



Embedded SDK (Software Development Kit)

Targeting Motorola DSP5685x Platform

SDK137/D
Rev. 3, 11/09/2002



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	xxxiii
Organization	xxxiii
Suggested Reading	xxxiii
Conventions	xxxiv
References.	xxxvi

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 DSP56858EVM Hardware	1-2
1.2.3 Install SDK	1-2
1.2.4 Build Released Software.	1-3

Chapter 2

Directory Structure

2.1 DSP56858 SDK Directory Structure	2-1

Chapter 3

Target Configuration for DSP5685x

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 CODEC_DMA	3-4
3.1.10 COP.	3-4
3.1.11 CPT.	3-5
3.1.12 CTG	3-5
3.1.13 CYCLE_COUNT	3-5
3.1.14 DES.	3-6
3.1.15 DMA	3-6



3.1.16	DSPFUNC	3-7
3.1.17	DTMF_DET	3-7
3.1.18	DTMF_GEN	3-8
3.1.19	ESSI	3-8
3.1.20	ESSI_DMA	3-8
3.1.21	FILEIO	3-9
3.1.22	G.165	3-9
3.1.23	G.168	3-9
3.1.24	G.711	3-10
3.1.25	G.726	3-10
3.1.26	GPIO	3-10
3.1.27	HI	3-11
3.1.28	IO	3-11
3.1.29	ITCN	3-11
3.1.30	Keypad	3-12
3.1.31	LCD	3-12
3.1.32	LED	3-12
3.1.33	MEMORY	3-13
3.1.34	NS	3-13
3.1.35	PC Master Software	3-13
3.1.36	PCS	3-14
3.1.37	PLL	3-14
3.1.38	QUAD_TIMER	3-14
3.1.39	RSA	3-15
3.1.40	SCI	3-15
3.1.41	SERIAL_DATA_FLASH	3-15
3.1.42	SIM	3-16
3.1.43	SPI	3-16
3.1.44	STACK_CHECK	3-16
3.1.45	TDC1	3-17
3.1.46	TOD	3-17
3.1.47	TIMER	3-17
3.1.48	UCOS	3-18
3.1.49	VAD	3-18
3.1.50	V.8bis	3-18
3.1.51	V.21	3-19
3.1.52	V.22bis	3-19
3.1.53	V.42bis	3-19
3.1.54	VR Lite-1	3-20
3.2	Memory Configuration	3-21
3.2.1	External Memory Operation	3-21
3.2.1.1	Linker Command File	3-21
3.2.1.2	Linking: Set Up Memory Partitions	3-23
3.2.1.3	Programming Target Memory	3-24
3.2.2	Internal Memory Operation	3-25
3.2.2.1	Linker Command File	3-26
3.2.2.2	Linking: Set Up Memory Partitions	3-27
3.2.2.3	Programming Target Memory	3-28

3.3	DSP5685x Boot Sequence Configuration.	3-30
3.3.1	First Stage Bootloader	3-31
3.3.1.1	Boot from Byte-Wide External Memory (Mode 0)	3-31
3.3.1.2	Boot from SPI (Mode 1)	3-32
3.3.1.3	Boot from Normal Expanded Mode (Mode 2)	3-34
3.3.1.4	Boot from Development Expanded Mode (Mode 3)	3-34
3.3.1.5	Boot from Host Interface using Single Strobe (Mode 4)	3-34
3.3.1.6	Boot from Host Interface using Dual Strobes (Mode 5)	3-35
3.3.1.7	Boot from SCI (Mode 6)	3-35
3.3.1.8	Unused Boot Mode (Mode 7)	3-37
3.3.2	Second Stage Bootloader	3-37
3.3.2.1	Second Stage SPI Bootloader	3-37
3.3.2.1.1	Second Stage SPI Bootloader Demo	3-39
3.3.3	User Application	3-41
3.3.4	DSP56858EVM Serial Data Flash Programmer.	3-41
3.3.4.1	Specification	3-41
3.4	Clock (PLL) Configuration.	3-43

Chapter 4 Interrupt Processing for the DSP5685x

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	Host Interface Driver	4-4
4.4.5	LED Driver	4-4
4.4.6	Quad Timer Driver	4-4
4.4.7	SCI Driver	4-4
4.4.8	Serial Data Flash Driver	4-5
4.4.9	SPI Driver	4-5
4.4.10	SSI Driver	4-5
4.4.11	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	DMA Driver	5-14
5.2.1	Introduction.	5-14
5.2.2	Initialization	5-14
5.2.3	API Definition.	5-19
5.2.4	DMA Device-Independent API Specification	5-20
5.2.4.1	open.	5-21
5.2.4.2	write	5-23
5.2.4.3	read	5-25
5.2.4.4	close	5-27
5.2.4.5	ioctl	5-29
5.2.5	DMA Device-Dependent API Specification.	5-33
5.2.5.1	dmaOpen.	5-34
5.2.5.2	dmaWrite	5-36
5.2.5.3	dmaRead	5-38
5.2.5.4	dmaClose	5-40
5.2.5.5	dmaIoctl	5-42
5.2.6	DMA Driver Tests	5-44
5.2.6.1	DMA Memory-to-Memory Test.	5-44
5.2.6.1.1	Set-up for DMA Memory-to-Memory Test	5-44
5.2.6.1.2	Procedure for DMA Memory-to-Memory Test	5-44
5.2.7	DMA Driver Demos	5-44
5.2.7.1	DMA Peripheral-to-Memory and Memory-to-Peripheral Tests	5-44
5.3	ESSI Driver.	5-45
5.3.1	Introduction.	5-45
5.3.2	Initialization	5-45
5.3.3	API Definition.	5-62
5.3.4	ESSI Device-Independent API Specification	5-64
5.3.4.1	open.	5-65
5.3.4.2	write	5-66
5.3.4.3	read	5-67
5.3.4.4	close	5-68
5.3.4.5	ioctl	5-69
5.3.5	ESSI Device-Dependent API Specification	5-71
5.3.5.1	essiOpen	5-72
5.3.5.2	essiWrite	5-73
5.3.5.3	essiRead	5-74
5.3.5.4	essiClose	5-75
5.3.5.5	essiIoctl	5-76



5.3.6	ESSI Driver Tests and Demos	5-78
5.4	ESSI DMA Driver	5-79
5.4.1	Introduction.	5-79
5.4.2	Initialization	5-79
5.4.3	API Definition.	5-89
5.4.4	ESSI DMA Device-Independent API Specification.	5-90
5.4.4.1	open.	5-91
5.4.4.2	write	5-92
5.4.4.3	read	5-93
5.4.4.4	close	5-94
5.4.4.5	ioctl	5-95
5.4.5	ESSI DMA Device-Dependent API Specification	5-98
5.4.5.1	essidmaOpen.	5-99
5.4.5.2	essidmaWrite	5-100
5.4.5.3	essidmaRead	5-101
5.4.5.4	essidmaClose	5-102
5.4.5.5	essidmaIoctl	5-103
5.4.6	ESSI DMA Driver Tests and Demos	5-106
5.5	GPIO Driver	5-106
5.5.1	Introduction.	5-106
5.5.2	Initialization	5-106
5.5.3	API Definition.	5-106
5.5.4	GPIO Device-Independent API Specification	5-108
5.5.4.1	open.	5-109
5.5.4.2	close	5-110
5.5.4.3	ioctl	5-111
5.5.5	GPIO Device-Dependent API Specification.	5-113
5.5.5.1	gpioOpen.	5-114
5.5.5.2	gpioClose	5-115
5.5.5.3	gpioIoctl	5-116
5.5.6	GPIO Driver Application	5-118
5.6	Host Interface Driver	5-119
5.6.1	Introduction.	5-119
5.6.2	Initialization	5-119
5.6.3	API Definition.	5-121
5.6.4	HI Device-Independent API Specification	5-122
5.6.4.1	open.	5-123
5.6.4.2	write	5-125
5.6.4.3	read	5-126
5.6.4.4	close	5-127
5.6.4.5	ioctl	5-128
5.6.5	HI Device-Dependent Specification	5-131
5.6.5.1	hiOpen.	5-132
5.6.5.2	hiWrite	5-134
5.6.5.3	hiRead	5-135
5.6.5.4	hiClose	5-136
5.6.5.5	hiIoctl	5-137

5.6.6	HI Driver Tests	5-140
5.6.7	HI Driver Demos	5-140
5.6.7.1	Host Interface (HI) Demo	5-140
5.6.7.1.1	Set-up for Host Interface Demo	5-140
5.6.7.1.2	Host Interface Demo	5-141
5.6.7.1.3	Procedure for Host Interface Demo Using One PC	5-141
5.6.7.1.4	Procedure for Host Interface Demo if Using Two PCs	5-141
5.6.7.1.5	Procedure for Host Interface Demo	5-141
5.7	Programmable Chip Select (PCS) Driver	5-143
5.7.1	Initialization	5-143
5.7.2	API Definition	5-145
5.7.2.1	pcsInit()	5-146
5.7.2.2	pcsInitEMI()	5-150
5.8	Quad Timer Driver	5-152
5.8.1	Introduction	5-152
5.8.2	Initialization	5-152
5.8.3	API Definition	5-153
5.8.4	Quad Timer Device-Independent API Specification	5-159
5.8.4.1	open	5-160
5.8.4.2	close	5-161
5.8.4.3	ioctl	5-162
5.8.5	Quad Timer Device-Dependent API Specification	5-166
5.8.5.1	qtOpen	5-167
5.8.5.2	qtClose	5-168
5.8.5.3	qtIoctl	5-169
5.8.6	Quad Timer Driver Tests	5-170
5.8.7	Quad Timer Driver Demos	5-170
5.8.7.1	Quad Timer Flashing LEDs Demo	5-170
5.8.7.1.1	Set-up for Quad Timer Flashing LEDs Demo	5-170
5.8.7.1.2	Procedure for Quad Timer Flashing LEDs Demo	5-170
5.9	SCI Driver	5-170
5.9.1	Introduction	5-170
5.9.2	Initialization	5-170
5.9.3	API Definition	5-174
5.9.4	SCI Device-Independent API Specification	5-176
5.9.4.1	open	5-177
5.9.4.2	write	5-178
5.9.4.3	read	5-179
5.9.4.4	close	5-180
5.9.4.5	ioctl	5-181
5.9.5	SCI Device-Dependent API Specification	5-184
5.9.5.1	sciOpen	5-185
5.9.5.2	sciWrite	5-186
5.9.5.3	sciRead	5-187
5.9.5.4	sciClose	5-188
5.9.5.5	sciIoctl	5-189
5.9.6	SCI Driver Tests	5-192



5.9.6.1	SCI Loopback Test	5-192
5.9.6.1.1	Set-up for SCI Loopback Test	5-192
5.9.6.1.2	Procedure for SCI Loopback Test	5-192
5.9.7	SCI Driver Demos	5-192
5.9.7.1	SCI Serial Port Demo	5-192
5.9.7.1.1	Set-up for SCI Serial Port Demo	5-192
5.9.7.1.2	Procedure for SCI Serial Port Demo	5-192
5.9.7.2	SCI Echo Demo	5-193
5.9.7.2.1	Set-up for SCI Echo Demo	5-193
5.9.7.2.2	Procedure for SCI Echo Demo	5-193
5.10	SPI Driver	5-193
5.10.1	Introduction	5-193
5.10.2	Initialization	5-193
5.10.3	API Definition	5-194
5.10.4	SPI Device-Independent API Specification	5-196
5.10.4.1	open	5-197
5.10.4.2	write	5-198
5.10.4.3	read	5-199
5.10.4.4	close	5-200
5.10.4.5	ioctl	5-201
5.10.5	SPI Device-Dependent API Specification	5-203
5.10.5.1	spiOpen	5-204
5.10.5.2	spiBlockedWrite, spiNonBlockedWrite	5-205
5.10.5.3	spiBlockedRead, spiNonBlockedRead	5-206
5.10.5.4	spiBlockedClose, spiNonBlockedClose	5-207
5.10.5.5	spiIoctl	5-208
5.10.6	SPI Driver Applications	5-210
5.10.7	SPI Driver Tests	5-210
5.10.7.1	SPI Test	5-210
5.10.7.1.1	Set-up for SPI Test	5-210
5.10.7.1.2	Procedure for SPI Test	5-210
5.11	Timer Driver	5-210
5.11.1	Introduction	5-210
5.11.2	Timer Driver Tests	5-210
5.11.3	Timer Driver Demos	5-211
5.11.3.1	Timer Driver Flashing LEDs Demo	5-211
5.11.3.1.1	Set-up for Timer Driver Flashing LEDs Demo	5-211
5.11.3.1.2	Procedure for Timer Driver Flashing LEDs Demo	5-211
5.12	Time of Day (TOD) Driver	5-211
5.12.1	Introduction	5-211
5.12.2	Initialization	5-211
5.12.3	TOD Device-Independent API Specification	5-212
5.12.4	TOD Device-Dependent API Specification	5-212
5.12.5	TOD Device-Independent API Specification	5-214
5.12.6	TOD Device-Dependent API Specification	5-215
5.12.6.1	todOpen	5-216
5.12.6.2	todEnableCallbacks	5-217
5.12.6.3	todSetAlarm	5-218

5.12.6.4	todGetTime	5-220
5.12.6.5	todClose	5-221
5.12.6.6	todIoctl	5-222
5.12.7	TOD Driver Application	5-226

