message is stored in *decr_txt.out*. The contents of *decr_txt.out* must be identical to *plaintext.in*. The contents of the input file, the plaintext, could be text, numbers and/or special characters.

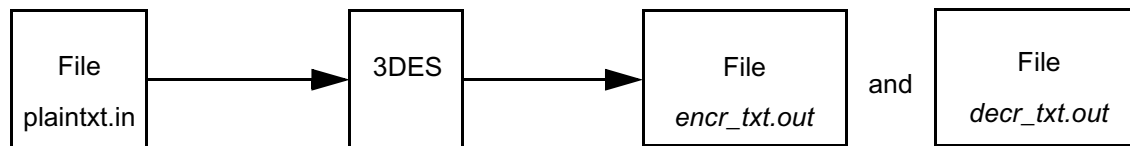


Figure 9-3. Triple DES Demo

9.2.2.1.1 Set-up for Triple DES Demo

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**.

File I/O:

This test is File I/O based.

- Connect the serial cable between the EVM and the COM1 port of the computer.
- Locate the file *plaintext.in* in:
`...\src\dsp56852evm\nos\applications\security\3des\io\`
- This file may be altered if desired
- If the user wants to change the input file *plaintext.in*, then the `#define PT_LEN` in the demo file *demo_3des.c* should be updated with the number of characters in the input file.
- To view the file *demo_3des.c*, see:
`...\src\dsp56852evm\nos\applications\security\3des\c_sources\`

9.2.2.1.2 Procedure for Triple DES Demo

- Invoke the PC File I/O application by executing the following file:
`...\Embedded SDK\src\x86\win32\applications\fileio\fileio.exe`
- Modify the settings to configure the File I/O application to run at 56000bps
- Using CodeWarrior, open:
`...\nos\applications\security\3des\3des_demo.mcp`

- Build and download the project; when the debugger reaches *main()*, it will stop
- Select Run in the debugger to continue executing the test
- The test results will be displayed on CodeWarrior's console
- After running 3DES encryption and decryption services, the encrypted and decrypted files can be viewed in this file:
...\\src\\dsp56852evm\\nos\\applications\\security\\3des\\io\\

9.2.2.2 Triple DES Real-time demo

The 3DES Real-time Demo shows how to use the 3DES library provided by the SDK. This demo allows the user to enable and disable the 3DES algorithm when processing audio samples from the codec. The processed audio samples are then sent to the codec for the user to hear.

9.2.2.2.1 Set-up for Triple DES Real-time Demo

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**. Toggle all three switches on the "Sample Rate" dip-switch (S4) to "Off", which will generate an 8000Hz sample rate. The demo set-up is shown in [Figure 9-4](#).

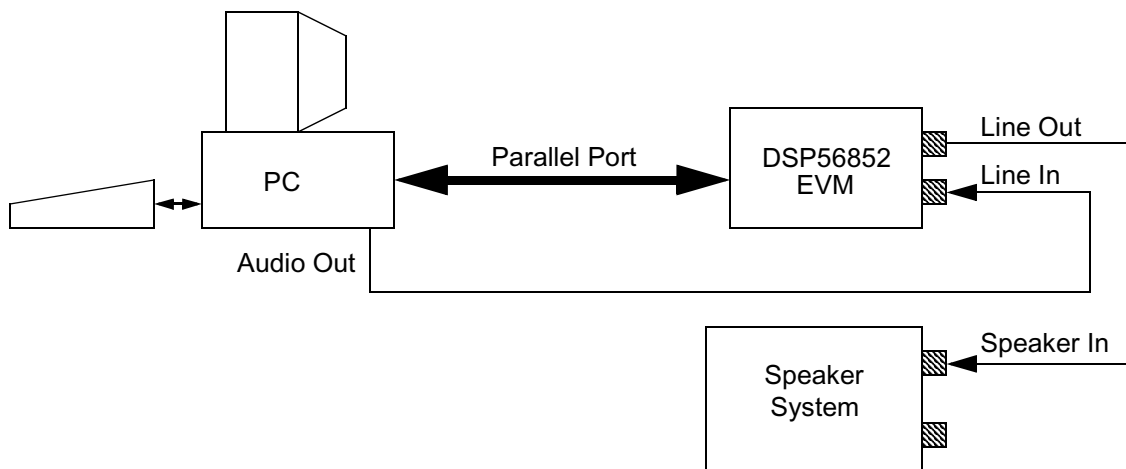


Figure 9-4. 3DES Demo Set-up

- Connect the parallel port of the PC to the parallel port of the EVM board
- Connect PC Audio Out to the Line In of the EVM board
- Connect Line Out of the EVM board to Speaker In of the speaker system

9.2.2.2.2 Procedure for Triple DES Real-time Demo

- Using CodeWarrior, open:
...\\nos\\applications\\security\\3des_realtime\\3des_realtime_demo.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the demo
- Once the debugger is running, send an audio input from the PC Audio Out port to the Line In of the EVM

- When the demo starts, the 3DES Encryption and Decryption are performed on the input data
 - The Green LED should be on
 - The audio from the Codec should be intelligible
- The IRQA interrupt switch on the EVM board (IRQA Button) is used to disable and enable 3DES decryption
 - Press the IRQA Button to disable the decryption
 - The input audio will appear unintelligible at the Line Out of the EVM
 - The Green LED should turn off
 - Press the IRQA Button again to enable the decryption
 - The input audio should be intelligible
 - The Green LED should be on

9.2.3 RSA Demo

Additional information about RSA can be found in the document **RSA Library**.

The demo is based on file I/O. The file *m.in* is used to store message words for RSA. After encryption, the coded message is stored in *m_Enc.out*; after decryption, the decoded message is stored in *m_Dec.out*. The contents of *m_Dec.out* must be the same as that of *m.in*. The contents of the input file (i.e., the message) could be text, numbers and/or special characters.

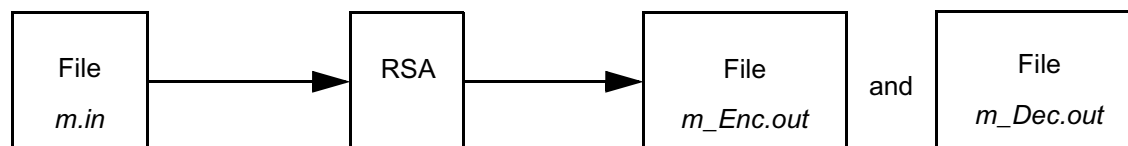


Figure 9-5. RSA Demo

9.2.3.1 Set-up for RSA Demo

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**.

File I/O:

This test is File I/O based.

- Connect the serial cable between the EVM and the COM1 port of the computer
- Locate the file *m.in* in:
`...\src\dsp56852evm\nos\applications\security\rsa\io\`
- This file may be altered if desired
- If the user wants to change the input file *m.in*, then the `#define NO_CHARS` in the demo file *demo_rsa.c* should be updated with the number of characters in the input file. The *demo_rsa.c* file can be found in:
`...\src\dsp56852evm\nos\applications\security\rsa\c_sources\`

9.2.3.2 Procedure for RSA Demo

- Invoke the PC File I/O application by executing the following file:
...\\Embedded SDK\\src\\x86\\win32\\applications\\fileio\\fileio.exe
- Modify the settings to configure the File I/O application to run at 56000bps
- Using CodeWarrior, open:
...\\nos\\applications\\security\\rsa\\rsa_demo.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select Run in the debugger to continue executing the test
- The test results will be displayed on CodeWarrior's console
- After running RSA encryption and decryption services, the encrypted and decrypted files *m_Enc.out* and *m_Dec.out*, respectively, can be viewed in:
...\\src\\dsp56852evm\\nos\\applications\\security\\rsa\\io\\

9.3 Telephony Demos

9.3.1 Automatic Gain Control (AGC) Demo

The AGC Demo demonstrates the use of the AGC library provided by the SDK. This demo allows the user to enable and disable the automatic gain control algorithm when processing audio samples from the codec. The processed audio samples are then sent to the codec for the user to hear.

9.3.1.1 Set-up for AGC Demo

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**. Toggle all three switches on the "Sample Rate" dip-switch (S4) to "Off", which will generate an 8000Hz sample rate. The demo set-up is shown in [Figure 9-6](#).

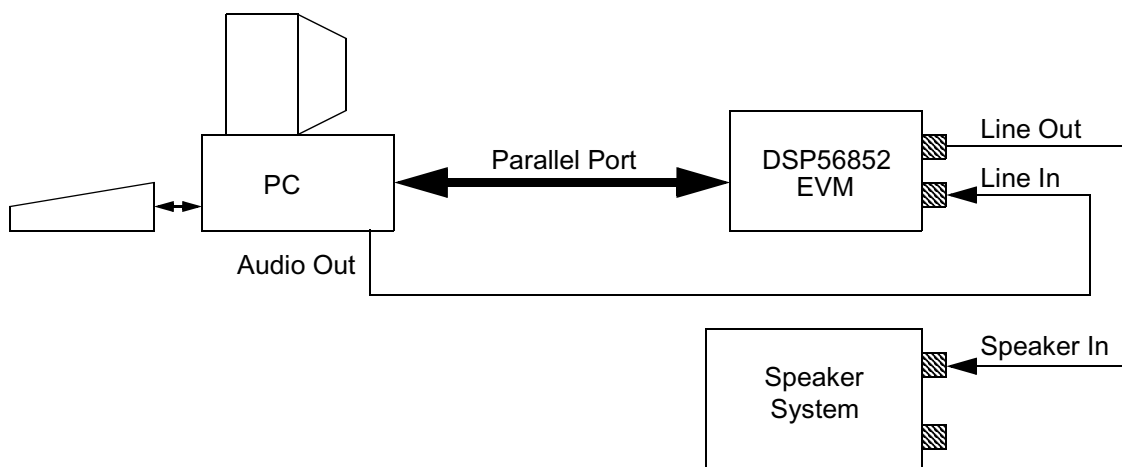


Figure 9-6. AGC Demo Set-up

- Connect the parallel port of the PC to the parallel port of the EVM board
- Connect PC Audio Out to the Line In of the EVM board
- Connect Line Out of the EVM board to Speaker -In of the speaker system

9.3.1.2 Procedure for AGC Demo

- Using CodeWarrior, open:
`...\nos\applications\telephony\agc\demo_agc.mcp`
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the demo
- Once the debugger is running, send an audio input from the PC Audio Out port to the Line In of the EVM board for which automatic gain is to be controlled
 - The audio input file is provided in:
`...\nos\applications\telephony\agc\io`
- **When the demo starts, the AGC is disabled by default**
 - The IRQA interrupt switch on the EVM board (IRQA Button) is used to enable/disable the AGC
 - When the AGC is disabled, the input audio will appear unaltered at the Line Out of the EVM.
 - To enable the AGC, press the IRQA Button once.
 - After the AGC is enabled, the gain-controlled audio output will be played on the speakers
 - The Green LED should turn on
 - By again pressing the IRQA Button once, the AGC can be disabled
 - The Green LED should turn off

9.3.2 CPT Demo

In this demo, a tone is fed to the algorithm through the specified API's. When a valid tone is detected, the demo is terminated and the detected tone is displayed on the console.