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	6-9
6.2.1	Introduction	6-9
6.2.2	Initialization	6-10
6.2.3	API Definition	6-16
6.2.4	Codec Device-Independent 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	6-27
6.2.5.1	codecOpen	6-28
6.2.5.2	codecWrite	6-29
6.2.5.3	codecRead	6-30
6.2.5.4	codecClose	6-31
6.2.5.5	codecIoctl	6-32
6.2.6	Codec Driver Tests	6-37
6.2.7	Codec Driver Applications	6-37
6.2.7.1	Codec Loopback Application	6-37
6.2.7.1.1	Set-up for Codec Loopback Application	6-37
6.2.7.1.2	Procedure for Codec Loopback Application	6-37
6.3	Codec DMA Driver	6-38
6.3.1	Introduction	6-38
6.3.2	Initialization	6-39



6.3.3	API Definition.	6-45
6.3.4	Codec DMA Device-Independent API Specification.	6-46
6.3.4.1	open.	6-47
6.3.4.2	write	6-48
6.3.4.3	read	6-49
6.3.4.4	close	6-50
6.3.4.5	ioctl	6-51
6.3.5	Codec DMA Device-Dependent API Specification.	6-56
6.3.5.1	codecdmaOpen	6-57
6.3.5.2	codecdmaWrite	6-58
6.3.5.3	codecdmaRead	6-59
6.3.5.4	codecdmaClose	6-60
6.3.5.5	codecdmaIoctl	6-61
6.3.6	Codec DMA Driver Tests.	6-64
6.3.7	Codec DMA Driver Applications.	6-64
6.3.7.1	Codec DMA Loopback Application.	6-64
6.3.7.1.1	Set-up for Codec DMA Loopback Application	6-64
6.3.7.1.2	Procedure for Codec DMA Loopback Application	6-64
6.4	FILE I/O Driver	6-66
6.4.1	Introduction.	6-66
6.4.2	Initialization	6-67
6.4.3	API Definition.	6-68
6.4.4	API Specifications	6-69
6.4.4.1	open.	6-70
6.4.4.2	write	6-71
6.4.4.3	read	6-72
6.4.4.4	close	6-73
6.4.4.5	ioctl	6-74
6.4.5	FILE I/O Driver Application	6-74
6.5	Keypad Driver.	6-75
6.5.1	Introduction.	6-75
6.5.2	Initialization	6-75
6.5.3	API Definition.	6-76
6.5.4	Keypad Device-Independent API Specification.	6-79
6.5.4.1	open.	6-80
6.5.4.2	ioctl	6-81
6.5.4.3	close	6-82
6.5.5	Keypad Device-Dependent API Specification	6-83
6.5.5.1	keypadOpen	6-84
6.5.5.2	keypadIoctl	6-85
6.5.5.3	keypadClose	6-88
6.5.6	Demo Applications	6-90
6.5.7	Test Applications	6-90
6.5.7.1	Hardware Set-up	6-91
6.5.7.2	Procedure for TDC1 Keypad Test	6-91
6.6	LCD Driver.	6-91
6.6.1	Introduction.	6-91
6.6.2	Initialization	6-92

6.6.3	API Definition.	6-93
6.6.4	LCD Device-Independent API Specification	6-94
6.6.4.1	open.	6-95
6.6.4.2	write	6-96
6.6.4.3	ioctl	6-97
6.6.4.4	close	6-98
6.6.5	LCD Device-Dependent API Specification	6-99
6.6.5.1	lcdOpen.	6-100
6.6.5.2	lcdWrite	6-101
6.6.5.3	lcdIoctl	6-103
6.6.5.4	lcdClose	6-105
6.6.6	Demo Applications	6-106
6.6.7	Test Applications	6-106
6.6.7.1	Hardware Set-up	6-106
6.6.7.2	Procedure for LCD Test	6-106
6.7	LED Driver	6-107
6.7.1	Introduction.	6-107
6.7.2	Initialization	6-107
6.7.3	API Definition.	6-107
6.7.4	LED Device-Independent I/O API Specifications	6-109
6.7.4.1	open.	6-110
6.7.4.2	close	6-111
6.7.4.3	ioctl	6-112
6.7.5	Low-Level Device Drivers I/O API Specifications	6-114
6.7.5.1	ledOpen.	6-115
6.7.5.2	ledClose	6-116
6.7.5.3	ledIoctl	6-117
6.7.6	LED Driver Tests	6-118
6.7.6.1	LED Driver Test	6-118
6.7.6.1.1	Set-up for LED Driver Test	6-118
6.7.6.1.2	Procedure for LED Driver Test	6-118
6.7.7	LED Driver Applications	6-118
6.8	PC Master Software SCI Communication Driver.	6-119
6.8.1	Introduction.	6-119
6.8.2	Initialization	6-119
6.8.3	API Definitions.	6-119
6.8.4	PC Master Software Low-Level I/O API Specification	6-123
6.8.4.1	pcmasterdrvInit - Initialize PC Master Software Communication	6-124
6.8.4.2	pcmasterdrvGetAppCmdSts - Get Application Command Status	6-125
6.8.4.3	pcmasterWriteAppCmdSts - Write Application Command Status.	6-126
6.8.4.4	pcmasterdrvIsr - Main Communication Function.	6-127
6.8.4.5	pcmasterdrvRecorder - Sample Recorded Variables	6-128
6.8.5	PC Master Software Tests.	6-129
6.8.6	PC Master Software Applications.	6-129
6.8.6.1	PC Master Software Demo	6-129
6.8.6.1.1	Set-up for PC Master Software Demo	6-129
6.8.6.1.2	Procedure for PC Master Software Demo	6-129



6.9	Serial Data Flash Driver	6-130
6.9.1	Introduction	6-130
6.9.2	Initialization	6-131
6.9.3	API Definition	6-131
6.9.4	Serial Data Flash Device-Independent I/O API Specifications	6-132
6.9.4.1	open	6-133
6.9.4.2	write	6-134
6.9.4.3	read	6-135
6.9.4.4	close	6-136
6.9.4.5	ioctl	6-137
6.9.5	Low-Level Device Drivers I/O API Specifications	6-140
6.9.6	Serial Data Flash Driver Tests	6-140
6.9.6.1	Serial Data Flash Driver Test	6-140
6.9.6.1.1	Set-up for Serial Data Flash Driver Test	6-140
6.9.6.1.2	Procedure for Serial Data Flash Driver Test	6-140
6.9.7	Serial Data Flash Driver Demo	6-140
6.10	TDC1 Driver	6-141
6.10.1	Introduction	6-141
6.10.2	Initialization	6-141
6.10.3	API Definition	6-150
6.10.4	Device-Independent API Specification	6-151
6.10.4.1	open	6-152
6.10.4.2	write	6-153
6.10.4.3	read	6-154
6.10.4.4	close	6-155
6.10.4.5	ioctl	6-156
6.10.5	Device-Dependent TDC1 Driver API	6-162
6.10.5.1	tdc1Open	6-163
6.10.5.2	tdc1Write	6-164
6.10.5.3	tdc1Read	6-165
6.10.5.4	tdc1Close	6-166
6.10.5.5	tdc1Ioctl	6-167
6.10.6	TDC1 Driver Tests	6-177
6.10.7	TDC1 Driver Applications	6-177
6.10.7.1	TDC1 Loopback Application	6-177
6.10.7.1.1	Set-up for TDC1 Loopback Application	6-177
6.10.7.1.2	Procedure for TDC1 Loopback Application	6-177
6.11	IDC Driver	6-178
6.11.1	Introduction	6-178
6.11.2	Initialization	6-178
6.11.3	Device-Independent API Definition	6-181
6.11.4	API Specifications	6-183
6.11.4.1	open	6-184
6.11.4.2	write	6-185
6.11.4.3	read	6-186
6.11.4.4	close	6-187
6.11.4.5	ioctl	6-188

6.11.5	Device-Dependent IDC Driver API	6-191
6.11.5.1	idcOpen	6-192
6.11.5.2	idcWrite	6-193
6.11.5.3	idcRead	6-194
6.11.5.4	idcClose	6-195
6.11.5.5	idcIoctl	6-196
6.11.6	IDC Driver Tests	6-199

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	7-5
7.1.2.2.4	ciff	7-6
7.1.2.2.5	corr	7-6
7.1.2.2.6	fir	7-7
7.1.2.2.7	iir	7-7
7.1.2.2.8	rfft	7-8
7.1.2.2.9	riff	7-9
7.2	Modem Libraries	7-9
7.2.1	V.8bis Library	7-9
7.2.1.1	Performance	7-10
7.2.2	V.21 Library	7-11
7.2.2.1	Performance	7-12
7.2.3	V.22bis Library	7-12
7.2.3.1	Performance	7-12
7.2.4	V.42bis Library	7-13
7.2.4.1	Performance	7-14
7.3	Security Libraries	7-14
7.3.1	DES Library	7-15
7.3.1.1	Performance	7-15
7.3.2	Triple DES Library	7-16
7.3.2.1	Performance	7-17
7.3.3	RSA Library	7-18
7.3.3.1	Performance	7-19
7.4	Telephony Libraries	7-20
7.4.1	Acoustic Echo Canceller Library	7-20
7.4.1.1	Performance	7-20
7.4.2	Automatic Gain Control (AGC) Library	7-21
7.4.3	CAS Detect Library	7-22
7.4.3.1	Performance	7-23



7.4.4	Call Progress Tones Detect Library	7-24
7.4.4.1	Performance	7-25
7.4.5	Caller ID Library	7-25
7.4.5.1	Performance	7-25
7.4.6	Common Tone Generation (CTG) Library	7-26
7.4.6.1	Performance	7-26
7.4.7	DTMF Detection Library	7-27
7.4.7.1	Performance	7-28
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	7-30
7.4.10	G.168 Library	7-30
7.4.10.1	Performance	7-31
7.4.11	G.711 Library	7-31
7.4.11.1	Performance	7-32
7.4.12	G.723.1A Speech Codec Library	7-33
7.4.12.1	Performance	7-33
7.4.13	G.726 Library	7-33
7.4.13.1	Performance	7-33
7.4.14	G.729AB Vocoder Library	7-34
7.4.14.1	Performance	7-35
7.4.15	MFCR2 Detection Library	7-35
7.4.15.1	Performance	7-35
7.4.16	Noise Suppression (NS) Library	7-36
7.4.17	Voice Activity Detector Library	7-37
7.4.17.1	Performance	7-37
7.5	Speech Libraries	7-38
7.5.1	VR Lite-1 Library	7-38
7.5.1.1	Performance	7-38
7.6	Feature Phone Libraries	7-40
7.6.1	Generic Echo Canceller Library	7-40
7.6.1.1	Performance	7-40
7.6.2	Type 1 and 2 Telephony Parser Library	7-41
7.6.2.1	Performance	7-41
7.6.3	Full Duplex Speakerphone Library	7-41
7.6.3.1	Performance	7-41
7.6.4	Type 1 Telephony Features Library	7-42
7.6.4.1	Performance	7-42
7.6.5	Type 1 and 2 Telephony Features Library	7-42
7.6.6	Features and Performance	7-43

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	8-3
8.3.1	Math Tests	8-3
8.3.1.1	Set-up for Math Test	8-3
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	8-4
8.3.3	RFFT Test	8-4
8.3.3.1	Set-up for RFFT Test	8-4
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	8-5
8.4.1	DES Loopback Test	8-5
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	8-6
8.4.1.6	RSA Loopback Test	8-6
8.4.1.7	Set-up for RSA Loopback Test	8-6
8.4.1.8	Procedure for RSA Loopback Test	8-6
8.5	Telephony Libraries	8-7
8.5.1	AEC Test	8-7
8.5.1.1	Set-up for AEC Test	8-7
8.5.1.2	Procedure for AEC Test	8-7
8.5.2	Automatic Gain Control (AGC) Test	8-8
8.5.2.1	Set-up for AGC Test	8-8
8.5.2.2	Procedure for AGC Test	8-8
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	8-11
8.5.4	CAS Detection Test	8-11
8.5.4.1	Set-up for CAS Detection Test	8-11
8.5.4.2	Procedure for CAS Detection Test	8-11



8.5.5	CPT Detection Test.	8-12
8.5.5.1	Set-up for CPT Detection Test	8-12
8.5.5.2	Procedure for CPT Detection Test	8-12
8.5.6	Common Tone Generation (CTG) Test	8-12
8.5.6.1	Set-up for Common Tone Generation Test.	8-12
8.5.6.2	Procedure for Common Tone Generation Test.	8-13
8.5.7	DTMF Detection Test.	8-13
8.5.7.1	Set-up for DTMF Detection Test	8-13
8.5.7.2	Procedure for DTMF Detection Test	8-14
8.5.8	DTMF Generation Test.	8-14
8.5.8.1	Set-up for DTMF Generation Test	8-14
8.5.8.2	Procedure for DTMF Generation Test	8-15
8.5.9	G.165 Test.	8-15
8.5.9.1	Set-up for G.165 Test	8-15
8.5.9.2	Procedure for G.165 Test	8-15
8.5.10	G.168 Test.	8-16
8.5.10.1	Set-up for G.168 Test	8-16
8.5.10.2	Procedure for G.168 Test	8-16
8.5.11	G.711 Test.	8-16
8.5.11.1	Set-up for G.711 Test	8-17
8.5.11.2	Procedure for G.711 Test	8-17
8.5.12	G.723.1A Test.	8-17
8.5.12.1	Set-up for G.723.1A Test	8-18
8.5.12.2	Procedure for G.723.1A Test	8-18
8.5.13	G.726 Test.	8-18
8.5.13.1	Set-up for G.726 Test	8-19
8.5.13.2	Procedure for G.726 Test	8-19
8.5.14	G.729AB Test.	8-19
8.5.14.1	Set-up for G.729AB Test	8-19
8.5.14.2	Procedure for G.729AB Encoder Test	8-20
8.5.14.3	Procedure for G.729AB Decoder Test	8-20
8.5.14.4	Procedure for G.729AB Multi-Channel Test	8-21
8.5.15	MFCR2 Detection Test.	8-21
8.5.15.1	Set-up for MFCR2 Detection Test	8-21
8.5.15.2	Procedure for MFCR2 Detection Test	8-22
8.5.16	Noise Suppression (NS) Test	8-22
8.5.16.1	Set-up for Noise Suppression Test	8-22
8.5.16.2	Procedure for NS Test.	8-22
8.5.17	VAD Test	8-23
8.5.17.1	Set-up for VAD Test.	8-23
8.5.17.2	Procedure for VAD Test.	8-23
8.6	Modem Libraries.	8-24
8.6.1	V.21 Loopback Test	8-24
8.6.1.1	Set-up for V.21 Loopback Test.	8-24
8.6.1.2	Procedure for V.21 Loopback Test.	8-24