9.3.2.1 Set-up for CPT Demo

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**. Toggle all three switches on the "Sample Rate" dipswitch (S4) to "Off", which will generate an 8000Hz sample rate.

Detecting call progress tones requires a telephone interface to the EVM board, which is set up as shown in **Figure 9-7**. A device called Modem Mate from RadioShack is used in this interface.

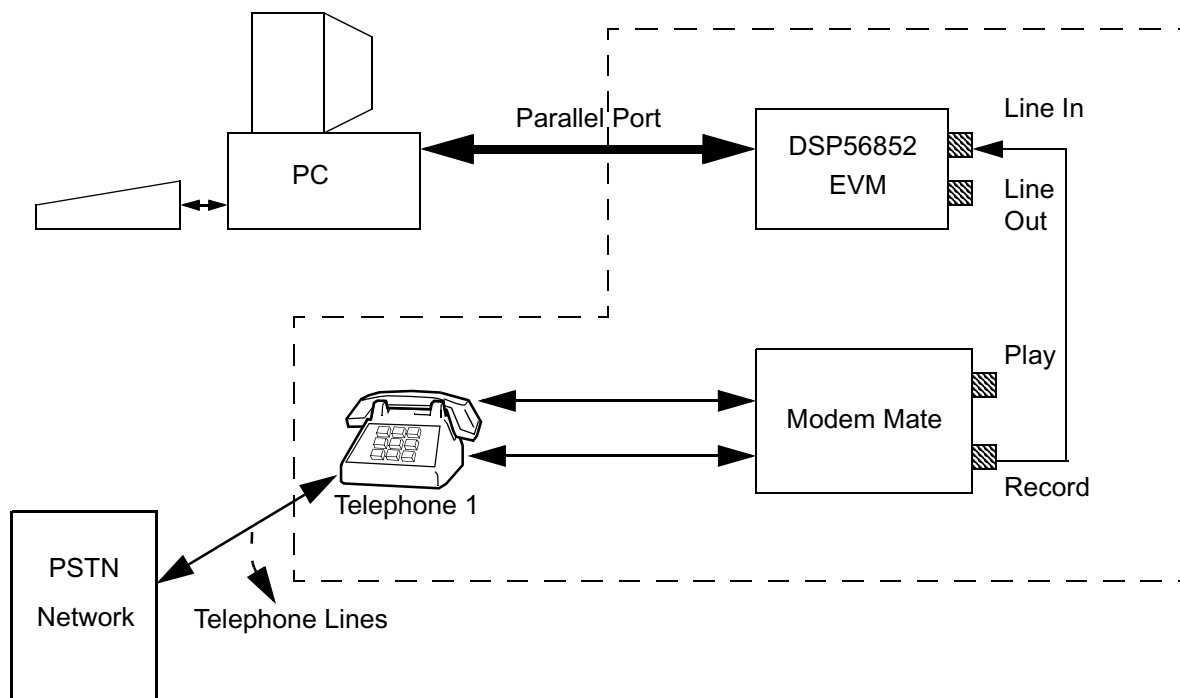


Figure 9-7. CPT Detection Demo Set-up

- As shown in [Figure 9-7](#), the Line In analog signal of the EVM is connected to the Record terminal of the Modem Mate
- The telephone handset is connected to the Modem Mate handset input
- The 4-wire cord coming out of the Modem Mate is connected to the telephone's handset input

9.3.2.2 Procedure for CPT Demo

- Using CodeWarrior, open:
`...\nos\applications\telephony\demo_cpt\demo_cpt.mcp`
- Build and download the project; when the debugger reaches `main()`, it will stop
- Select Run in the debugger to continue executing the demo
- To test any tone, the tone must be generated while the application is running
- When detected, the tone is displayed on the console and the demo is terminated
 - For example, to test for dial tone:
 - Go off-hook to generate the dial tone
 - The application will detect the dial tone and display the result on the console
 - Repeat this procedure for each tone to be detected

9.3.3 Common Tone Generation Demo

The Common Tone Generation (CTG) Demo, which runs on the DSP56852 platform, generates North American Call Progress Tones with specifications for:

- Dial Tone
- Message Waiting Indicator Tone
- Recall Dial Tone / Confirmation Tone
- Line Busy Tone
- Re-Order Tone
- Audible Ringing Tone

After generation, the demo feeds the output samples to a speaker via a codec.

9.3.3.1 Set-Up for CTG Demo

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**. Toggle all three switches on the "Sample Rate" dipswitch (S4) to the off-position which will generate a 8000 Hz sample rate.

Running the Common Tone Generation Test:

- Connect the EVM's Line Out to a speaker
- Locate *demo_ctg.mcp*, found in:
...nos\applications\telephony\demo_ctg\
- Once accessed, press function key [F7] to begin the build
- Select *Enable debugger* from the Metrowerks *Project* menu
- Press function key [F5]
- You can hear the specified tones in the speaker
- Once the demo has run, the message "Demo Completed" is flagged out on the MW console

9.3.4 DTMF Detection Demo

This demo includes dialing the phone with a particular key, detecting, and displaying the same key on the console. For more information on DTMF Detection, see the **DTMF Detection Library** manual.

The demo is set up so that the DTMF signals generated by the telephone set are given as inputs to the DTMF Detection algorithm. The detected keys are displayed on CodeWarrior's console when they are detected. The key displayed on the console should match those dialed.

9.3.4.1 Set-up for DTMF Detection Demo

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**. Toggle all three switches on the "Sample Rate" dipswitch (S4) to "Off", which will generate an 8000Hz sample rate.

Detecting DTMF requires a telephone interface to the EVM board, which is set up as shown in [Figure 9-8](#). A device called Modem Mate from RadioShack is used in this interface.

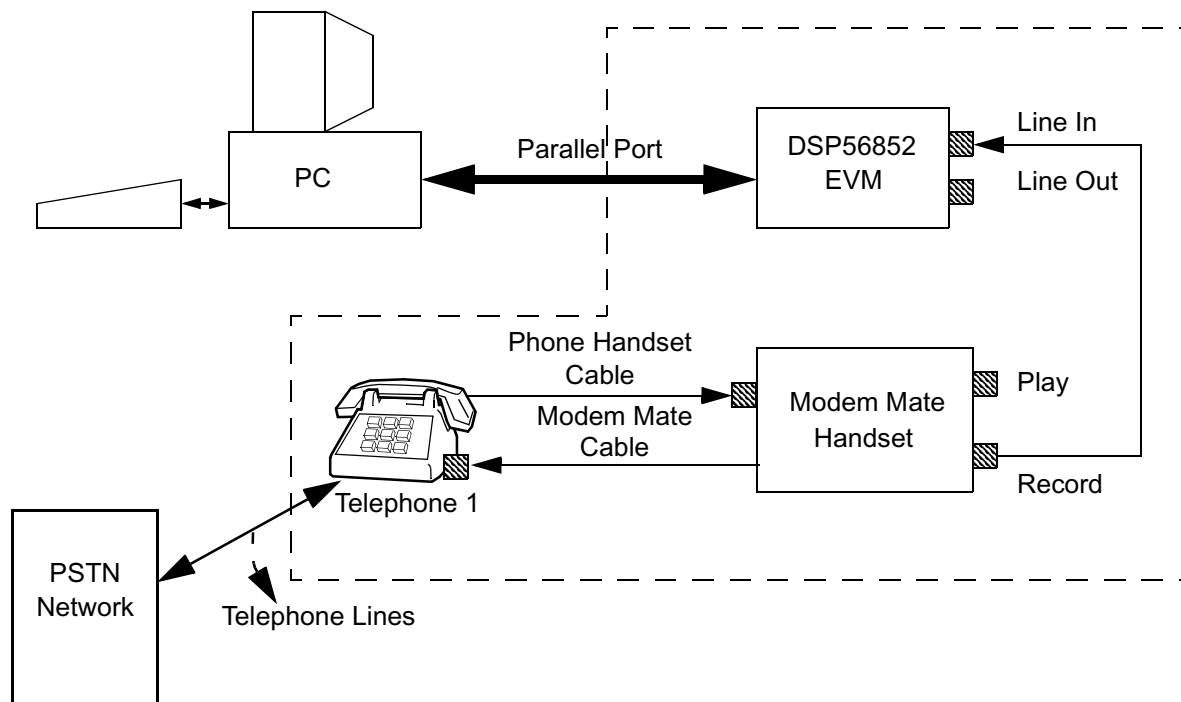


Figure 9-8. DTMF Detection Demo Set-up

- As shown in [Figure 9-8](#), the Line In analog signal of the EVM board is connected to the Record terminal of the Modem Mate
- The handset of the telephone is connected to the Modem Mate's handset input
- The 4-wire cord coming out of Modem Mate is connected to the telephone's handset input
- The telephone to which the EVM is interfaced should be on-hook

9.3.4.2 Procedure for DTMF Detection Demo

- Using CodeWarrior, open:
`...\nos\applications\telephony\dtmf_det\demo_dtmf_det.mcp`
- Build and download the project; when the debugger reaches `main()`, it will stop
- Select Run in the debugger to continue executing the demo
- Once the debugger is running, use the telephone keypad to input DTMF signals to the DTMF Detection algorithm
- Press the digits to be detected on the telephone keypad, followed by the star (*) key
- The detected DTMF digits are displayed on CodeWarrior's console

9.3.5 DTMF Generation Demo

The DTMF Generation Demo, which runs on the DSP56852 EVM, includes entering keys through a keyboard, and dialing the phone with corresponding numbers.

The demo set-up uses a serial cable to connect a PC to the DSP56852EVM board. The user types the DTMF characters through a hyperterminal. The execution of the DTMF Generation algorithm generates the audible tones, which can be heard through a speaker or used to actually dial the phone.

For more information about the DTMF Generation library, see the **DTMF Generation Library** manual.