8.6.2	V.22bis Loopback Test	8-25
8.6.2.1	Set-up for V.22bis Loopback Test	8-25
8.6.2.2	Procedure for V.22bis Loopback Test	8-26
8.6.3	V.8bis Test	8-26
8.6.3.1	Set-up for V.8bis Test	8-27
8.6.3.2	Procedure for V.8bis Test	8-27
8.7	Speech Libraries	8-28
8.7.1	VR Lite-1 Test	8-28
8.7.1.1	Set-up for VR Lite-1 Test	8-29
8.7.1.2	Procedure for VR Lite-1 Test	8-29
8.8	Feature Phone Libraries	8-30
8.8.1	Generic Echo Cancellation Library	8-30
8.8.1.1	Generic Echo Cancellation Library Application	8-30
8.8.1.2	Test Application	8-30
8.8.2	Type 1 and 2 Telephony Parser Library	8-32
8.8.2.1	Type 1 and 2 Telephony Parser Library Verification Test	8-32
8.8.2.1.1	Test Set-up and Procedure	8-32
8.8.2.2	Type 1 and 2 Telephony Parser Library Example Application	8-32
8.8.3	Full Duplex Speakerphone Library	8-36
8.8.3.1	Full Duplex Speakerphone Application	8-36
8.8.3.2	Test Application	8-36
8.8.3.3	Diagnostics Application	8-39
8.8.4	Type 1 Telephony Features Library	8-39
8.8.4.1	Type 1 Verification Test	8-39
8.8.4.1.1	Test Set-up and Procedure	8-39
8.8.4.2	Example Application Using cid1.lib	8-39
8.8.5	Type 1 and 2 Telephony Features Library	8-43
8.8.5.1	Type 2 Verification Test	8-43
8.8.5.1.1	Test Set-up and Procedure	8-43
8.8.5.2	Example Application Using cid12.lib	8-44

Chapter 9 DSP5685x Applications

9.1	BSP Demos	9-1
9.1.1	Codec Demo	9-1
9.1.2	Codec DMA Demo	9-1
9.1.3	COP Demo	9-1
9.1.4	Host Interface Demo	9-1
9.1.5	PC Master Demo	9-2
9.1.6	Quad Timer Demo	9-2
9.1.7	SCI Demo	9-2
9.1.8	SCI Echo Demo	9-2
9.1.9	TDC1 Demo	9-2
9.1.10	Timer Demo	9-2
9.1.11	TOD Demo	9-2
9.2	Security Demos	9-3
9.2.1	DES Demo	9-3



9.2.1.1	DES File I/O demo	9-3
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	9-4
9.2.1.2	DES Realtime Demo	9-4
9.2.1.2.1	Set-up for DES Real-time Demo	9-4
9.2.1.2.2	Procedure for DES Realtime Demo	9-5
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	9-6
9.2.2.1.2	Procedure for Triple DES Demo	9-6
9.2.2.2	Triple DES Realtime Demo	9-6
9.2.2.2.1	Set-up for Triple DES Realtime Demo	9-6
9.2.2.2.2	Procedure for Triple DES Realtime Demo	9-7
9.2.3	RSA Demo	9-7
9.2.3.1	Set-up for RSA Demo	9-8
9.2.3.2	Procedure for RSA Demo	9-8
9.3	Telephony Demos	9-9
9.3.1	Automatic Gain Control (AGC) Demo	9-9
9.3.1.1	Set-up for AGC Demo	9-9
9.3.1.2	Procedure for AGC Demo	9-9
9.3.2	CPT Demo	9-10
9.3.2.1	Set-up for CPT Demo	9-10
9.3.2.2	Procedure for CPT Demo	9-11
9.3.3	Common Tone Generation (CTG) Demo	9-11
9.3.3.1	Set-up for CTG Demo	9-12
9.3.3.2	Procedure for CTG Demo	9-12
9.3.4	DTMF Detection Demo	9-12
9.3.4.1	Set-up for DTMF Detection Demo	9-13
9.3.4.2	Procedure for DTMF Detection Demo	9-13
9.3.5	DTMF Generation Demo	9-14
9.3.5.1	Set-up for DTMF Generation Demo	9-14
9.3.5.2	Procedure for DTMF Generation Demo	9-15
9.3.6	Feature Phone Application	9-16
9.3.7	G.165 Demo	9-16
9.3.7.1	Set-up for G.165 Demo	9-16
9.3.7.2	Procedure for G.165 Demo	9-17
9.3.8	G.168 Demo	9-17
9.3.8.1	Set-up for G.168 Demo	9-17
9.3.8.2	Procedure for G.168 Demo	9-18
9.3.9	G.711 Demo	9-18
9.3.9.1	Set-up for G.711 Demo	9-19
9.3.9.2	Procedure for G.711 Demo	9-19
9.3.10	G.723.1A Demo	9-19
9.3.10.1	Set-up for the G.723.1A Demo	9-20
9.3.10.2	Procedure for G.723.1A Demo	9-20
9.3.11	G.726 Demo	9-20
9.3.11.1	Set-up for the G.726 Demo	9-21
9.3.11.2	Procedure for G.726 Demo	9-21



9.3.12	G729 Demo	9-21
9.3.12.1	Vocoder Recorder Player Demo	9-21
9.3.12.1.1	Set-up for Vocoder Recorder Player Demo	9-22
9.3.12.1.2	Procedure for Vocoder Recorder Player Demo	9-22
9.3.12.2	Vocoder Loopback Demo	9-23
9.3.12.2.1	Set-up for Vocoder Loopback Demo	9-23
9.3.12.3	Procedure for Vocoder Loopback Demo	9-24
9.3.13	Noise Suppression (NS) Demo	9-24
9.3.13.1	Set-up for Noise Suppression Demo	9-24
9.3.13.2	Procedure for Noise Suppression Demo	9-25
9.3.14	VAD Demo	9-25
9.3.14.1	Set-up for VAD Demo	9-25
9.3.14.2	Procedure for VAD Demo	9-25
9.4	Modem Demos	9-26
9.4.1	V.22bis and Demo	9-26
9.4.1.1	V.22bis Demo	9-26
9.4.1.2	Hardware set-up	9-26
9.4.1.3	Answering Modem Demo for 2400bps	9-27
9.4.1.4	Answering Modem Demo for 1200bps	9-28
9.4.1.5	Calling Modem Demo for 2400bps	9-29
9.4.1.6	Calling Modem Demo for 1200bps	9-30
9.4.1.7	Special Considerations During Demo	9-32
9.4.2	V.42bis Demo	9-32
9.4.2.1	Set-up for V.42bis Demo	9-32
9.4.2.2	Procedure for V.42bis Demo	9-32
9.5	Speech Demos	9-33
9.5.1	VRLite-1 Demo	9-33
9.5.1.1	Set-up for VRLite-1 Demo	9-36
9.5.1.2	Procedure for VRLite-1 Demo	9-38
9.5.1.2.1	Procedure for demo_vrlite1	9-38
9.5.1.2.2	Procedure for demo_dial_vrlite1	9-39
9.5.1.3	Demo Limitations	9-40

List of Tables

Table 3-1	DSP5685x OMR Configuration	3-7
Table 3-2	DSP5685x SR Configuration	3-7
Table 3-3	Format of Data in External Memory.	3-31
Table 3-4	Format of Data in SPI Data Flash.	3-33
Table 3-5	Format of Data for Boot from Host Interface.	3-34
Table 3-6	Format of Data for Boot from SCI	3-36
Table 4-1	SDK ISR Overhead for DSP5685x (Running From Internal Memory)	4-2
Table 4-2	SDK ISR Stack Requirements for DSP5685x	4-2
Table 5-1	COP Driver Arguments - copInitialize()	5-5
Table 5-2	DMA Configuration Items for appconfig.h	5-15
Table 5-3	DMA Driver Arguments - open	5-21
Table 5-4	DMA Device Names for DSP5685x Chips	5-21
Table 5-5	DMA Driver Arguments - write	5-23
Table 5-6	DMA Driver Arguments - read.	5-25
Table 5-7	DMA Driver Arguments - close	5-27
Table 5-8	DMA Driver Arguments - ioctl	5-29
Table 5-9	DMA ioctl Commands	5-29
Table 5-10	DMA Driver Arguments - dmaOpen	5-34
Table 5-11	DMA Device Names for DSP5685x Chips	5-34
Table 5-12	DMA Driver Arguments - dmaWrite	5-36
Table 5-13	DMA Driver Arguments - dmaRead	5-38
Table 5-14	DMA Driver Arguments - dmaClose	5-40
Table 5-15	DMA Driver Arguments - dmaIoctl	5-42
Table 5-16	ESSI Configuration Items for appconfig.h	5-45
Table 5-17	ESSI Driver Arguments - open.	5-65
Table 5-18	ESSI Device Names for DSP5685x Chips	5-65
Table 5-19	ESSI Driver Arguments - write	5-66
Table 5-20	ESSI Driver Arguments - read	5-67
Table 5-21	ESSI Driver Arguments - close	5-68
Table 5-22	ESSI Driver Arguments - ioctl	5-69
Table 5-23	ESSI ioctl Commands.	5-69
Table 5-24	ESSI Driver Arguments - essiOpen	5-72
Table 5-25	ESSI Device Names for DSP5685x Chips	5-72
Table 5-26	ESSI Driver Arguments - essiWrite	5-73

Table 5-27	ESSI Driver Arguments - essiRead	5-74
Table 5-28	ESSI Driver Arguments - essiClose	5-75
Table 5-29	ESSI Driver Arguments - essiIoctl	5-76
Table 5-30	ESSI essiIoctl Commands.	5-76
Table 5-31	ESSI DMA Configuration Items for appconfig.h.....	5-80
Table 5-32	ESSI DMA Driver Arguments - open.	5-91
Table 5-33	ESSI DMA Device Names for DSP5685x Chips	5-91
Table 5-34	ESSI DMA Driver Arguments - write	5-92
Table 5-35	ESSI DMA Driver Arguments - read	5-93
Table 5-36	ESSI DMA Driver Arguments - close	5-94
Table 5-37	ESSI DMA Driver Arguments - ioctl	5-95
Table 5-38	ESSI DMA ioctl Commands.	5-95
Table 5-39	ESSI DMA Driver Arguments - essidmaOpen.....	5-99
Table 5-40	ESSI DMA Device Names for DSP5685x Chips	5-99
Table 5-41	ESSI DMA Driver Arguments - essidmaWrite	5-100
Table 5-42	ESSI DMA Driver Arguments - essidmaRead.....	5-101
Table 5-43	ESSI DMA Driver Arguments - essidmaClose	5-102
Table 5-44	ESSI DMA Driver Arguments - essidmaIoctl	5-103
Table 5-45	GPIO Driver Arguments - open	5-109
Table 5-46	GPIO Ports Available for the DSP5685x Chip.....	5-109
Table 5-47	GPIO Driver Arguments - close	5-110
Table 5-48	GPIO Driver Arguments - ioctl	5-111
Table 5-49	GPIO ioctl Commands	5-111
Table 5-50	GPIO Driver Arguments - gpioOpen	5-114
Table 5-51	GPIO Driver Arguments - close	5-115
Table 5-52	GPIO Driver Arguments.....	5-116
Table 5-53	gpiolock Commands.....	5-117
Table 5-54	HI Configuration Items for appconfig.h.....	5-119
Table 5-55	HI Driver Arguments - open.....	5-123
Table 5-56	HI Device Names for DSP5685x Chips	5-123
Table 5-57	HI Driver Arguments - write	5-125
Table 5-58	HI Driver Arguments - read	5-126
Table 5-59	HI Driver Arguments - close	5-127
Table 5-60	HI Driver Arguments - ioctl	5-128
Table 5-61	HI ioctl Commands.....	5-128
Table 5-62	HI Driver Arguments - hiOpen.....	5-132
Table 5-63	HI Device Names for DSP5685x Chips	5-132
Table 5-64	HI Driver Arguments - hiWrite	5-134
Table 5-65	HI Driver Arguments - hiRead	5-135