9.3.5.1 Set-up for DTMF Generation Demo

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**. Toggle all three switches on the "Sample Rate" dipswitch (S4) to "Off", which will generate an 8000Hz sample rate.

Generating DTMF requires a telephone interface to the EVM board, which is set up as shown in **Figure 9-9**. A device called Modem Mate from RadioShack is used in this interface.

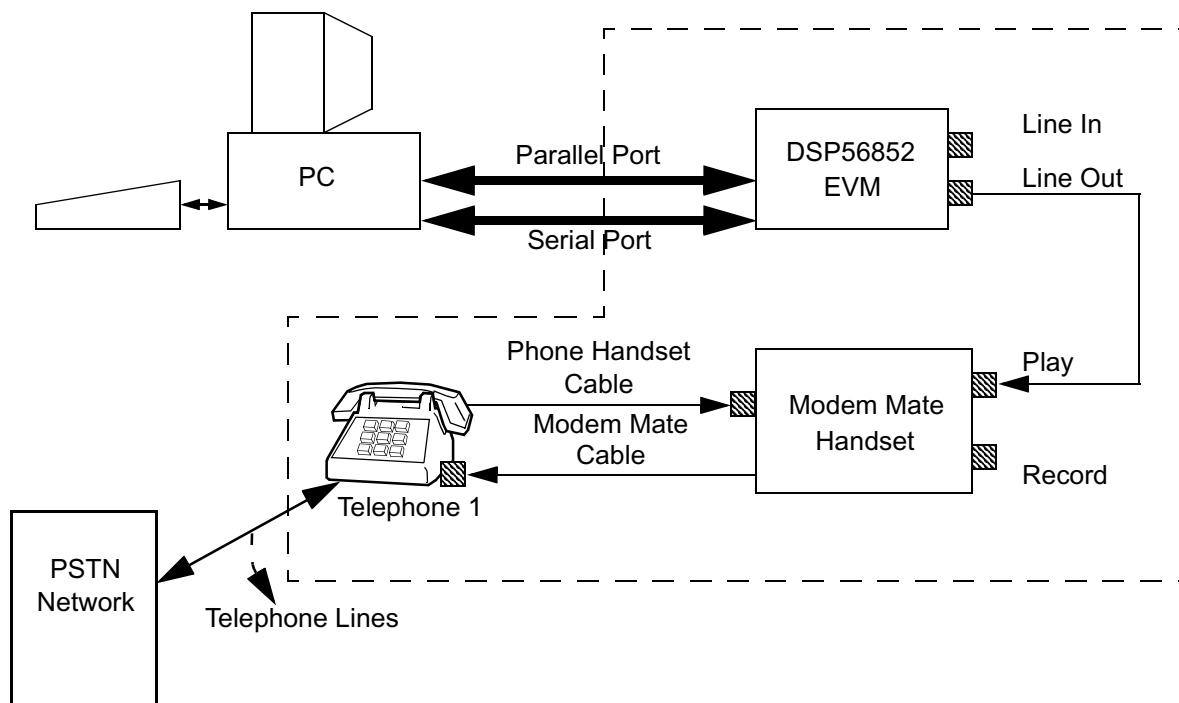


Figure 9-9. DTMF Generation Demo Set-up

- As shown in **Figure 9-9**, the Line Out analog signal of the EVM board is connected to the Play terminal of the Modem Mate
- The handset of the telephone is connected to the Modem Mate's handset input
- The 4-wire cord coming out of Modem Mate is connected to the telephone's handset input
- The serial port of the PC (COM 1) is connected to the serial port on the EVM board
- The telephone to which the EVM is interfaced should be on-hook
- Execute HyperTerminal on the PC
- Configure HyperTerminal for:
 - Baud Rate: 9600
 - Data Bits: 8
 - Parity: None
 - Stop Bit: 1

Note: If you do not have access to a telephone interface such as a Modem Mate, you can plug in a speaker to the Line Out jack of the codec. Hold a telephone handset up to the speaker to dial the telephone as DTMF tones are generated.

9.3.5.2 Procedure for DTMF Generation Demo

- Using CodeWarrior, open:
...\\nos\\applications\\telephony\\dtmf_gen\\demo_dtmf_gen.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select Run in the debugger to continue executing the demo
- Once the debugger is running, lift the telephone connected to the EVM and **immediately** (before dial tones die out) type the number you would like to dial through your keyboard into the hyperterminal program
- The demo should generate DTMF tones for each key pressed
- If all works well, the phone you are dialing should ring
- If you press the “Enter” key on the keyboard, the demo will stop executing

9.3.6 Feature Phone Application

The Feature Phone Application is detailed in the **Feature Phone Application for the DSP56852** manual.

9.3.7 G.165 Demo

G.165 offers three services:

- Echo Cancellation
- Tone Disabler
- Hold Release Logic

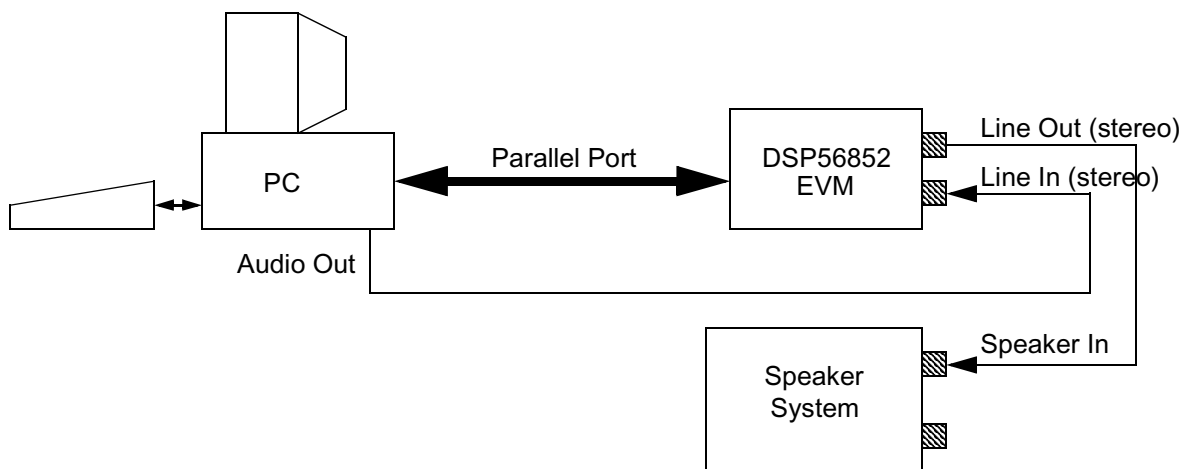
The G.165 demo is intended to demonstrate the accuracy of G.165 echo cancellation, hold release logic and tone disabler logic algorithms.

Additional information on G.165 is found in the **G.165 Library** manual.

9.3.7.1 Set-up for G.165 Demo

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User’s Manual**.


Figure 9-10. G.165 Demo Set-up

- Connect the parallel port of the PC to the parallel port of the EVM board
- Connect PC Audio Out to the Line In of the EVM board
- Connect Line Out of the EVM board to Speaker In of the speaker system

9.3.7.2 Procedure for G.165 Demo

- Locate the speech files (*sample1.wav*, *sample2.wav*) present at :
..\nos\application\telphony\g165\inputs
- Play one of the speech (.wav) files in a loop (continuous) mode.
- Using CodeWarrior, open:
..\nos\application\telphony\g165\demo_g165.mcp
- Set the #define BYPASS_ECHOCANCELLER to 1, in the file *g165_speech.c* located at:
..\nos\application\telphony\g165\c_sources
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select Run in the debugger to continue executing the demo
- You will hear conversation **with** echo
- Now kill the debugger and change #define BYPASS_ECHOCANCELLER to 0, in the file *g165_speech.c*
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select Run in the debugger to continue executing the demo
- You will hear conversation **without** echo

9.3.8 G.168 Demo

G.168 offers three services:

- Echo Cancellation
- Tone Disabler
- Hold Release Logic

The G.168 demo is intended to demonstrate only the accuracy of G.168 echo cancellation algorithm.

Additional information on G.168 is found in the **G.168 Library** manual.

9.3.8.1 Set-up for G.168 Demo

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**.

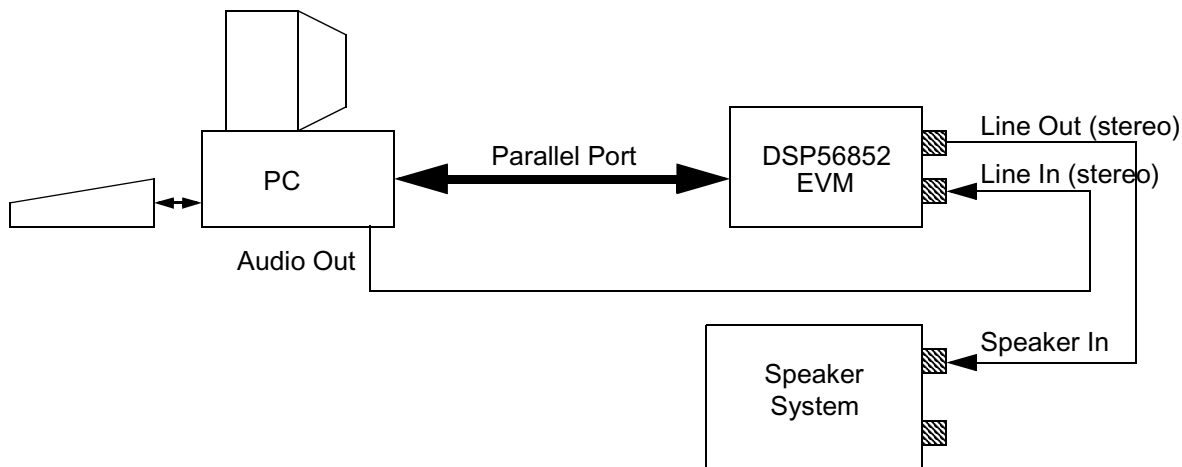


Figure 9-11. G.168 Demo Set-up

- Connect the parallel port of the PC to the parallel port of the EVM board
- Connect PC Audio Out to the Line In of the EVM board
- Connect Line Out of the EVM board to Speaker In of the speaker system

9.3.8.2 Procedure for G.168 Demo

- Locate the speech files (*sample1.wav*, *sample2.wav*) present at :
`..\nos\applications\telephony\g168\inputs`
- Play one of the speech (.wav) files in a loop (continuous) mode.
- Using CodeWarrior, open:
`..\nos\applications\telephony\g168\demo_g168.mcp`
- Set the #define BYPASS_ECHOCANCELLER to 1 in the file *g168_speech.c*, located at:
`..\nos\applications\telephony\g168\c_sources`
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select Run in the debugger to continue executing the demo
- You will hear conversation **with** echo
- Now kill the debugger and change #define BYPASS_ECHOCANCELLER to 0, in the file *g168_speech.c*
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select Run in the debugger to continue executing the demo
- You will hear conversation **without** echo

9.3.9 G.711 Demo

The G.711 Demo demonstrates the use of the G.711 library provided by the SDK.