Table 5-66	HI Driver Arguments - hiClose	5-136
Table 5-67	HI Driver Arguments - hiIoctl	5-137
Table 5-68	HI hiIoctl Commands	5-137
Table 5-69	PCS Argument Values	5-146
Table 5-70	PCS Argument Values	5-150
Table 5-71	Quad Timers	5-153
Table 5-72	Quad Timer Data Structure Members	5-155
Table 5-73	Quad Driver Arguments - open	5-160
Table 5-74	Quad Timer Driver Arguments - close	5-161
Table 5-75	Quad Timer Driver Arguments - ioctl	5-162
Table 5-76	Quad Timer ioctl / qtIoctl Commands	5-162
Table 5-77	Quad Timer Driver Arguments - open	5-167
Table 5-78	Quad Timer Driver Arguments - close	5-168
Table 5-79	Quad Timer Driver Arguments - ioctl	5-169
Table 5-80	SCI Configuration Items for appconfig.h	5-171
Table 5-81	SCI Driver Arguments - open	5-177
Table 5-82	SCI Device Names for DSP5685x Chips	5-177
Table 5-83	SCI Driver Arguments - write	5-178
Table 5-84	SCI Driver Arguments - read	5-179
Table 5-85	SCI Driver Arguments - close	5-180
Table 5-86	SCI Driver Arguments - ioctl	5-181
Table 5-87	SCI ioctl Commands	5-181
Table 5-88	SCI Driver Arguments - sciOpen	5-185
Table 5-89	SCI Device Names for DSP5685x Chips	5-185
Table 5-90	SCI Driver Arguments - sciWrite	5-186
Table 5-91	SCI Driver Arguments - sciRead	5-187
Table 5-92	SCI Driver Arguments - sciClose	5-188
Table 5-93	SCI Driver Arguments - sciIoctl	5-189
Table 5-94	SCI sciIoctl Commands	5-189
Table 5-95	SPI Configuration Items for appconfig.h	5-194
Table 5-96	SPI Driver Arguments - open	5-197
Table 5-97	SPI Driver Arguments - write	5-198
Table 5-98	SPI Driver Arguments - read	5-199
Table 5-99	SPI Driver Arguments - close	5-200
Table 5-100	SPI Driver Arguments - ioctl	5-201
Table 5-101	SPI ioctl Commands	5-201
Table 5-102	SPI Driver Arguments - spiOpen	5-204
Table 5-103	SPI Driver Arguments - spiWrite	5-205
Table 5-104	SPI Driver Arguments - spiRead	5-206
Table 5-105	SPI Driver Arguments - spiClose	5-207

Table 5-106	SPI Driver Arguments - spiIoctl	5-208
Table 5-107	SPI spiIoctl Commands	5-208
Table 5-108	TOD Driver Arguments - todOpen	5-216
Table 5-109	TOD Driver Arguments - todEnableCallbacks	5-217
Table 5-110	TOD Driver Arguments - todSetAlarm	5-218
Table 5-111	TOD Driver Arguments - todGetTime	5-220
Table 5-112	TOD Driver Arguments - todClose	5-221
Table 5-113	TOD Driver Arguments - todIoctl	5-222
Table 5-114	todIoctl Commands	5-222
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	6-28
Table 6-14	Codec Driver Arguments - codecWrite	6-29
Table 6-15	Codec Driver Arguments - codecRead	6-30
Table 6-16	Codec Driver Arguments - codecClose	6-31
Table 6-17	Codec Driver Arguments - codecIoctl	6-32
Table 6-18	Codec codecIoctl Commands	6-32
Table 6-19	Codec DMA Configuration Items for appconfig.h	6-39
Table 6-20	Codec DMA Driver Arguments - open	6-47
Table 6-21	Codec DMA Driver Arguments - write	6-48
Table 6-22	Codec DMA Driver Arguments - read	6-49
Table 6-23	Codec DMA Driver Arguments - close	6-50
Table 6-24	Codec DMA Driver Arguments - ioctl	6-51
Table 6-25	Codec DMA ioctl Commands	6-51
Table 6-26	Codec DMA Driver Arguments - codecdmaOpen	6-57
Table 6-27	Codec DMA Driver Arguments - codecdmaWrite	6-58
Table 6-28	Codec DMA Driver Arguments - codecdmaRead	6-59
Table 6-29	Codec DMA Driver Arguments - codecdmaClose	6-60
Table 6-30	Codec DMA Driver Arguments - codecdmaIoctl	6-61



Table 6-31	File I/O Configuration Items for appconfig.h	6-67
Table 6-32	File I/O Driver Arguments - open.	6-70
Table 6-33	File I/O Driver Arguments - write	6-71
Table 6-34	File I/O Driver Arguments - read	6-72
Table 6-35	File I/O Driver Arguments - close	6-73
Table 6-36	KEYPAD Driver Arguments - open.	6-80
Table 6-37	Keypad Driver Arguments - ioctl	6-81
Table 6-38	Keypad Driver Arguments - close	6-82
Table 6-39	Keypad Driver Arguments - keypadOpen	6-84
Table 6-40	TDC1 Keypad Driver Arguments - keypadIoctl.	6-85
Table 6-41	keypadIoctl Commands	6-85
Table 6-42	TDC1 Keypad Driver Arguments - keypadClose.	6-88
Table 6-43	LCD Driver Arguments - open.	6-95
Table 6-44	LCD Driver Arguments - write.	6-96
Table 6-45	LCD Driver Arguments - ioctl	6-97
Table 6-46	LCD Driver Arguments - close.	6-98
Table 6-47	LCD Driver Arguments - lcdOpen	6-100
Table 6-48	LCD Driver Arguments - lcdWrite.	6-101
Table 6-49	LCD Driver Arguments - lcdIoctl.	6-103
Table 6-50	lcdIoctl Commands.	6-103
Table 6-51	LCD Driver Arguments - lcdClose.	6-105
Table 6-52	LEDs and Peripheral Multiplexing.	6-107
Table 6-53	LED Driver Arguments - open	6-110
Table 6-54	LED Driver Arguments - close.	6-111
Table 6-55	LED Driver Arguments - ioctl	6-112
Table 6-56	LEDs for Specific EVM Boards (pParams - input parameter).	6-112
Table 6-57	LED Ioctl Commands.	6-112
Table 6-58	Low-Level LED Driver Arguments - ledOpen.	6-115
Table 6-59	LED Driver Arguments - ledClose.	6-116
Table 6-60	Low-Level LED Driver Arguments	6-117
Table 6-61	PC Master appconfig.h Settings	6-119
Table 6-62	sPCMasterComm structure.	6-122
Table 6-63	timeBase format	6-122
Table 6-64	pcmasterdrvInit parameters.	6-124
Table 6-65	pcmasterdrvWriteAppCmdSts arguments	6-126
Table 6-66	Serial Data Flash Driver Arguments - open	6-133
Table 6-67	Serial Data Flash Driver Arguments - write.	6-134
Table 6-68	Serial Data Flash Driver Arguments - read	6-135
Table 6-69	Serial Data Flash Driver Arguments - close.	6-136



Table 6-70	Serial Data Flash Driver Arguments.....	6-137
Table 6-71	Serial Data Flash ioctl Commands.....	6-137
Table 6-72	TDC1 Configuration Items for appconfig.h.....	6-142
Table 6-73	TDC1 Driver Arguments - open.....	6-152
Table 6-74	TDC1 Driver Arguments - write.....	6-153
Table 6-75	TDC1 Driver Arguments - read.....	6-154
Table 6-76	TDC1 Driver Arguments - close.....	6-155
Table 6-77	TDC1 Driver Arguments - ioctl.....	6-156
Table 6-78	TDC1 Driver Arguments - tdc1Open.....	6-163
Table 6-79	TDC1 Driver Arguments - tdc1Write.....	6-164
Table 6-80	TDC1 Driver Arguments - tdc1Read.....	6-165
Table 6-81	TDC1 Driver Arguments - tdc1Close.....	6-166
Table 6-82	TDC1 Driver Arguments - tdc1Ioctl.....	6-167
Table 6-83	TDC1 tdc1Ioctl Commands.....	6-167
Table 6-84	IDC Configuration Items for appconfig.h.....	6-178
Table 6-85	IDC Driver Arguments - open.....	6-184
Table 6-86	IDC Driver Arguments - write.....	6-185
Table 6-87	IDC Driver Arguments - read.....	6-186
Table 6-88	IDC Driver Arguments - close.....	6-187
Table 6-90	IDC Ioctl Commands.....	6-188
Table 6-89	IDC Driver Arguments - ioctl.....	6-188
Table 6-91	IDC Driver Arguments - idcOpen.....	6-192
Table 6-92	IDC Driver Arguments - idcWrite.....	6-193
Table 6-93	IDC Driver Arguments - idcRead.....	6-194
Table 6-94	IDC Driver Arguments - idcClose.....	6-195
Table 6-95	IDC Driver Arguments - idcIoctl.....	6-196
Table 7-1	Trigonometric Math Function Performance.....	7-3
Table 7-2	autoCorr Performance.....	7-4
Table 7-3	cbitrev Performance.....	7-4
Table 7-4	cfft Performance.....	7-5
Table 7-5	cifft Performance.....	7-6
Table 7-6	corr Performance.....	7-6
Table 7-7	iir Performance.....	7-7
Table 7-8	rfft Performance.....	7-8
Table 7-9	rifft Performance.....	7-9
Table 7-10	V.8bis Memory and MIPS Requirements for Internal Memory on the DSP5685x.....	7-10
Table 7-11	MIPS on the DSP5685x.....	7-11
Table 7-12	Frequencies Used in V.21.....	7-11

Table 7-13	V.21 Memory and MIPS Requirements for Internal Memory on the DSP5685x	7-12
Table 7-14	V.22bis Memory Requirements on the DSP5685x	7-13
Table 7-15	V.42bis Memory Requirements on the DSP5685x	7-14
Table 7-16	V.42bis Cycle Count for Internal Memory on the DSP5685x	7-14
Table 7-17	DES Memory and MIPS Requirements for Internal Memory on the DSP5685x	7-15
Table 7-18	DES Cycle Counts for Internal Memory on the DSP5685x	7-16
Table 7-19	3DES Memory and MIPS Requirements for Internal Memory on the DSP5685x	7-17
Table 7-20	3DES Cycle Count for Internal Memory on the DSP5685x	7-18
Table 7-21	RSA Memory and MIPS Requirements for Internal Memory on the DSP5685x	7-19
Table 7-22	RSA Cycle Count for Internal Memory on the DSP5685x	7-19
Table 7-23	AEC Memory and MIPS Requirements for Internal Memory on the DSP5685x	7-21
Table 7-24	Automatic Gain Control Memory and MIPS Requirements for Internal Memory on the DSP56858 for Small Memory Model	7-22
Table 7-25	Automatic Gain Control Memory and MIPS Requirements for Internal Memory on the DSP56858 for Large Memory Model	7-22
Table 7-26	CAS Electrical Characteristics	7-23
Table 7-27	CAS Detect Memory and MIPS Requirements for Internal Memory on the DSP5685x	7-23
Table 7-28	Tone Specifications	7-24
Table 7-29	CPT Memory and MIPS Requirements for Internal Memory on the DSP5685x	7-25
Table 7-30	Caller ID Memory and MIPS Requirements for Internal Memory on the DSP5685x	7-26
Table 7-31	Common Tone Generation Memory and MIPS Requirements for Internal Memory on the DSP5685x for Small Memory Model	7-27
Table 7-32	Common Tone Generation Memory and MIPS Requirements for Internal Memory on the DSP5685x for Large Memory Model	7-27
Table 7-33	DTMF Detection Memory and MIPS for Internal Memory on the DSP5685x	7-28
Table 7-34	MIPS and Memory Requirements for Internal Memory on the DSP5685x for Small Memory Model	7-29
Table 7-35	MIPS and Memory Requirements for Internal Memory on the DSP5685x for Large Memory Model	7-29
Table 7-36	G.165 Memory and MIPS Requirements for Internal Memory on the DSP5685x	7-30
Table 7-37	G.168 Memory and MIPS Requirements for Internal Memory on the DSP5685x	7-31



Table 7-38	G.711 Memory and MIPS Requirements for Internal Memory on the DSP5685x for Small Memory Model.	7-32
Table 7-39	G.711 Memory and MIPS Requirements for Internal Memory on the DSP5685x for Large Memory Model.	7-32
Table 7-40	G.723.1A Memory and MIPS Requirements for Internal Memory on the DSP5685x.	7-33
Table 7-41	G.726 Memory and MIPS Requirements for Internal Memory on the DSP5685x7-34	
Table 7-42	Performance Figures for the G.729AB Vocoder Library	7-35
Table 7-43	MFCR2 Detection Memory and MIPS for Internal Memory on the DSP5685x.	7-36
Table 7-44	NS Memory and MIPS Requirements for Internal Memory on the DSP5685x.	7-36
Table 7-45	VAD Memory and MIPS Requirements for Internal Memory on the DSP5685x for Small Memory Model.	7-37
Table 7-46	VAD Memory and MIPS Requirements for Internal Memory on the DSP5685x for Large Memory Model.	7-38
Table 7-47	VRLite-1 Memory and MIPS Requirements for Internal Memory on the DSP5685x.	7-39
Table 7-48	Instruction Cycle Counts for Internal Memory on the DSP5685x.	7-39
Table 7-49	Generic Echo Canceller Memory and MIPS Requirements.	7-40
Table 7-50	Type 1 and 2 Telephony Parser Library Memory and MIPS Requirements . .	7-41
Table 7-51	Full Duplex Speakerphone Memory and MIPS Requirements	7-42
Table 7-52	Type 1 Telephony Features Library Memory and MIPS Requirements	7-42
Table 7-53	Type 1 and 2 Telephony Features Memory and MIPS Requirements	7-43

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\dsp56858evm\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	DSP56858 Directories	2-1
Figure 3-1	Set Up DSP5685x Memory Partitions	3-24
Figure 3-2	Programming Target Memory	3-25
Figure 3-3	Set Up DSP5685x Memory Partitions	3-28
Figure 3-4	Programming Target Memory	3-29
Figure 3-5	DSP5685x Boot Sequence	3-30
Figure 3-6	DSP5685x Boot Sequence With Second Stage SPI Bootloader	3-38
Figure 3-7	SDK Clock Configuration for the DSP5685x	3-43
Figure 5-1	EPP Rev D Board Installed on the DSP56858	5-140
Figure 5-2	TOD Alarm	5-219
Figure 6-1	Codec Configuration	6-43
Figure 6-2	Basic Set-up for File I/O Utilization	6-66
Figure 6-3	PC Master Software Application Window	6-130
Figure 7-1	Triple-DES: (a) Encryption and (b) Decryption	7-17
Figure 8-1	Test Set-up for V.21	8-24
Figure 8-2	V.22bis Set-up for Digital and Analog Loopback Tests	8-26
Figure 8-3	V.8bis Test Set-up	8-27
Figure 8-4	V.8bis Test Set-up with Speakers	8-28
Figure 9-1	DES Demo	9-3
Figure 9-2	DES Demo Set-up	9-4
Figure 9-3	Triple DES Demo	9-5
Figure 9-4	3DES Demo Set-up	9-7
Figure 9-5	RSA Demo	9-8
Figure 9-6	AGC Demo Set-up	9-9
Figure 9-7	CPT Detection Demo Set-up	9-11
Figure 9-8	CTG Demo Set-up	9-12
Figure 9-9	DTMF Detection Demo Set-up	9-13
Figure 9-10	DTMF Generation Demo Set-up	9-15



Figure 9-11	G.165 Demo Set-up	9-16
Figure 9-12	G.168 Demo Set-up	9-18
Figure 9-13	G.711 Demo Set-up	9-19
Figure 9-14	G.723.1A ADPCM Codec Test Set-up.....	9-20
Figure 9-15	G.726 ADPCM Codec Test Set-up.....	9-21
Figure 9-16	Recorder Player Demo Set-up	9-22
Figure 9-17	Vocoder Loopback Demo.....	9-23
Figure 9-18	Noise Suppression Demo Set-up	9-24
Figure 9-19	V.22bis Demo Set-up	9-27
Figure 9-20	V.42bis Demo Set-up	9-32
Figure 9-21	State Diagram of VRLite-1 Demo System.....	9-34
Figure 9-22	Flowchart for VRLite-1 Demo System.....	9-35
Figure 9-23	Demo Set-up for demo_vrlite1	9-37
Figure 9-24	Demo Set-up for demo_dial_vrlite1	9-38

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	CODEC DMA - appconfig.h File	3-4
Code Example 3-10	COP - appconfig.h File	3-4
Code Example 3-11	CPT - appconfig.h File	3-5
Code Example 3-12	CTG - appconfig.h File	3-5
Code Example 3-13	Cycle Count - appconfig.h File	3-5
Code Example 3-14	DES - appconfig.h File	3-6
Code Example 3-15	DMA - appconfig.h File	3-6
Code Example 3-16	DSPFUNC - appconfig.h File	3-7
Code Example 3-17	DTMF_DET - appconfig.h File	3-7
Code Example 3-18	DTMF_GEN - appconfig.h File	3-8
Code Example 3-19	ESSI - appconfig.h File	3-8
Code Example 3-20	ESSI DMA - appconfig.h File	3-8
Code Example 3-21	FILEIO - appconfig.h File	3-9
Code Example 3-22	G.165 - appconfig.h File	3-9
Code Example 3-23	G.168 - appconfig.h File	3-9
Code Example 3-24	G.711 - appconfig.h File	3-10
Code Example 3-25	G.726 - appconfig.h File	3-10
Code Example 3-26	GPIO - appconfig.h File	3-10
Code Example 3-27	Host Interface - appconfig.h File	3-11
Code Example 3-28	IO - appconfig.h File	3-11
Code Example 3-29	ITCN - appconfig.h File	3-11
Code Example 3-30	Keypad - appconfig.h File	3-12
Code Example 3-31	LCD - appconfig.h File	3-12
Code Example 3-32	LED - appconfig.h File	3-12
Code Example 3-33	MEMORY - appconfig.h File	3-13
Code Example 3-34	Noise Suppresion - appconfig.h File	3-13

Code Example 3-35	PC Master Software - appconfig.h File	3-13
Code Example 3-36	PCS - appconfig.h File	3-14
Code Example 3-37	PLL - appconfig.h File	3-14
Code Example 3-38	QUAD_TIMER - appconfig.h File	3-14
Code Example 3-39	RSA - appconfig.h File	3-15
Code Example 3-40	SCI - appconfig.h File	3-15
Code Example 3-41	Serial Data Flash - appconfig.h File	3-15
Code Example 3-42	SIM - appconfig.h File	3-16
Code Example 3-43	SPI - appconfig.h File	3-16
Code Example 3-44	STACK_CHECK - appconfig.h File	3-16
Code Example 3-45	TDC1 - appconfig.h File	3-17
Code Example 3-46	Time of Day - appconfig.h File	3-17
Code Example 3-47	TIMER - appconfig.h File	3-17
Code Example 3-48	UCOS - appconfig.h File	3-18
Code Example 3-49	VAD - appconfig.h File	3-18
Code Example 3-50	V.8BIS - appconfig.h File	3-18
Code Example 3-51	V.21 - appconfig.h File	3-19
Code Example 3-52	V.22BIS - appconfig.h File	3-19
Code Example 3-53	V.42BIS - appconfig.h File	3-19
Code Example 3-54	VR Lite-1- appconfig.h File	3-20
Code Example 3-55	Linker.cmd, External Memory	3-21
Code Example 3-56	Linker.cmd, Internal Memory	3-26
Code Example 3-57	PLL_MUL / appconfig.h File	3-44
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	5-8
Code Example 5-4	Sample appconfig.h Code for Defining a Callback Function	5-18
Code Example 5-5	Open DMA Device	5-22
Code Example 5-6	Write to DMA Device	5-23
Code Example 5-7	Read from DMA Device	5-25
Code Example 5-8	Close DMA Device	5-27
Code Example 5-9	Control DMA Device	5-31
Code Example 5-10	Open DMA Device	5-34
Code Example 5-11	Write to DMA Device	5-36
Code Example 5-12	Read from DMA Device	5-39
Code Example 5-13	Close DMA Device	5-40
Code Example 5-14	Control DMA Device	5-42
Code Example 5-15	ESSI Driver Usage	5-69
Code Example 5-16	ESSI Driver Usage	5-76



Code Example 5-17	Sample appconfig.h code	5-88
Code Example 5-18	ESSI DMA Driver Device-Independent API Example	5-96
Code Example 5-19	ESSI DMA Driver Device-Dependent API Example	5-103
Code Example 5-20	GPIO Driver Usage	5-112
Code Example 5-21	GPIO Driver Usage	5-117
Code Example 5-22	HI Driver Usage	5-130
Code Example 5-23	HI Driver Usage	5-139
Code Example 5-24	appconfig.h file/Chip Select 3	5-148
Code Example 5-25	main.c file/Chip Select 3	5-149
Code Example 5-26	appconfig.h file/EMI	5-151
Code Example 5-27	main.c file/Chip Select 3	5-151
Code Example 5-28	Timer Configuration	5-152
Code Example 5-29	qt_sState Data Structure	5-157
Code Example 5-30	Quad Timer Driver Usage	5-163
Code Example 5-31	Device-Dependent API Quad Timer Driver Usage	5-169
Code Example 5-32	SCI Driver Usage	5-183
Code Example 5-33	SCI Driver Usage	5-191
Code Example 5-34	SPI Driver Settings in appconfig.h	5-200
Code Example 5-35	SPI Driver Usage	5-200
Code Example 5-36	SPI Driver Usage	5-202
Code Example 5-37	SPI Driver Usage with ioctl call	5-207
Code Example 5-38	SPI Driver Usage	5-209
Code Example 5-39	todSetAlarm	5-218
Code Example 5-40	TOD Device-Dependent API Driver Usage	5-224
Code Example 5-41	TOD Device-Independent API Driver Usage	5-225
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 Device-Independent API Example	6-25
Code Example 6-5	Codec Driver Low Level API Example	6-36
Code Example 6-6	Sample appconfig.h code	6-44
Code Example 6-7	Codec DMA Driver Device-Independent API Example	6-53
Code Example 6-8	Codec DMA Driver Device-Dependent API Example	6-62
Code Example 6-9	Sample appconfig.h code	6-67
Code Example 6-10	File I/O Driver Usage	6-74
Code Example 6-11	Use of open	6-80
Code Example 6-12	Use of ioctl	6-81
Code Example 6-13	Use of close	6-82
Code Example 6-14	Use of keypadOpen	6-84



Code Example 6-15	Use of keypadIoctl	6-87
Code Example 6-16	use of keypadClose	6-88
Code Example 6-17	Use of open	6-95
Code Example 6-18	Use of write	6-96
Code Example 6-19	Use of ioctl	6-97
Code Example 6-20	Use of close	6-98
Code Example 6-21	Use of lcdOpen	6-100
Code Example 6-22	Use of lcdWrite	6-101
Code Example 6-23	Use of lcdIoctl	6-104
Code Example 6-24	use of lcdClose	6-105
Code Example 6-25	LED Driver Usage	6-112
Code Example 6-26	Low-Level LED Driver Usage	6-117
Code Example 6-27	Serial Data Flash Driver Usage	6-138
Code Example 6-28	Sample appconfig.h code	6-149
Code Example 6-29	TDC1 Driver Device-Independent API Example	6-157
Code Example 6-30	TDC1 Driver Low-Level API Example	6-172
Code Example 6-31	Sample appconfig.h code	6-179
Code Example 6-32	IDC Driver Usage	6-188
Code Example 6-33	TDC1 Driver Low-Level API Example	6-196
Code Example 8-1	testapp.c	8-30
Code Example 8-2	Use of CIDMessageParser Interface	8-33
Code Example 8-3	testapp.c	8-37
Code Example 8-4	Use of Type1CID Interface	8-40
Code Example 8-5	Use of Type12CID Interface	8-44

About This Book

This manual describes how to use the Motorola Embedded Software Development Kit (SDK) to develop software for Motorola's DSP5685x processor.

Audience

This document targets software developers using Motorola's Embedded Software Development Kit to develop software for Motorola's DSP5685x 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 DSP5685x**,—explains how to change SDK configurations
- **Chapter 4, Interrupt Processing for the DSP5685x**,—describes all of the interrupt processing functions
- **Chapter 5, On-Chip Drivers**,—details the DSP5685x on chip device drivers
- **Chapter 6, Off-Chip Drivers**,—discusses specific evaluation module hardware peripheral drivers included in the DSP5685x
- **Chapter 7, Libraries**,—describes all of the SDK libraries that are not part of the core libraries and are part of the SDK DSP5685x release
- **Chapter 8, Library Tests**,—describes the tests available to confirm that SDK libraries are operating properly
- **Chapter 9, DSP5685x Applications**,—details DSP5685x applications

Suggested Reading

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

- *DSP56800E Reference Manual*, DSP56800ERM/D
- *DSP5685x 16-Bit Digital Signal Processor User's Manual*, DSP5685xUM/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 considered 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.



Freescale Semiconductor, Inc.

ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

3DES	Triple Data Encryption Standard
ADPCM	Adaptive Differential Pulse Code Modulation
AEC	Acoustic Echo Cancellor
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	Program Counter
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. *DSP5685x 16-Bit Digital Signal Processor User's Manual*, DSP5685xUM/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.723.1A Speech Codec Library*, SDK138/D
17. *G.726 Speech Codec Library*, SDK118/D
18. *G.729AB Vocoder Library*, SDK139/D
19. *Customer Premises Equipment Alerting Signal (CAS) Detect Library*, SDK124/D
20. *CPT Detection Library*, SDK123/D
21. *Acoustic Echo Canceller Library*, SDK125/D
22. *Feature Phone Application for DSP56858*, SDK144/D

Chapter 1

Introduction

This manual describes how to use Motorola's Embedded Software Development Kit (SDK) to develop software for the Motorola DSP5685x processor.

1.1 Before You Start

This manual contains information specific to the SDK only as it applies to Motorola DSP5685x 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 DSP56858EVM Hardware

The SDK for DSP5685x has been designed and tested with DSP56858EVM target hardware. If you want to quickly exercise the software applications included with SDK, DSP56858EVM hardware must be installed.

The DSP56858EVM 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 DSP5685x 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](#).