For more information about the G.711 library, see the **G.711 Library** manual.

9.3.9.1 Set-up for G.711 Demo

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**. Toggle all three switches on the "Sample Rate" dipswitch (S4) to "Off", which will generate an 8000Hz sample rate.

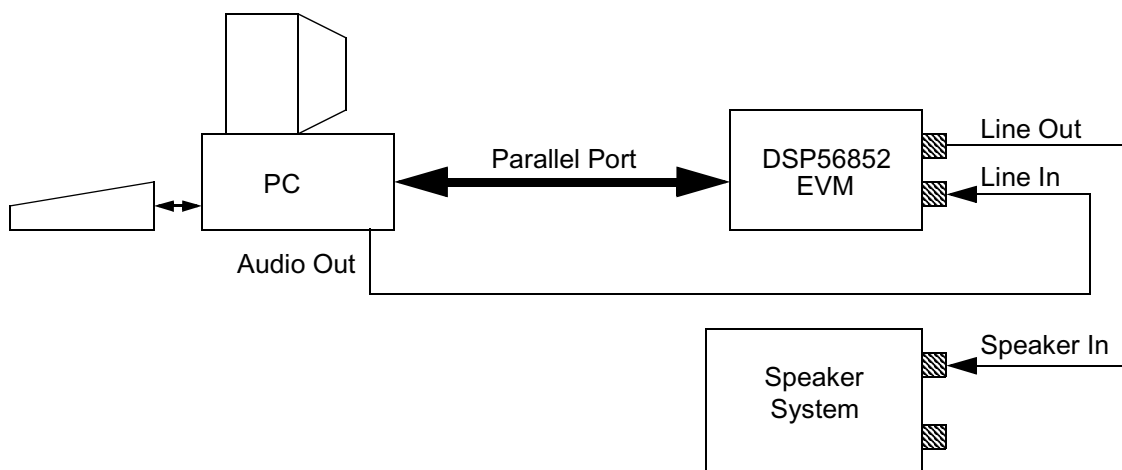


Figure 9-12. G.711 Demo Set-up

- Connect the parallel port of the PC to the parallel port of the EVM board
- Connect PC Audio Out to the Line In of the EVM board
- Connect Line Out of the EVM board to Speaker In of the speaker system

9.3.9.2 Procedure for G.711 Demo

- Using CodeWarrior, open:
`...\nos\applications\telephony\g711\demog711.mcp`
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select Run in the debugger to continue executing the demo
- Once the debugger is running, send an audio input from the PC Audio Out port to the Line In of the EVM board
- The same audio signal should be available at the speaker system output

9.3.10 G.726 Demo

The G.726 Demo demonstrates use of the G.726 library provided by the SDK.

In this demo, audio is input from a PC to the G.726 ADPCM Codec (encoder and decoder) algorithm and the decoded output is played by the G.726 ADPCM Codec on a speaker system.

For more information about the G.726 library, see the **G.726 Library** manual.

9.3.10.1 Set-up for the G.726 Demo

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**. Toggle all three switches on the "Sample Rate" dipswitch (S4) to "Off", which will generate an 8000Hz sample rate.

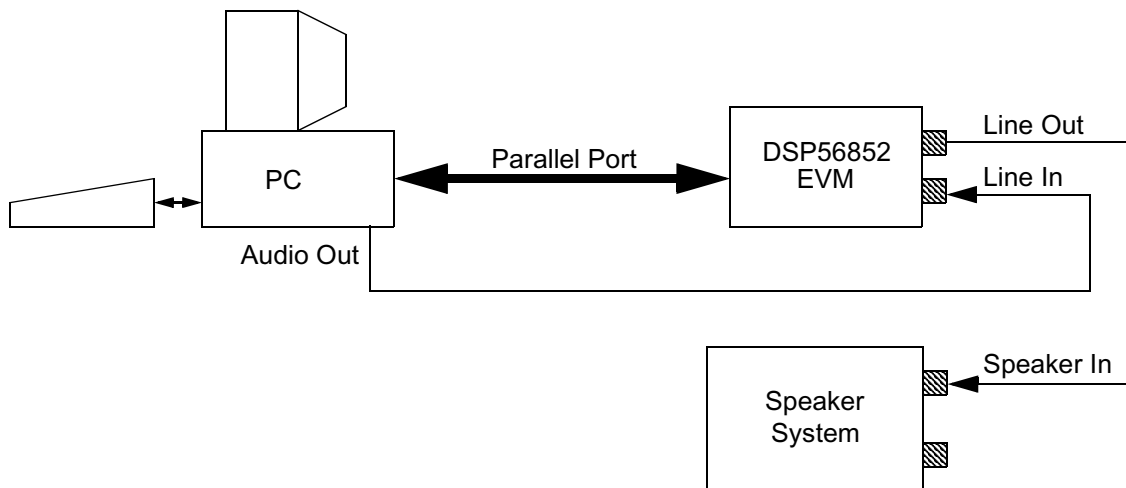


Figure 9-13. G.726 ADPCM Codec Test Set-up

- Connect the parallel port of the PC to the parallel port of the EVM board
- Connect PC Audio Out to the Line In of the EVM board
- Connect Line Out of the EVM board to Speaker In of the speaker system

9.3.10.2 Procedure for G.726 Demo

- Using CodeWarrior, open:
`...\\nos\\applications\\telephony\\g726\\demo_g726.mcp`
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select Run in the debugger to continue executing the demo
- Once the debugger is running, send an audio input from the PC Audio Out port to the Line In of the EVM board
- The same audio signal should be available at the speaker system output, with an increase in noise level as the rate (40, 32, 24, 16Kbits/s) decreases from 40 to 16Kbits/s

9.3.11 Noise Suppression (NS) Demo

The Noise Suppression Demo demonstrates the use of the NS library provided by the SDK. This demo allows the user to enable and disable the noise suppression algorithm when processing audio samples from the codec. The processed audio samples are then sent to the codec for the user to hear.

9.3.11.1 Set-up for NS Demo

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**. Toggle all three switches on the "Sample Rate" dip-switch (S4) to "Off", which will generate an 8000Hz sample rate. The demo set-up is shown in [Figure 9-14](#).

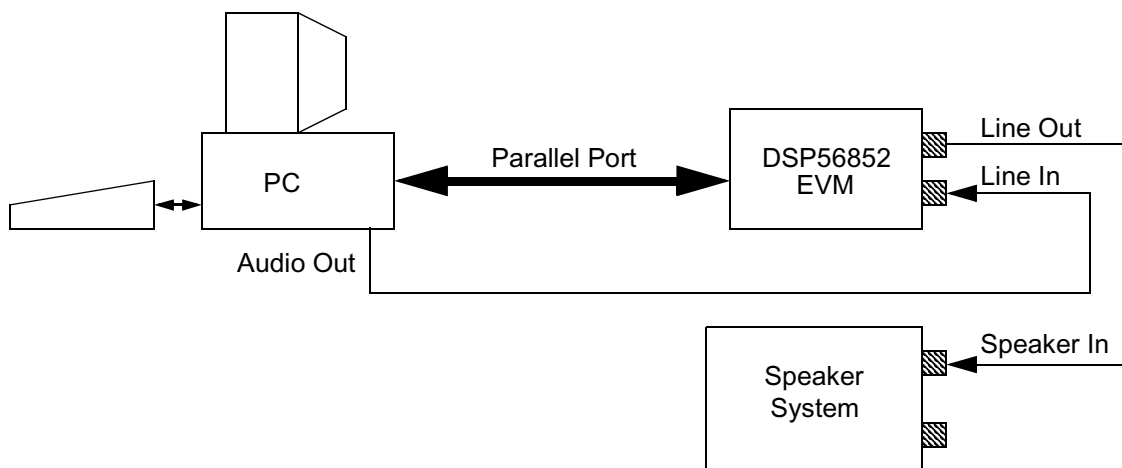


Figure 9-14. NS Demo Set-up

- Connect the parallel port of the PC to the parallel port of the EVM board
- Connect PC Audio Out to the Line In of the EVM board
- Connect Line Out of the EVM board to Speaker In of the speaker system

9.3.11.2 Procedure for NS Demo

- Using CodeWarrior, open:
`...\nos\applications\telephony\ns\demo_ns.mcp`
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the demo
- Once the debugger is running, send an audio input from the PC Audio Out port to the Line In of the EVM board for which noise is to be suppressed
- Audio files that contain noise are provided in:
`...\nos\applications\telephony\ns\inputs`
- When the demo starts, Noise Suppression is disabled by default
- The IRQA interrupt switch on the EVM board (IRQA Button) is used to enable/disable Noise Suppression
 - When the Noise Suppression is disabled, the input audio will appear unaltered at the Line Out of the EVM
 - To enable Noise Suppression, press the IRQA Button once
 - After Noise Suppression is enabled, the noise-suppressed audio output will be played on the speakers
 - The Green LED should turn on
 - By again pressing the IRQA Button once, Noise Suppression can be disabled
 - The Green LED should turn off

9.3.12 VAD Demo

The Voice Activity Detector, VAD, service detects speech and silence frames in the speech signal. This section describes how to set up and execute the VAD Application.

More information about VAD can be found in the **VAD Library** manual.

9.3.12.1 Set-up for VAD Demo

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**.

File I/O:

This test is File I/O based.