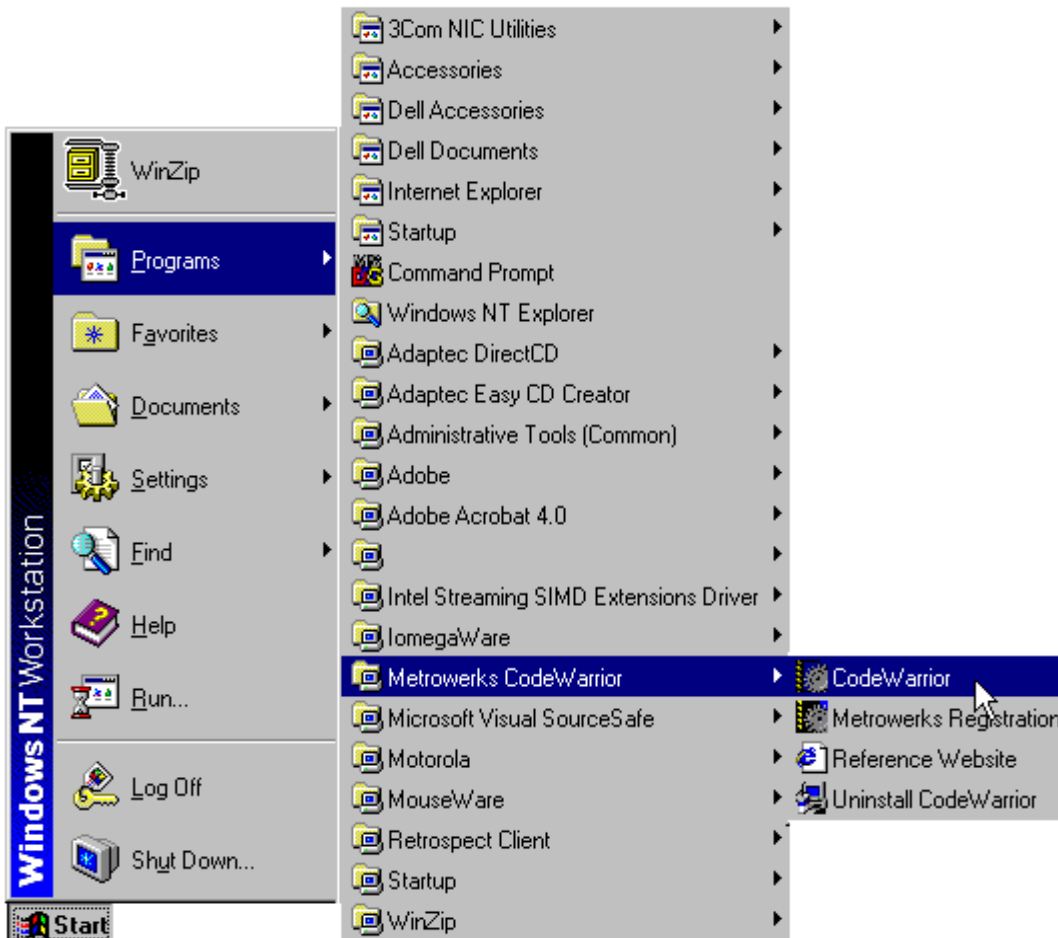


Figure 1-1. Launch CodeWarrior Tools

Step 2: Using the *File/Open* command ([Figure 1-2](#)), open the *buildall.mcp* project by selecting the *...\\sdk\\src\\dsp56858evm\\nos* directory ([Figure 1-3](#)), then choosing the *buildall.mcp* folder ([Figure 1-4](#)). The *buildall.mcp* project is shown in [Figure 1-5](#).

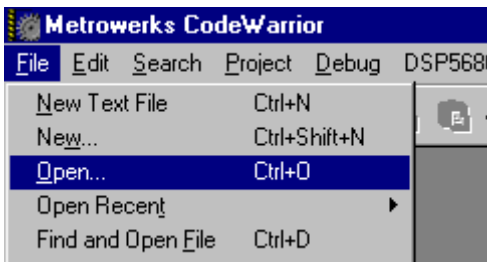


Figure 1-2. Select *File/Open*

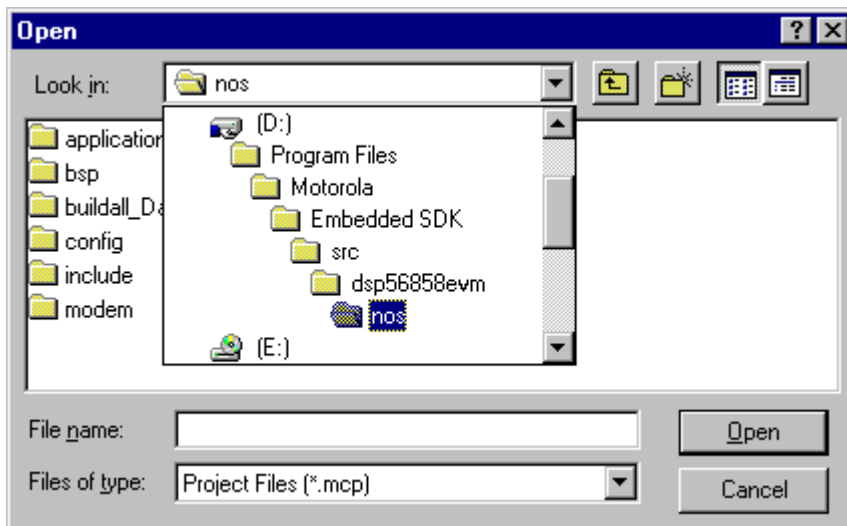


Figure 1-3. Select the ...\\sdk\\src\\dsp56858evm\\nos Directory

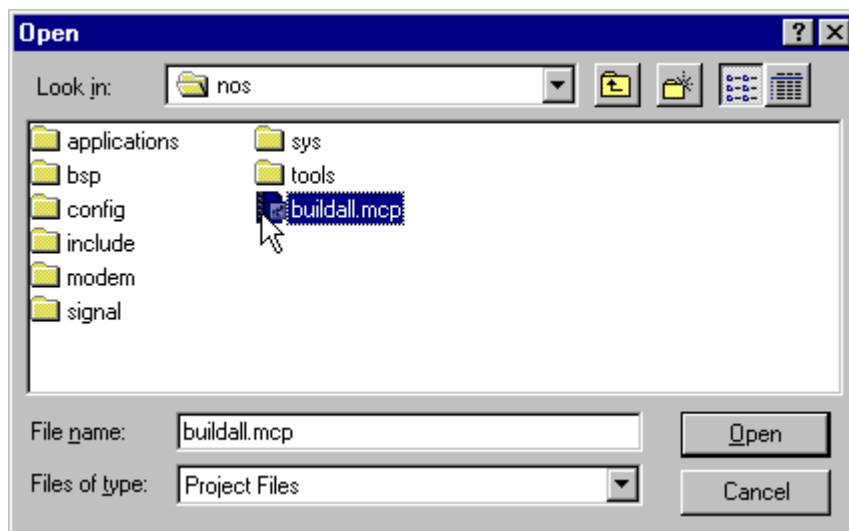


Figure 1-4. Select *buildall.mcp* Project

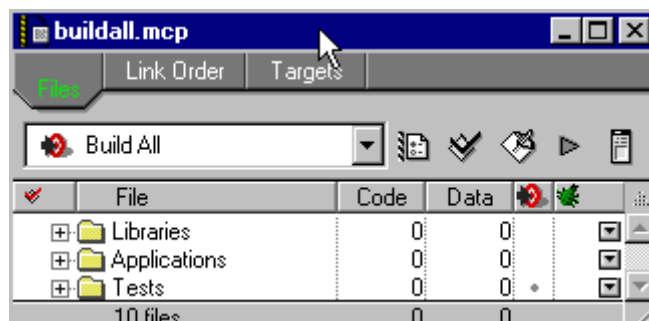


Figure 1-5. *buildall.mcp* Project

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

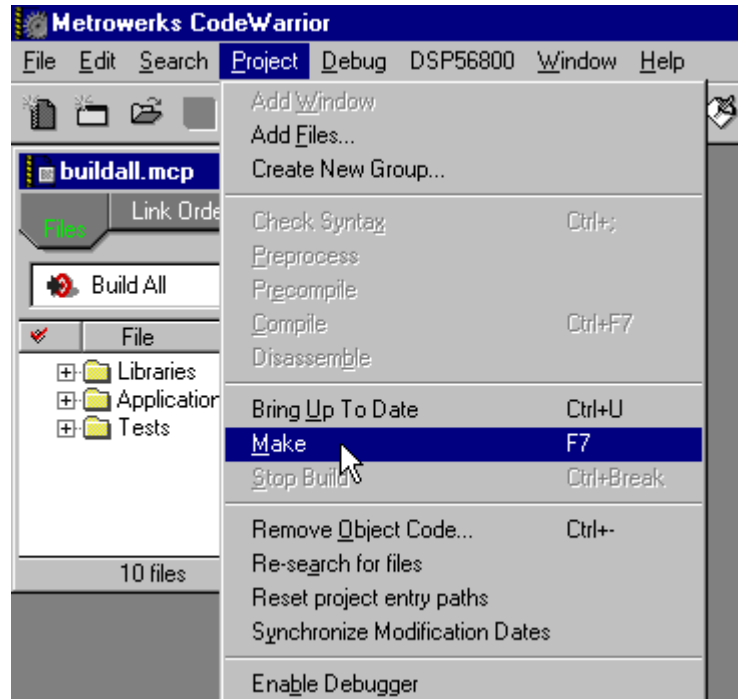


Figure 1-6. Execute *Make* Command

At this point, the *buildall.mcp* project will build every library and application that is part of the SDK DSP5685x release.



ntroduction

Freescal Semiconductor, Inc.

ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

Freescal Semiconductor, Inc.

ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

Chapter 2

Directory Structure

2.1 DSP56858 SDK Directory Structure

Figure 2-1 illustrates the directory structure for DSP56858EVM.

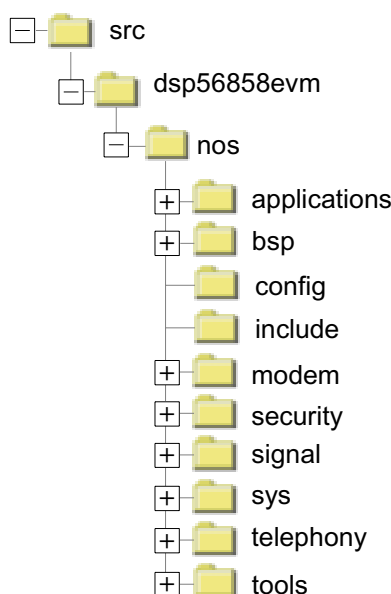


Figure 2-1. DSP56858 Directories

In the SDK DSP56858EVM 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 DSP5685x

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 DSP5685x'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**, **CODEC_DMA**, **COP**, **CPT**, **DES**, **DSPFUNC**, **DTMF_DET**, **DTMF_GEN**, **ESSI**, **ESSI_DMA**, **FILEIO**, **G.165**, **G.168**, **G.711**, **G.726**, **GPIO**, **HI**, **IO**, **ITCN**, **Keypad**, **LCD**, **LED**, **MEMORY**, **NS**, **PC Master Software**, **PCS**, **PLL**, **QUAD_TIMER**, **RSA**, **SCI**, **SERIAL_DATA_FLASH**, **SIM**, **SPI**, **STACK_CHECK**, **TOD**, **TIMER**, **UCOS**, **UCOS**, **V.8bis**, **V.21**, **V.22bis**, **V.42bis**, **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 **ESSI** 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 CODEC_DMA

Macro(s):

INCLUDE_CODECDMA

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

Default: Undefined and not included.

Special Issues: When the CODEC DMA component is used, the **ESSI_DMA** 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 DMA Driver**.

Code Example 3-9. CODEC DMA - *appconfig.h* File

```
#define INCLUDE_CODECDMA
```

3.1.10 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 3-10. COP - *appconfig.h* File

```
#define INCLUDE_COP
```

3.1.11 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 3-11. CPT - *appconfig.h* File

```
#define INCLUDE_CPT
```

3.1.12 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 3-12. CTG - *appconfig.h* File

```
#define INCLUDE_CTG
```

3.1.13 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 3-13. Cycle Count - *appconfig.h* File

```
#define INCLUDE_CYCLECOUNT
```

3.1.14 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 3-14. DES - *appconfig.h* File

```
#define INCLUDE_DES
```

3.1.15 DMA

Macro(s):

INCLUDE_DMA

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

Default: Undefined and not included.

Special Issues: The DMA component is automatically included if the **CODEC_DMA** or **ESSI_DMA** component is defined. Also, when *INCLUDE_IO* is used, IO service is automatically included in the application to provide device-independent interface for the **DMA Driver**.

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

```
#define INCLUDE_DMA
```

3.1.16 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 DSP5685x core registers will be changed:

Table 3-1. DSP5685x 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. DSP5685x SR Configuration

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

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

```
#define INCLUDE_DSPFUNC
```

3.1.17 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 3-17. DTMF_DET - *appconfig.h* File

```
#define INCLUDE_DTMF_DET
```

3.1.18 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 3-18. DTMF_GEN - *appconfig.h* File

```
#define INCLUDE_DTMF_GEN
```

3.1.19 ESSI

Macro(s):

INCLUDE_ESSI

Description: When *INCLUDE_ESSI* is defined and included in the *appconfig.h* file as shown below, the [ESSI Driver](#) will be included in the application.

Default: Undefined and not included.

Special Issues: The ESSI component is automatically included if the [CODEC](#) component is defined. Also, when *INCLUDE_IO* is used, IO service is automatically included in the application to provide device-independent interface for the [ESSI Driver](#).

Code Example 3-19. ESSI - *appconfig.h* File

```
#define INCLUDE_ESSI
```

3.1.20 ESSI_DMA

Macro(s):

INCLUDE_ESSIDMA

Description: When *INCLUDE_ESSIDMA* is defined and included in the *appconfig.h* file as shown below, the [ESSI DMA Driver](#) will be included in the application.

Default: Undefined and not included.

Special Issues: When the ESSI DMA component is used, the [DMA](#) component will automatically be included in the application. The ESSI DMA component is automatically included if the [CODEC_DMA](#) component is defined. Also, when *INCLUDE_IO* is used, IO service is automatically included in the application to provide device-independent interface for the [ESSI DMA Driver](#).

Code Example 3-20. ESSI DMA - *appconfig.h* File

```
#define INCLUDE_ESSIDMA
```

3.1.21 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 3-21. FILEIO - *appconfig.h* File

```
#define INCLUDE_FILEIO
```

3.1.22 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 3-22. G.165 - *appconfig.h* File

```
#define INCLUDE_G165
```

3.1.23 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 3-23. G.168 - *appconfig.h* File

```
#define INCLUDE_G168
```

3.1.24 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 3-24. G.711 - *appconfig.h* File

```
#define INCLUDE_G711
```

3.1.25 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 3-25. G.726 - *appconfig.h* File

```
#define INCLUDE_G726
```

3.1.26 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 3-26. GPIO - *appconfig.h* File

```
#define INCLUDE_GPIO
```

3.1.27 HI

Macro(s):`INCLUDE_HI`

Description: When *INCLUDE_HI* is defined and included in the *appconfig.h* file as shown below, the **Host Interface 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 **Host Interface Driver**.

Code Example 3-27. Host Interface - *appconfig.h* File

```
#define INCLUDE_HI
```

3.1.28 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 3-28. IO - *appconfig.h* File

```
#define INCLUDE_IO
```

3.1.29 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 3-29. ITCN - *appconfig.h* File

```
#define INCLUDE_ITCN
```


3.1.30 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 3-30. Keypad - *appconfig.h* File

```
#define INCLUDE_KEYPAD
```

3.1.31 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 3-31. LCD - *appconfig.h* File

```
#define INCLUDE_LCD
```

3.1.32 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 3-32. LED - *appconfig.h* File

```
#define INCLUDE_LED
```

3.1.33 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 3-33. MEMORY - *appconfig.h* File

```
#define INCLUDE_MEMORY
```

3.1.34 NS

Macro(s):`INCLUDE_NS`

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

Default: Undefined and not included.

Code Example 3-34. Noise Suppression - *appconfig.h* File

```
#define INCLUDE_NS
```

3.1.35 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 3-35. PC Master Software - *appconfig.h* File

```
#define INCLUDE_PCMaster
```

3.1.36 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: Also, when *INCLUDE_IO* is used, IO service is automatically included in the application to provide device-independent interface for the [Programmable Chip Select \(PCS\) Driver](#).

Code Example 3-36. PCS - *appconfig.h* File

```
#define INCLUDE_PCS
```

3.1.37 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 3-37. PLL - *appconfig.h* File

```
#define INCLUDE_PLL
```

3.1.38 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 3-38. QUAD_TIMER - *appconfig.h* File

```
#define INCLUDE_QUAD_TIMER
```

3.1.39 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 3-39. RSA - *appconfig.h* File

```
#define INCLUDE_RSA
```

3.1.40 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 3-40. SCI - *appconfig.h* File

```
#define INCLUDE_SCI
```

3.1.41 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 3-41. Serial Data Flash - *appconfig.h* File

```
#define INCLUDE_SERIAL_DATA_FLASH
```

3.1.42 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 3-42. SIM - appconfig.h File

```
#define INCLUDE_SIM
```

3.1.43 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 3-43. SPI - appconfig.h File

```
#define INCLUDE_SPI
```

3.1.44 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 3-44. STACK_CHECK - appconfig.h File

```
#define INCLUDE_STACK_CHECK
```

3.1.45 TDC1

Macro(s):`INCLUDE_TDC1`

Description: When *INCLUDE_TDC1* is defined and included in the *appconfig.h* file as shown below, the [IDC Driver](#) will be included in the application.

Default: Undefined and not included.

Code Example 3-45. TDC1 - *appconfig.h* File

```
#define INCLUDE_TDC1
```

3.1.46 TOD

Macro(s):`INCLUDE_TIME_OF_DAY`

Description: When *INCLUDE_TIME_OF_DAY* is defined and included in the *appconfig.h* file as shown below, the [Time of Day \(TOD\) Driver](#) will be included in the application.

Default: Undefined and not included.

Code Example 3-46. Time of Day - *appconfig.h* File

```
#define INCLUDE_TIME_OF_DAY
```

3.1.47 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 3-47. TIMER - *appconfig.h* File

```
#define INCLUDE_TIMER
```

3.1.48 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 3-48. UCOS - *appconfig.h* File

```
#define INCLUDE_UCOS
```

3.1.49 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 3-49. VAD - *appconfig.h* File

```
#define INCLUDE_VAD
```

3.1.50 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 3-50. V.8BIS - *appconfig.h* File

```
#define INCLUDE_V8BIS
```

3.1.51 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 3-51. V.21 - *appconfig.h* File

```
#define INCLUDE_V21
```

3.1.52 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 3-52. V.22BIS - *appconfig.h* File

```
#define INCLUDE_V22BIS
```

3.1.53 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 3-53. V.42BIS - *appconfig.h* File

```
#define INCLUDE_V42BIS
```




3.1.54 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 3-54. 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 DSP5685x 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 DSP5685x platform.

3.2.1.1 Linker Command File

Code Example 3-55. *Linker.cmd*, External Memory

```

#*****
#
# Linker.cmd file for DSP56858 Internal RAM
#   using both internal and external program
#   and data memory.
#
#*****
MEMORY {
    .pInterruptVector    (RWX) : ORIGIN = 0x000000, LENGTH = 0x000082
    .pIntrAM              (RWX) : ORIGIN = 0x000082, LENGTH = 0x00177e
    .pExtRAM              (RWX) : ORIGIN = 0x001800, LENGTH = 0x1EE800

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

    .xIntrAM              (RW)   : ORIGIN = 0x000000, LENGTH = 0x000800
    .xIntrAM_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) + SIZEOF(.xStack) - 1;
F_Xdata_start_addr_in_RAM = .;

    # Define variables for SDK mem library

    # Data (X) Memory Layout

        _EX_BIT        = 0;

        # Internal Memory Partitions (for mem.h partitions)

        _NUM_IM_PARTITIONS = 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
```

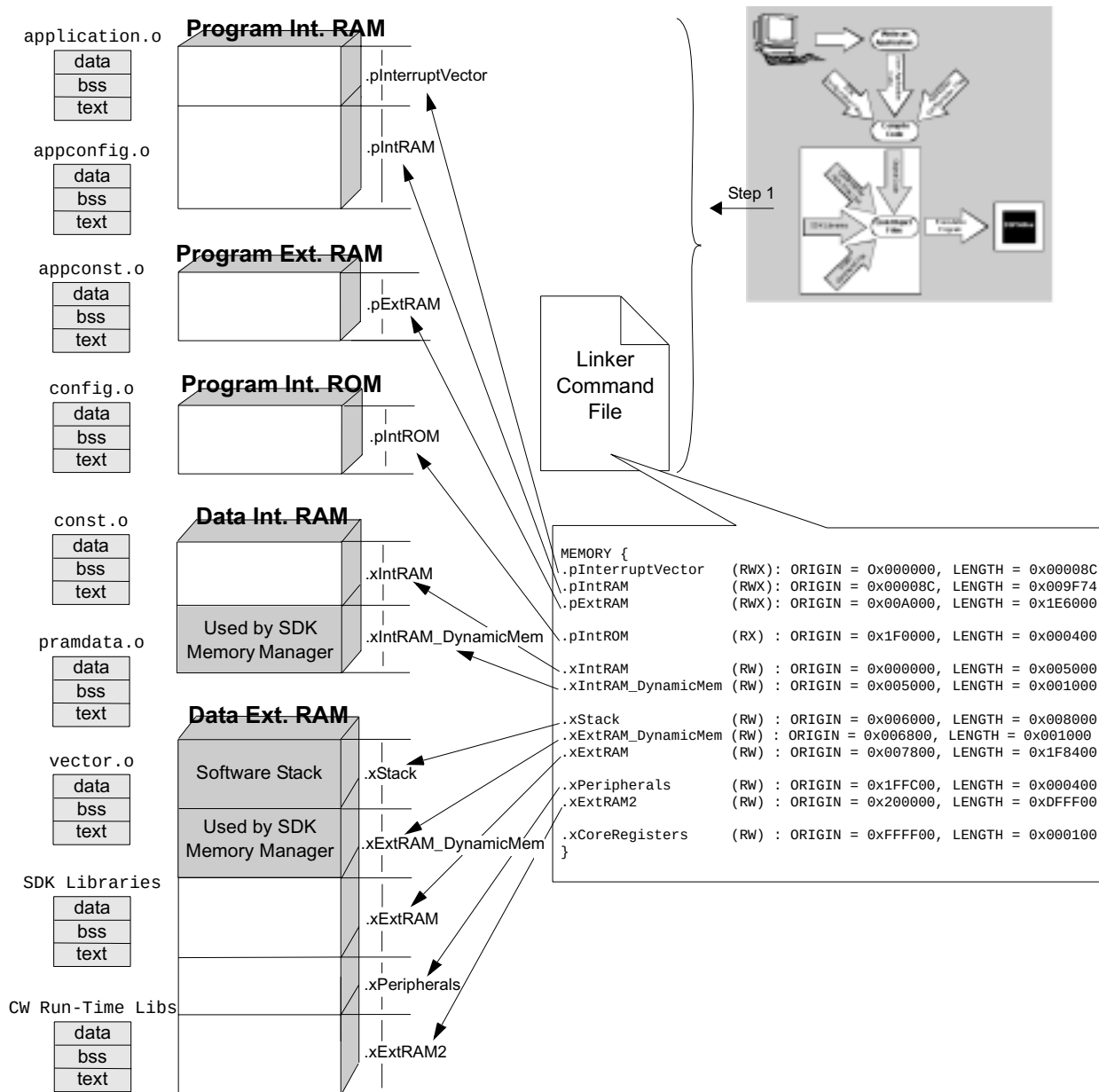
```
*****
```

```
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 DSP5685x 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 set-up of 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 DSP5685x target.

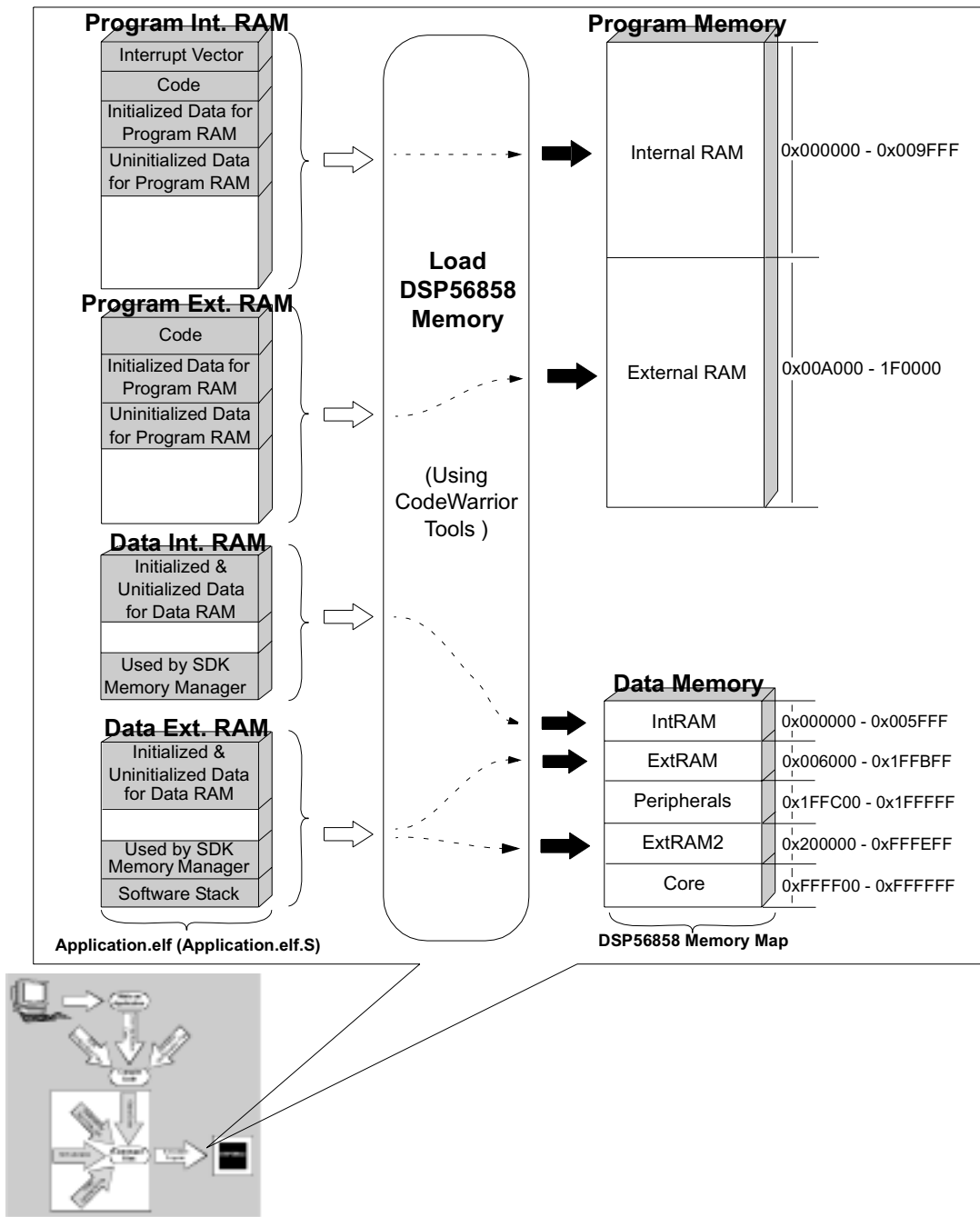


Figure 3-2. Programming Target Memory

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 DSP5685x platform.

3.2.2.1 Linker Command File**Code Example 3-56. *Linker.cmd*, Internal Memory**

```

*****
#
# Linker.cmd file for DSP56858 Internal RAM
#   using only internal program and data memory.
#
*****

MEMORY {

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

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

    .xStack               (RW)  : ORIGIN = 0x000000, LENGTH = 0x000400
    .xIntRAM              (RW)  : ORIGIN = 0x000400, LENGTH = 0x005C00

    .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 DSP5685x 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.