Connect the serial cable between the EVM and the COM1 port of the computer

The VAD flag data is stored in the file *vad.ref*, found in:

...\src\dsp56852evm\nos\applications\telephony\demo_vad\inputs\

The speech data *speech.in* and the playable format of the file *speech.au* are both stored in the directory *inputs*, found in:

...\src\dsp56852evm\nos\applications\telephony\demo_vad\inputs\

9.3.12.2 Procedure for VAD Demo

- Invoke the PC File I/O application by executing the following file:
...\Embedded SDK\src\x86\win32\applications\fileio\fileio.exe
- Modify the settings to configure the File I/O application to run at 56000bps
- Using CodeWarrior, open:
...\nos\applications\telephony\demo_vad\demo_vad.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select Run in the debugger to continue executing the demo
- Once the debugger is running, the concatenated speech frames are stored in *conc_spch.out* and the silence frames are discarded
— To find the file *conc_spch.out*, access this path:
...\src\dsp56852evm\nos\applications\telephony\demo_vad\outputs\
- Convert the output file *conc_spch.out* to playable format (.au) with the utility *Conv2au.exe*, located in file:
...\src\dsp56852evm\nos\applications\telephony\demo_vad\outputs\
- Play *speech.au* to hear the speech with silence regions, then play the concatenated output *conc_spch.au* (generated by using utility *Conv2au.exe*)
- Confirm that the speech frames are correctly retained and the silence regions are discarded

9.4 Modem Demos

9.4.1 V.22bis and Demo

Two examples of the V.22bis library are discussed in the following sections. More information about the V.22bis is available in the **V.22bis Library** manual.

In the demo, the V.22bis application is connected to a standard modem to test for interoperability. The standard modems that have been tested are a Motorola Premier 33.6kbps modem and a Psion Dacom Gold Card Global 56k+fax modem.

9.4.1.1 V.22bis Demo

The V.22bis library implements only the data-pump. Since V.22bis is a low bit-rate modem, no other higher layer protocol is required (except V.14, which implements sync to async and vice-versa, to accept and deliver data to and from the PC to the DSP EVM) to demonstrate text / data transfer between two hosts using hyperterminal.

The V.22bis modem operates in two modes: call modem and answer modem. In call modem, the V.22bis modem initiates the call by dialing the other modem; in answer modem, it is reversed. Both demos are explained in the following sections. The V.22bis library has neither dialup capability nor ring detect capability, so manual intervention is needed for the demo.

9.4.1.2 Hardware set-up

To connect the V.22bis data-pump to a standard modem, telephone line interface must be provided to the EVM board, as shown in **Figure 9-15**. A device called Modem Mate from RadioShack is used for this interface.

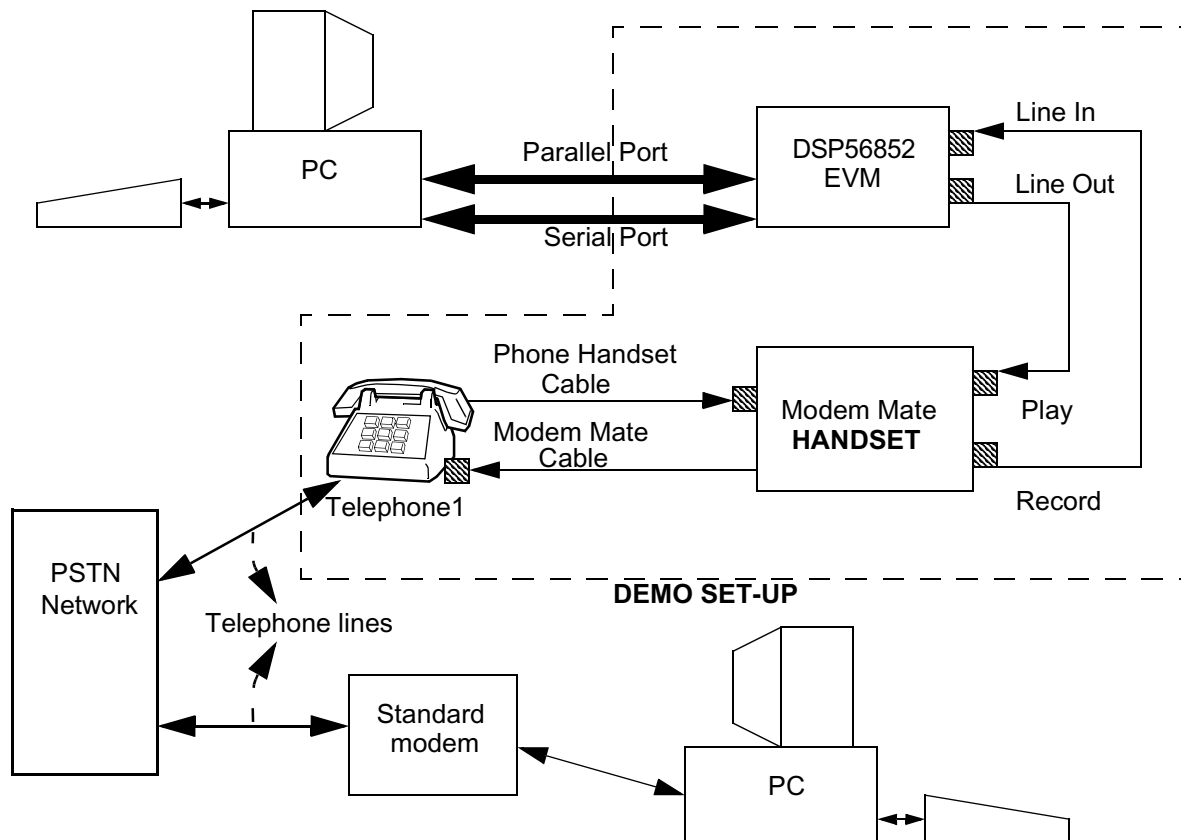


Figure 9-15. V.22bis Demo Set-up

- As shown in [Figure 9-15](#), the Line In and Line Out analog signals of the EVM are connected to the Modem Mate's Record and Play terminals, respectively
- The phone handset cable of the telephone is connected to the Modem Mate's HANDSET input
- The Modem Mate HANDSET Cable is connected to the telephone's handset input

9.4.1.3 Answering Modem Demo for 2400bps

Follow these steps to connect the V.22bis 2400bps soft-modem as the Answering modem and the standard (remote) modem as the Calling modem. This will test the asynchronous mode of operation.

- Load `...\nos\applications\modem\v22bis\interopans.mcp` onto Metrowerks' CodeWarrior and build the target. The telephone to which the EVM is interfaced should be on-hook.
- Invoke the HyperTerminal on the PC to which the EVM board is connected and configure it to:
 - 115200bps
 - 8 bits
 - 1 stop bit
 - no parity
 - flow control = none

- On the settings panel, enable terminal keys and ANSI emulation and set backscroll buffer lines to 100 (setting can be any number)
- On the ASCII set-up, enable send line ends with line feeds; echo typed characters locally
 - Append line feeds to incoming line ends and wrap lines that exceed terminal width
- After all the above settings, on the call menu, first disconnect and then connect
- On the remote side, connect the standard modem to the PC COM port. Invoke the hyperterminal on the PC. The hyperterminal is used to communicate (transmit AT Commands) to the standard modem. Configure the hyperterminal to:
 - **2400bps**
 - 8 bits
 - 1 stop bit
 - no parity
 - flow control = hardware
- On the settings panel, enable terminal keys and ANSI emulation and set backscroll buffer lines to 100 (setting can be any number)
- On the ASCII set-up, enable send line ends with line feeds; echo typed characters locally. Append line feeds to incoming line ends and wrap lines that exceed terminal width
- After the above settings, on the call menu, first disconnect and then connect
- Configure the standard modem
 - Disable V.42, an error-correcting protocol, which will enable only V.14 (sync to async and vice-versa) protocol
 - Type “at\n0”
 - Enable V.22bis in **2400bps** mode
 - For Motorola Premier 33.6kbps modem: Type “at%l3%b3”
 - For Psion Dacom Gold Card Global 56k+fax modem: Type “at+ms=2,0,2400,2400,0,0,2400”
- Make the standard modem go off-hook and dial the telephone connected to the EVM
 - Type “atdt” followed by the number of the telephone to which the EVM is connected
- Lift the telephone connected to the EVM and **immediately** run the test from CodeWarrior
- Both the modems should now negotiate and connect for V.22bis 2400bps modulation scheme
- Full duplex communication is now possible between the two hyperterminals

9.4.1.4 Answering Modem Demo for 1200bps

Follow these steps to connect the V.22bis 1200bps soft-modem as the Answering modem and the standard (remote) modem as the Calling modem. This will test the asynchronous mode of operation.

- Load ...\\nos\\applications\\modem\\v22bis\\interopans.mcp onto Metrowerks' CodeWarrior and build the target. The telephone to which the EVM is interfaced should be on-hook.
- Invoke the hyperterminal on the PC to which the EVM board is connected and configure it to:
 - 115200bps
 - 8 bits

- 1 stop bit
 - no parity
 - flow control = none
- On the settings panel, enable terminal keys and ANSI emulation and set backscroll buffer lines to 100 (setting can be any number)
- On the ASCII set-up, enable send line ends with line feeds; echo typed characters locally. Append line feeds to incoming line ends and wrap lines that exceed terminal width
- After all the above settings, on the call menu, first disconnect and then connect
- On the remote side, connect the standard modem to the PC COM port. Invoke the hyperterminal on the PC. The hyperterminal is used to communicate (transmit AT Commands) to the standard modem. Configure the hyperterminal to:
 - **1200bps**
 - 8 bits
 - 1 stop bit
 - no parity
 - flow control = hardware
- On the settings panel, enable terminal keys and ANSI emulation and set backscroll buffer lines to 100 (setting can be any number)
- On the ASCII set-up, enable send line ends with line feeds; echo typed characters locally. Append line feeds to incoming line ends and wrap lines that exceed terminal width.
- After all the above settings, on the call menu, first disconnect and then connect
- Configure the standard modem:
 - Disable V.42, an error-correcting protocol. This will enable only V.14 (sync to async and vice-versa) protocol
 - Type “at\n0”
 - Enable V.22bis in **1200bps** mode:
 - For Motorola Premier 33.6kbps modem: Type “at%l2%b2”
 - For Psion Dacom Gold Card Global 56k+fax modem: Type “at+ms=2,0,1200,1200,0,0,2400”
- Make the standard modem go off-hook and dial the telephone connected to the EVM
 - Type “atdt”, followed by the number of the telephone to which the EVM is connected
- Lift the telephone connected to the EVM and **immediately** run the test from CodeWarrior
- Both modems should now negotiate and connect for V.22bis 1200bps modulation scheme
- Full duplex communication is now possible between the two hyperterminals

9.4.1.5 Calling Modem Demo for 2400bps

Follow these steps to connect the V.22bis 2400bps soft-modem as the Calling modem and the standard (remote) modem as the Answering modem. This will test the asynchronous mode of operation.

- Load ...\\nos\\applications\\modem\\v22bis\\interopcall.mcp onto Metrowerks' CodeWarrior and build the target. The telephone to which the EVM is interfaced should be on-hook.
- Invoke the hyperterminal on the PC to which the EVM board is connected and configure it to:

- 115200bps
 - 8 bits
 - 1 stop bit
 - no parity
 - flow control = none
- On the settings panel, enable terminal keys and ANSI emulation and set backscroll buffer lines to 100 (setting can be any number)
- On the ASCII set-up, enable send line ends with line feeds; echo typed characters locally. Append line feeds to incoming line ends and wrap lines that exceed terminal width.
- After all the above settings, on the call menu, first disconnect and then connect
- On the remote side, connect the standard modem to the PC COM port. Invoke the hyperterminal on the PC. The hyperterminal is used to communicate (transmit AT Commands) to the standard modem. Configure the hyperterminal to:
 - **2400bps**
 - 8 bits
 - 1 stop bit
 - no parity
 - flow control = hardware
- On the settings panel, enable terminal keys and ANSI emulation and set backscroll buffer lines to 100 (setting can be any number)
- On the ASCII set-up, enable send line ends with line feeds; echo typed characters locally. Append line feeds to incoming line ends and wrap lines that exceed terminal width.
- After all the above settings, on the call menu, first disconnect and then connect
- Configure the standard modem:
 - Disable V.42, an error-correcting protocol, which will enable only V.14 (sync to async and vice-versa) protocol
 - Type “at\n0”
 - Enable V.22bis in **2400bps** mode:
 - For Motorola Premier 33.6kbps modem: Type “at%l3%b3”
 - For Psion Dacom Gold Card Global 56k+fax modem: Type “at+ms=2,0,2400,2400,0,0,2400”
- Lift the telephone connected to the EVM and dial the number of the line to which the standard modem is connected
- In the hyperterminal connected to the standard modem, type “ata” to answer the call and **immediately** run the test from CodeWarrior
- Both the modems should now negotiate and connect for V.22bis 2400bps modulation scheme
- Full duplex communication is now possible between the two hyperterminals

9.4.1.6 Calling Modem Demo for 1200bps

Follow these steps to connect the V.22bis 1200bps soft-modem as the Calling modem and the standard (remote) modem as the Answering modem. This will test the asynchronous mode of operation.

- Load ...\\nos\\applications\\modem\\v22bis\\interopcall.mcp onto Metrowerks' CodeWarrior and build the target. The telephone to which the EVM is interfaced should be on-hook.
- Invoke the hyperterminal on the PC to which the EVM board is connected and configure it to:
 - 115200bps
 - 8 bits
 - 1 stop bit
 - no parity
 - flow control = none
 - On the settings panel enable terminal keys and ANSI emulation and set backscroll buffer lines to 100 (setting can be any number)
 - On the ASCII set-up, enable send line ends with line feeds; echo typed characters locally; append line feeds to incoming line ends and wrap lines that exceed terminal width.
 - After all the above settings, on the call menu, first disconnect and then connect
- On the remote side, connect the standard modem to the PC COM port. Invoke the hyperterminal on the PC. The hyperterminal is used to communicate (transmit AT Commands) to the standard modem. Configure the hyperterminal to:
 - **1200bps**
 - 8 bits
 - 1 stop bit
 - no parity
 - flow control = hardware
 - On the settings panel, enable terminal keys and ANSI emulation and set backscroll buffer lines to 100 (setting can be any number)
 - On the ASCII set-up, enable send line ends with line feeds; echo typed characters locally; append line feeds to incoming line ends and wrap lines that exceed terminal width
 - After all the above settings, on the call menu, first disconnect and then connect
- Configure the standard modem:
 - Disable V.42, an error-correcting protocol, which will enable only V.14 (sync to async and vice-versa) protocol
 - Type “at\\n0”
 - Enable V.22bis in **1200bps** mode:
 - For Motorola Premier 33.6kbps modem: Type “at%l2%b2”
 - For Psion Dacom Gold Card Global 56k+fax modem: Type “at+ms=2,0,1200,1200,0,0,2400”
- Lift the telephone connected to the EVM and dial the number of the line to which the standard modem is connected
- In the hyperterminal connected to the standard modem, type “ata” to answer the call and **immediately** run the test from CodeWarrior
- Both modems should now negotiate and connect for V.22bis 1200bps modulation scheme
- Full duplex communication is now possible between the two hyperterminals

9.4.1.7 Special Considerations During Demo

- For detailed implementation of the demo applications, refer to the demo C files *interopans.c* and *interopcan.c*
- Both C application routines mentioned above use the UART drivers to interface the hyperterminal to the EVM board through the serial (COM) port; the driver call used to read characters from the hyperterminal is “*read_nonblocking*” and the driver call used to write the characters to the hyperterminal is “*write*”
- After every new setting on the hyperterminal, disconnect and connect on the call menu
- If the standard modem doesn't dial/detect the ring, type “ath” (hang) and then try to connect/answer
- Make sure that the Modem Mate switch option is connected to **voice** mode
- Since the codec is connected to the telephone through the Modem Mate, when the telephone connected to the EVM goes off-hook, both the mouthpiece and the earpiece are connected acoustically, increasing noise on the telephone line. To avoid this, cover both the mouthpiece and the earpiece with the cellophane tape, which will decrease the bit error rate.

9.4.2 V.42bis Demo

The demo encodes an input file using the V.42bis encoder algorithm and decodes the encoded output using the V.42bis decoder algorithm.

In the V.42bis demo, input data for the application is taken from the file *input.in* and encoded. The encoder output is written into the *enc_output.out* file. The encoded output is passed through the decoder and the decoder output is written into the file *output.out*. The file *output.out* should match the input file *input.in*.

9.4.2.1 Set-up for V.42bis Demo

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**.

File I/O:

This test is File I/O based.

- Connect the serial cable between the EVM and the COM1 port of the computer.

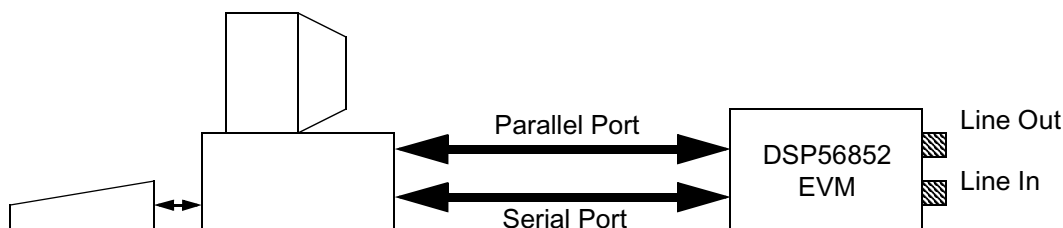


Figure 9-16. V.42bis Demo Set-up

9.4.2.2 Procedure for V.42bis Demo

- Invoke the PC File I/O application by executing the following file:
...\Embedded SDK\src\x86\win32\applications\fileio\fileio.exe
- Modify the settings to configure the File I/O application to run at 56000bps
- Using CodeWarrior, open: **...\nos\applications\modem\v42bis\v42bis_demo.mcp**
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select Run in the debugger to continue executing the demo
- The *V42bis_demo* will read input from the file *input.in* and will generate two output files, *enc_output.out* and *output.out*.
 - Encoder output is put into *enc_output.out*
 - Decoder output is put into *output.out*
- The files *output.out* and *input.in* should be identical
- To find the files *input.in*, *enc_output.out* and *output.out*, see:
...\nos\applications\modem\v42bis\testcases directory

9.5 Speech Demos

9.5.1 VRLite-1 Demo

The VRLite-1 library provides two demos:

- ***demo_vrlite1*** allows the user to train and recognize the following words in real time:
 - Answer
 - AreaCode
 - Book
 - Call
 - Cancel
 - ChargeCard
 - CreditCard
 - Dial Operator
 - Eight Hundred
 - Emergency
 - End
 - Friend
 - Home
 - Information
 - Office
 - Public Directory
 - Secretary
 - Security
 - Stop Demo

These words have been chosen to illustrate the capability of the VRLite-1 library in cell phone applications.

- *demo_dial_vrlite1* allows the user to train and recognize words in real time and dial the number using SDK's DTMF Generation library.

For example, suppose the user trains the word Dial Operator¹. Once the user recognizes this word, the demo dials the number stored in an array that corresponds to the Operator .

The VRLite-1 demo is designed to have three states, shown in [Figure 9-17](#). The description of each state follows.

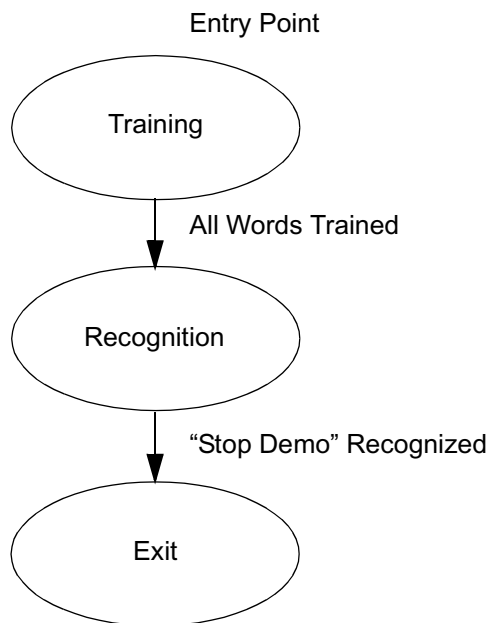


Figure 9-17. State Diagram of VRLite-1 Demo System

Training: This state is entered as soon as the demo system is started. After all words have been trained, control goes to the **Recognition State**.

Recognition: This state is entered only from the **Training State**. This state is not exited until the user says “Stop Demo”, when control goes to the **Exit State**.

Exit: This state is entered only from the **Recognition State**. If this state is entered, the VRLite-1 demo system halts.

The design of the VRLite-1 demo system is shown in the flowchart in [Figure 9-18](#).

1. The word DIALOPERATOR should not be trained as two separate words. It should be trained as a single word with a very slight pause between the words DIAL and OPERATOR

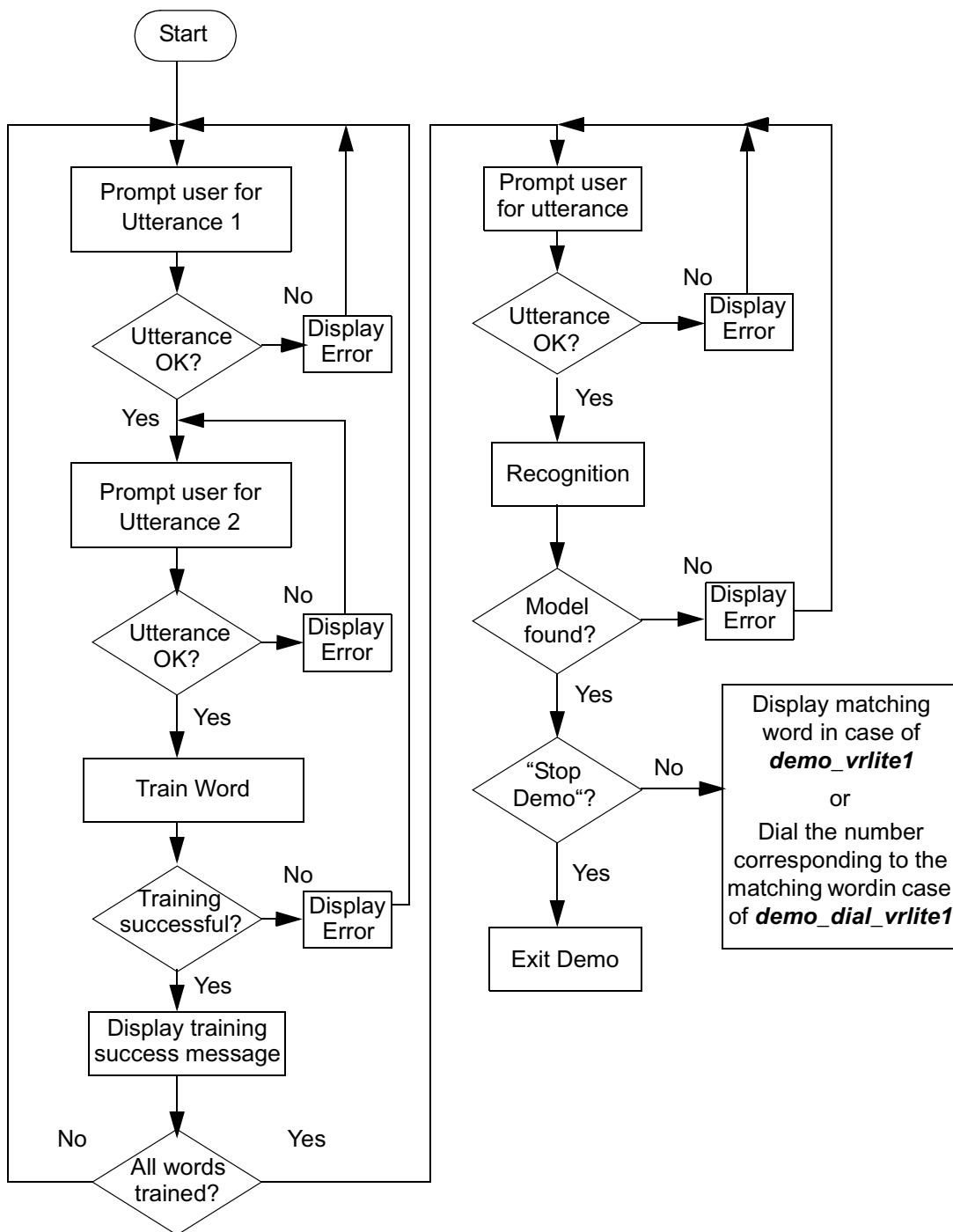


Figure 9-18. Flowchart for VRLite-1 Demo System

Demo details:

- All words defined by the application must be trained
 - The user is prompted on CodeWarrior's console to speak the first word in the list that has not yet been trained
 - If the user speaks a word properly, the user is asked to repeat the word, because training

- requires two utterances of the same word.
- If the user does not speak the word clearly, he will be prompted to repeat the word until it is clear
- After clearly speaking the two utterances, one of the following messages will be displayed on CodeWarrior's console:
 - **Training Unsuccessful** indicates that the training was not successful and prompts the user to train again
 - **Model Rejected** indicates that training is successful, but the model has been rejected since a similar model already exists in the vocabulary
 - **Model Accepted** indicates that training is successful and the model is accepted for storage in the vocabulary
- Upon successful completion of training of the current word, Step 1 will be repeated (starting with the next word) until all words have been successfully trained
- Recognition mode is now entered
 - The user is prompted on the CodeWarrior console to speak a word
 - If the user speaks a word and it is not clear, he will be prompted to speak again
 - If the user speaks a word which is already in the vocabulary, and the utterance is clear, the recognized word is displayed on CodeWarrior's console (*demo_vrlite1*), or the recognized word is used to dial the number corresponding to the recognized word (*demo_dial_vrlite1*)
 - If the user speaks a word which is NOT in the vocabulary and the utterance is clear, the message "Word Not in Vocabulary" is displayed on the CodeWarrior's console
 - Step 2 is repeated until user says "Stop Demo"
- Exit/Stop
 - Upon user saying "Stop Demo", the application is exited

9.5.1.1 Set-up for VRLite-1 Demo

EVM Jumper Settings:

Use the default settings shown in the **DSP56852 Evaluation Module User's Manual**. Toggle all three switches on the "Sample Rate" dipswitch (S4) to "Off", which will generate an 8000Hz sample rate.

The set-up for *demo_vrlite1* is shown in [Figure 9-19](#).

- Connect the parallel port of the PC to the parallel port of the EVM board
- Connect the microphone to Mic In of the PC
- Connect PC Audio Out to Speaker In of the Speaker System, and the Headphone jack of the Speaker System to Line In of the EVM board. The Speaker System is used here as an amplifier to generate a signal of sufficient strength (see Note). If desired, the user can replace the speaker system with a good amplifier system.

Note: The amplification level should be high enough to recognize the spoken word, and low enough to protect the EVM codec from damage. If a speaker system is being used (Harman-Kardon was used at the development site), then a good rule of thumb is to keep the volume level to approximately one-half the full scale.

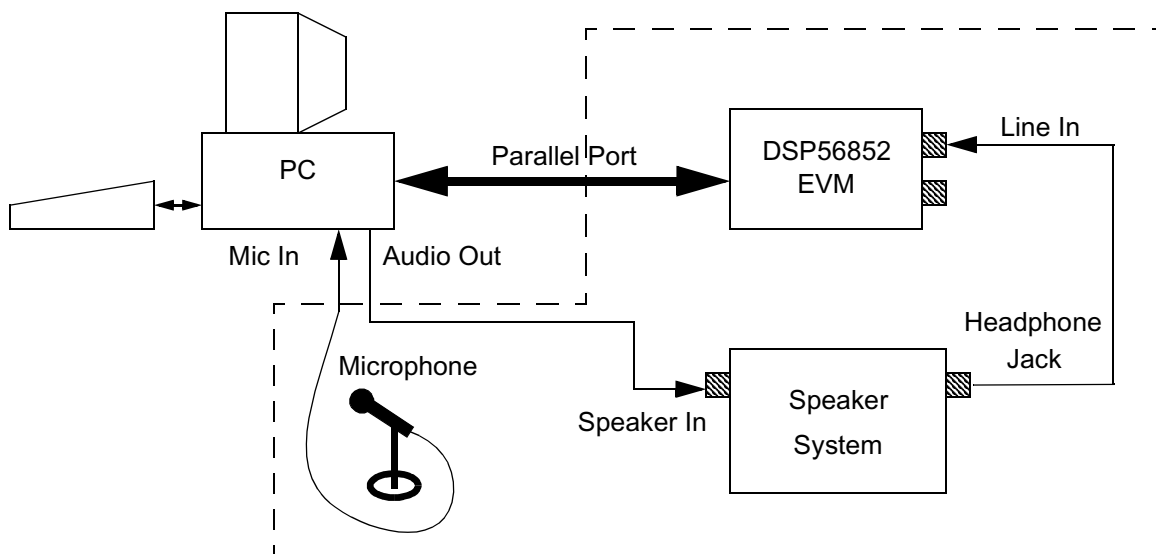
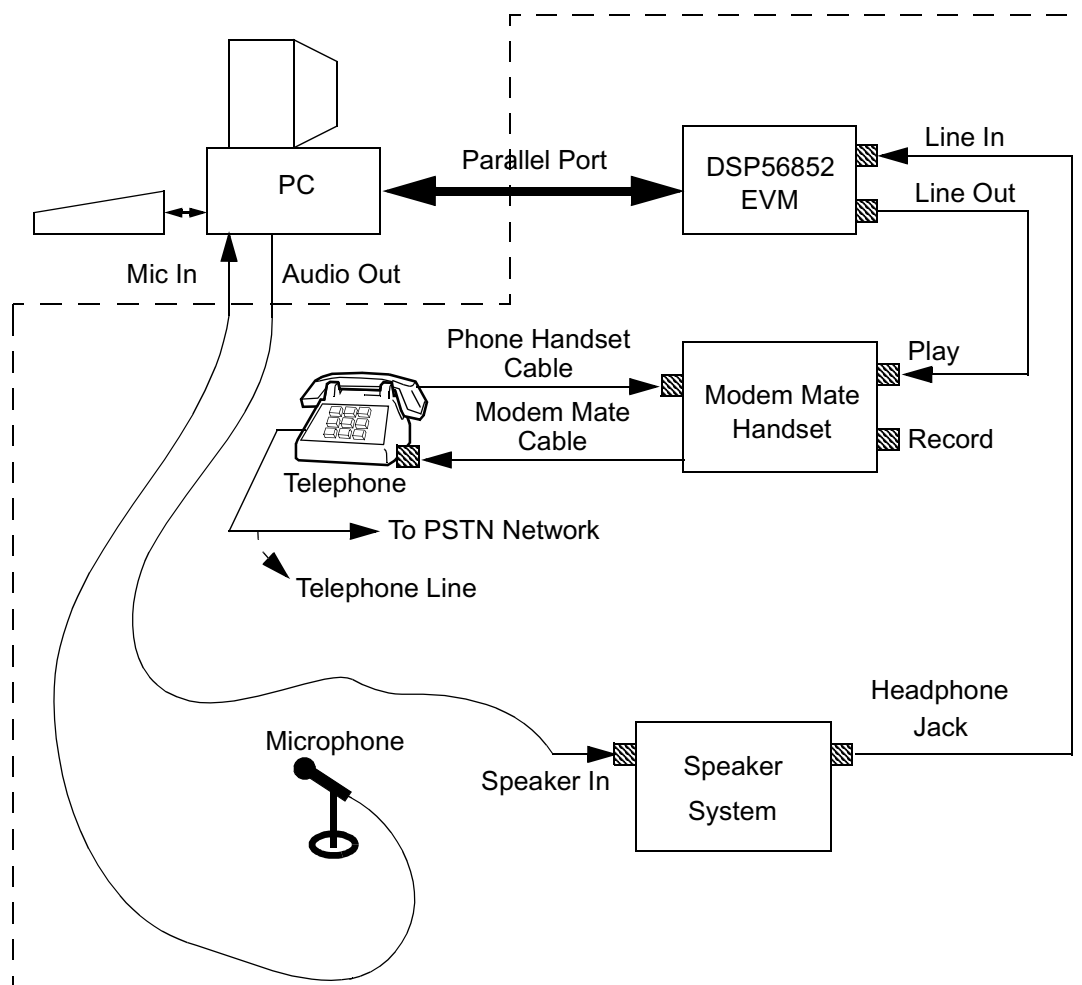


Figure 9-19. Demo Set-up for *demo_vrlite1*

The setup for *demo_dial_vrlite1* is shown in [Figure 9-20](#).

- Connect the parallel port of the PC to the parallel port of the EVM board
- Connect the microphone to Mic In of the PC
- Connect PC Audio Out to Speaker In of the Speaker System, and the Headphone jack of the Speaker System to Line In of the EVM board
 - The Speaker System is used here as an amplifier to generate a signal of sufficient strength (see Note below)
 - If desired, the user can replace the Speaker System with a good amplifier system
- The Line Out analog signal of the EVM board is connected to the Play terminal of the Modem Mate
- The handset of the telephone is connected to the Modem Mate's handset input
- The 4-wire cord coming out of Modem Mate is connected to the telephone's handset input

Note: The amplification level should be high enough to recognize the spoken word and low enough to protect the EVM codec from damage. If a speaker system is being used (Harman-Kardon was used at the development site), then a good rule of thumb is to keep the volume level to approximately one-half the full scale.


 Figure 9-20. Demo Set-up for *demo_dial_vrlite1*

9.5.1.2 Procedure for VRLite-1 Demo

A detailed procedure follows for each demo.

9.5.1.2.1 Procedure for *demo_vrlite1*

- Using CodeWarrior, open:
`...\nos\applications\speech\vrlite1\demo_vrlite1\demo_vrlite1.mcp`
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select Run in the debugger to continue executing the demo
- Follow the instructions displayed on the CodeWarrior console for training all of the words. The demo allows the user to train and recognize a total of 19 words. Train all of the words from the following list, in the order given:
 - Answer
 - Areacode
 - Book
 - Call
 - Cancel

- Chargecard
- Creditcard
- Dial Operator
- Eight Hundred
- Emergency
- End
- Friend
- Home
- Information
- Office
- Public Directory
- Secretary
- Security
- Stop Demo

Note: While uttering dual words like “Dial Operator”, there should not be a long pause between the words.

- Once training is complete, follow the instructions displayed on CodeWarrior’s console for recognizing the words
- It is not necessary to recognize words in the order trained; they can be recognized in a random order
- To exit the demo, say “Stop Demo”

9.5.1.2.2 Procedure for *demo_dial_vrlite1*

- Using CodeWarrior, open:
`..\nos\applications\speech\vrlite1\demo_dial_vrlite1\demo_dial_vrlite1.mcp`
- Open the file *dial_vrlite1.c*, located at:
`..\nos\applications\speech\vrlite1\demo_dial_vrlite1\c_sources\`
- Locate the arrays *RecWords*[][] and *TelNumbers*[][].
- Use the following instructions to set up these arrays:
 - Replace the existing strings in the array *RecWords*[][] with the strings of your choice.
 - For example, a string like DIAL OPERATOR can be stored in the *RecWords*[][] array. Before replacing, understand how it exists currently.
 - Replace the existing strings in the array *TelNumbers*[][] with the telephone numbers corresponding to the strings in the *RecWords*[][] array.
 - Example:

```
char RecWords[ ][20] = {"DIAL OPERATOR"};
```

```
char TelNumbers[ ][20] = {"2222"};
```

In this example, the number 2222 corresponds to Operator

Note: While uttering dual words like “Dial Operator”, there should not be a long pause between the words. Train words in the same order as the *RecWords*[][] array.

- Save the changes made to the file *dial_vrlite1.c* and close it
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select Run in the debugger to continue executing the demo
- Follow the instructions displayed on CodeWarrior’s console for training the words specified in the array *RecWords*[][]

- Once the words are trained, the demo system is ready for recognition
- Go off-hook on the telephone connected in the set-up
- Follow the instructions displayed on CodeWarrior's console for recognizing the words
 - It is not necessary to recognize words in the order trained; it can be recognized in a random order
 - Once the uttered word is recognized, it is displayed on CodeWarrior's console and the VRLite-1 demo dials the corresponding telephone number
 - Be sure there is a the dial-tone before dialing
- To exit the demo, say "STOP DEMO"

9.5.1.3 Demo Limitations

- Be sure that external noise is minimal, especially during training
- The DSP56852 has no flash memory, so the trained models are stored in external RAM and will be erased once the demo is stopped. This means the user must retrain the demo system to start recognition.
- The demo system must be trained in the order given in step 3, [Section 9.5.1.2.1](#), because to display the recognized word, a character array has been used which stores the string corresponding to the voice tag in that order. This is enforced by the demo.
- The demo allows the user to train a maximum of 20 words; however, this number can be increased, provided memory is not a constraint



Index

Numerics

3DES [xxxi](#)

A

ADPCM [xxxi](#)

AEC [xxxi](#)

autoCorr Performance [7-3](#)

B

BSP [xxxi](#)

buildall.mcp [1-3](#)

C

Caller ID Detection Test [8-8](#)

CAS [xxxi](#)

CBC [xxxi](#)

cbitrev Performance [7-4](#)

CFB [xxxi](#)

CFFT [xxxi](#)

CID [xxxi](#)

cid1.lib [8-35](#)

cid1test.mcp [8-35](#)

CODEC [xxxi](#)

Codec [6-10](#)

CodeWarrior

Registration [1-2](#)

Conventions [xxx](#)

COP [xxxi](#)

CPE [xxxi](#)

CPT [xxxi](#)

CPT Detection Test [8-11](#)

D

DCE [xxxi](#)

Decoder [8-16](#)

DES [xxxi](#)

DES Loopback Test [8-4](#)

Directory Structure [2-1](#)

domain specific libraries [7-1](#)

DSP [xxxi](#)

DSP56800 Family Manual [xxxi](#)

DSP56852

Install Hardware [1-2](#)

DSP56852 User's Manual [xxxi](#)

DTE [xxxi](#)

DTMF [xxxi](#)

Generation Test [8-13](#)

DTMF Detection Test [8-12](#)

E

Echo Cancellation [9-14](#), [9-15](#)

Embedded SDK Programmer's Guide [xxxi](#)

Encoder [8-16](#)

ERLE [xxxi](#)

EVM [xxxi](#)

F

FFT [xxxi](#)

FILE [6-36](#)

FIR [xxxi](#)

G

G.711

Test Set-up [8-16](#)

G.726

Test Set-up [8-17](#)

H

Hold Release logic [9-14](#), [9-15](#)

I

I/O [xxxi](#)

IDE [xxxi](#)

IIR [xxxi](#)

Install CodeWarrior [1-1](#)

Install SDK [1-2](#)

ITCN [xxxi](#)

L

license key [1-2](#)

LSB [xxxi](#)

M

MAC [xxxi](#)

Memory and MIPS Requirements [7-38](#), [7-39](#), [7-41](#)

memory and MIPS requirements [7-40](#)

Metrowerks CodeWarrior [1-1](#)

MIPS [xxxi](#)

MSB [xxxi](#)

N

NLMS [xxxi](#)

O

OMR [xxxi](#)

OnCE [xxxi](#)

P

PC [xxxi](#)
PC Master appconfig.h Settings [6-89](#)
PCM [xxxi](#), [7-32](#)
PLL [xxxi](#)
PSTN [xxxi](#)

Q

Quick Start [1-1](#)

R

Register CodeWarrior [1-2](#)
RFFT [xxxi](#)
RinBuffer [8-14](#), [8-15](#)
RSA [xxxi](#)

S

SAS [xxxi](#)
SCI [xxxi](#)
SDK [xxxi](#)
SIM [xxxi](#)
SinBuffer [8-14](#), [8-15](#)
SP [xxxi](#)
SPCS [xxxi](#)
SPI [xxxi](#)
SR [xxxi](#)
SRC [xxxi](#)
SSI [xxxi](#)

T

Tone Disabler [9-14](#), [9-15](#)
Trigonometric Math Function Performance [7-2](#)
Type1CID [8-35](#)

V

V.21
 Test Set-up [8-20](#)
V.21 Loopback Test [8-19](#)
V.22bis
 Loopback Test [8-20](#)
 Loopback Test Set-up [8-21](#)
V.22bis Library [7-11](#)
V.8bis
 Test Setup [8-22](#)
V.8bis Test [8-22](#)
V22bis
 Loopback Test [8-19](#), [8-20](#)
VAD [xxxi](#)
 Test Set-up [8-7](#), [8-18](#), [8-19](#)
VAD Test [8-7](#), [8-18](#)
VR [xxxi](#)
VRLite-1
 Test Setup [8-25](#)



Freescale Semiconductor, Inc.

ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

Freescale Semiconductor, Inc.

ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

**For More Information On This Product,
Go to: www.freescale.com**

Because of an order from the United States International Trade Commission, BGA-packaged product lines and part numbers indicated here currently are not available from Freescale for import or sale in the United States prior to September 2010: DSP56852VFE



Freescale Semiconductor, Inc.

ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

Freescale Semiconductor, Inc.

ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

Motorola reserves the right to make changes without further notice to any products herein. Motorola makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Motorola 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 Motorola 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. Motorola does not convey any license under its patent rights nor the rights of others. Motorola 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 Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola 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 Motorola was negligent regarding the design or manufacture of the part. Motorola and the Stylized M Logo are registered trademarks of Motorola, Inc. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

MOTOROLA and the Stylized M Logo are registered in the US Patent & Trademark Office. All other product or service names are the property of their respective owners. © Motorola, Inc. 2002.

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/>



MOTOROLA

**For More Information On This Product,
Go to: www.freescale.com**

SDK131/D

Because of an order from the United States International Trade Commission, BGA-packaged product lines and part numbers indicated here currently are not available from Freescale for import or sale in the United States prior to September 2010: DSP56852VFE