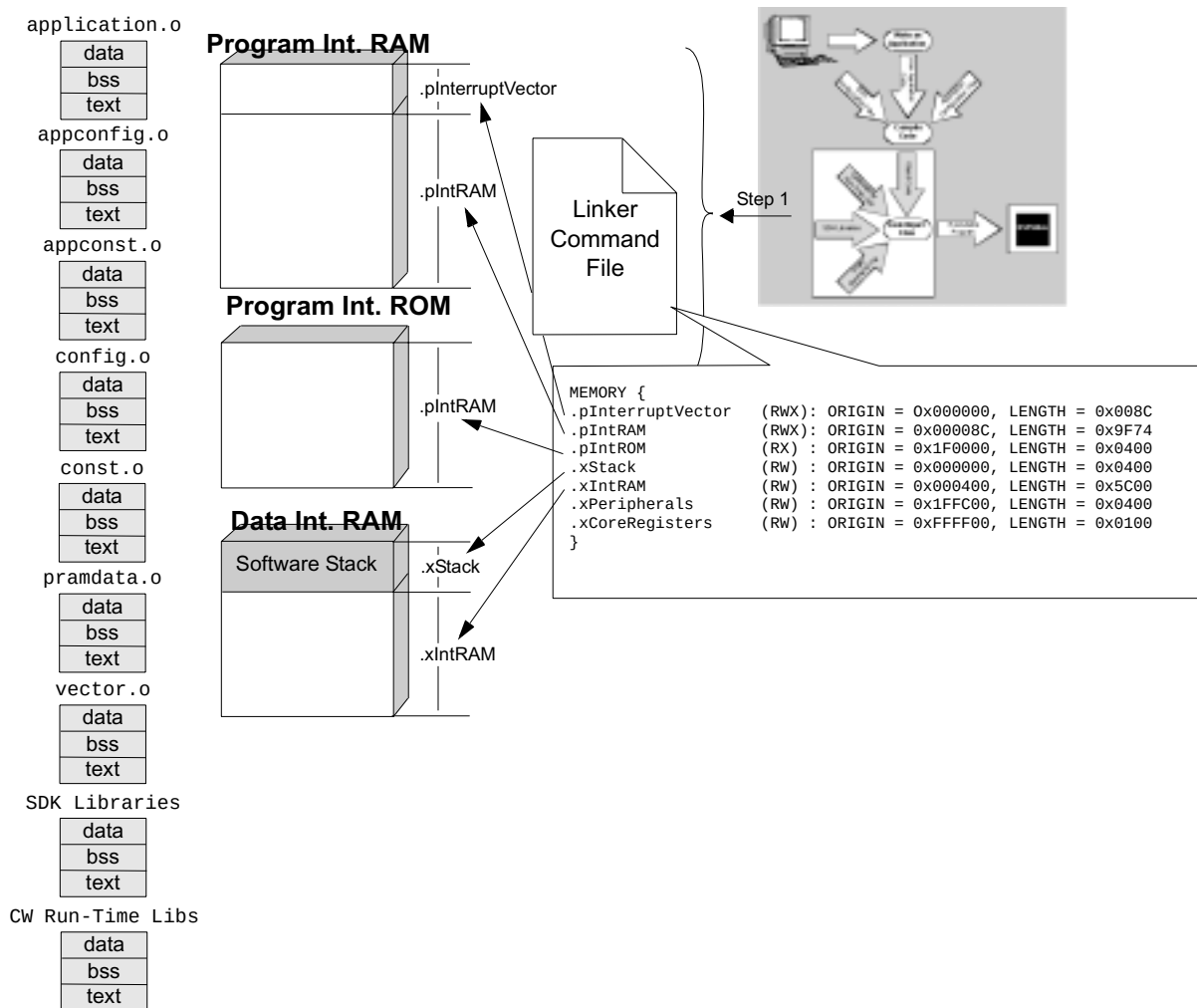
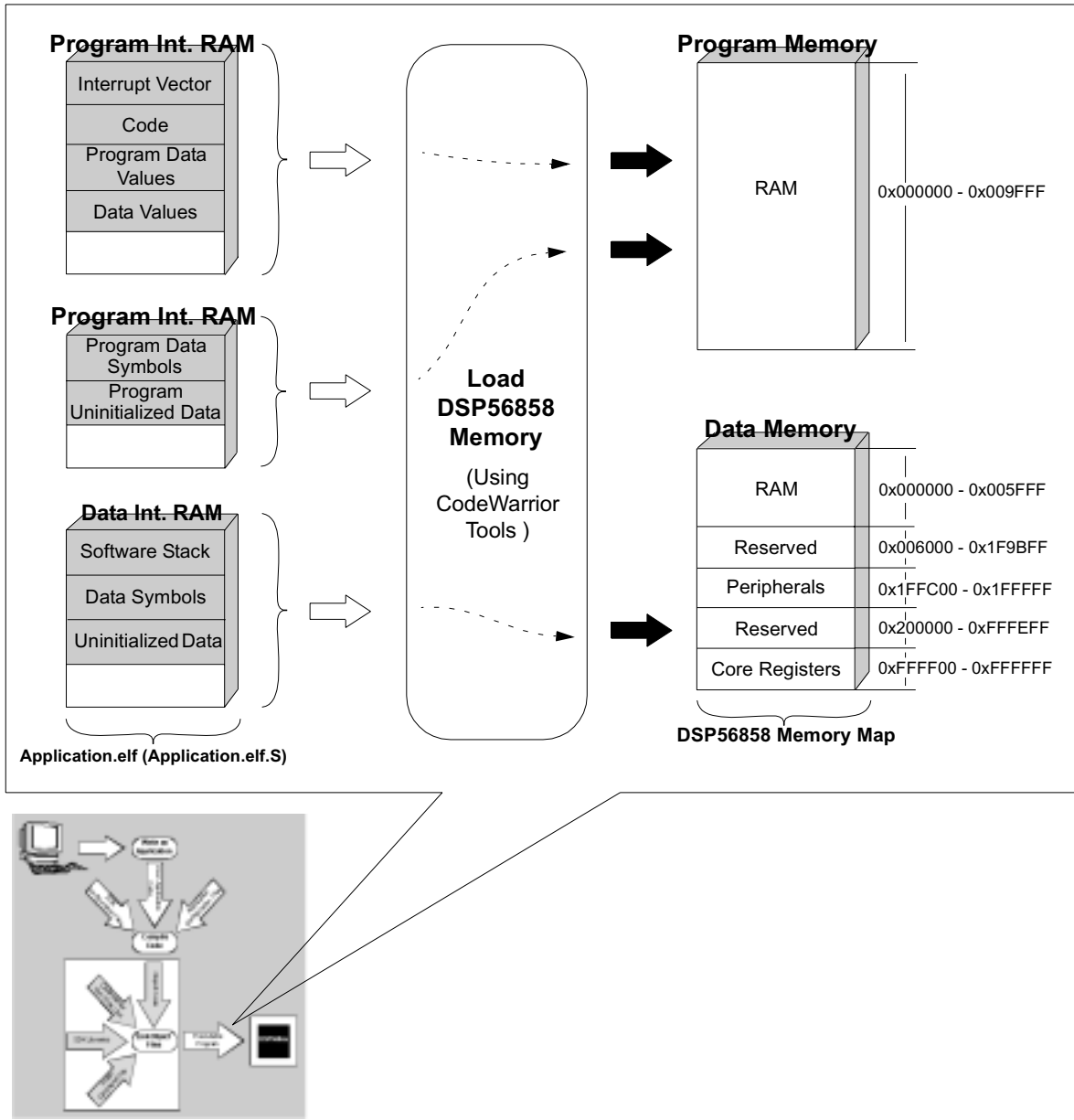


Figure 3-3. Set Up DSP5685x Memory Partitions

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 DSP5685x target.



3.3 DSP5685x Boot Sequence Configuration

In order to execute an application on the DSP5685x, 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 DSP5685x and a Second Stage Bootloader provided in the SDK. [Figure 3-5](#) shows the process of loading a User Application.

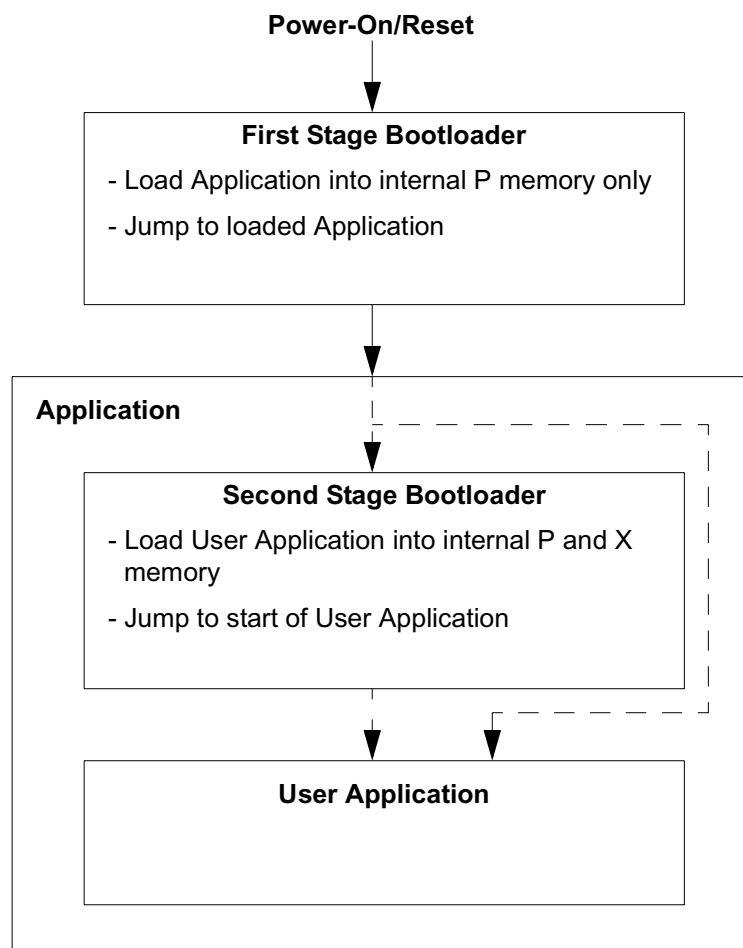


Figure 3-5. DSP5685x Boot Sequence

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 DSP5685x executes its internal boot ROM code (First Stage Bootloader) in order to load the program memory with code. The DSP5685x 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 DSP5685x boots up depends on the settings of the three MOD pins (MODA, MODB, and MODC) connected to the DSP (please see the **DSP5685x 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 DSP5685x 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 DSP5685x 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 Host Interface using Single Strobe (Mode 4)

When configured to boot from the Host Interface using Single Strobe, the First Stage Bootloader will load code from the 16-bit data bus (host port) of a host processor. Mode 4 has all internal program and data RAM memories enabled. Mode 4 is a single strobe bus mode (i.e., a host bus with a single Data Strobe Signal). In Single Strobe Bus mode, a Data Strobe signal qualifies the access, while a Read-Write signal specifies its direction.

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

Table 3-5. Format of Data for Boot from Host Interface

Address	Data Content (16-bit wide)
0x000000	length[0] (LSW)
0x000001	length[1] (MSW)
0x000002	ProgStartAddr[0] (LSW)
0x000003	ProgStartAddr[1] (MSW)
0x000004	prog[0]
0x000005	prog[1]
0x000006	prog[2]
0x000007	prog[3]

Table 3-5. Format of Data for Boot from Host Interface

Address	Data Content (16-bit wide)
0x000008	prog[4]
...	...
0x000003 + length	prog[length-1]

Example

If length = \$87654321, then:

length[1] = \$8765

length[0] = \$4321

If ProgStartAddr = \$87654321, then:

ProgStartAddr[1] = \$8765

ProgStartAddr[0] = \$4321

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 Boot from Host Interface using Dual Strokes (Mode 5)

When configured to boot from the Host Interface using Dual Strokes, the First Stage Bootloader will load code from the 16-bit data bus (host port) of a host processor. Mode 5 has all internal program and data RAM memories enabled, and is a Dual Strobe Bus Mode (i.e., a bus with separated Read and Write data strobes). In Dual Strobe Bus mode, special RD and WR signals each qualify the access as read or write, respectively.

Table 3-5 shows how the data coming from the Host Interface must be organized (identical to Mode 4). The first two words read is the length of the code in words (LSW first, MSW last). The next two words read is the program address where to start storing the code (LSW first, MSW last). From then on, the next word read will be the next word to store in program memory.

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.

3.3.1.7 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 3-6 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 3-6. 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]
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.

3.3.1.8 Unused Boot Mode (Mode 7)

If configured for Mode 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.

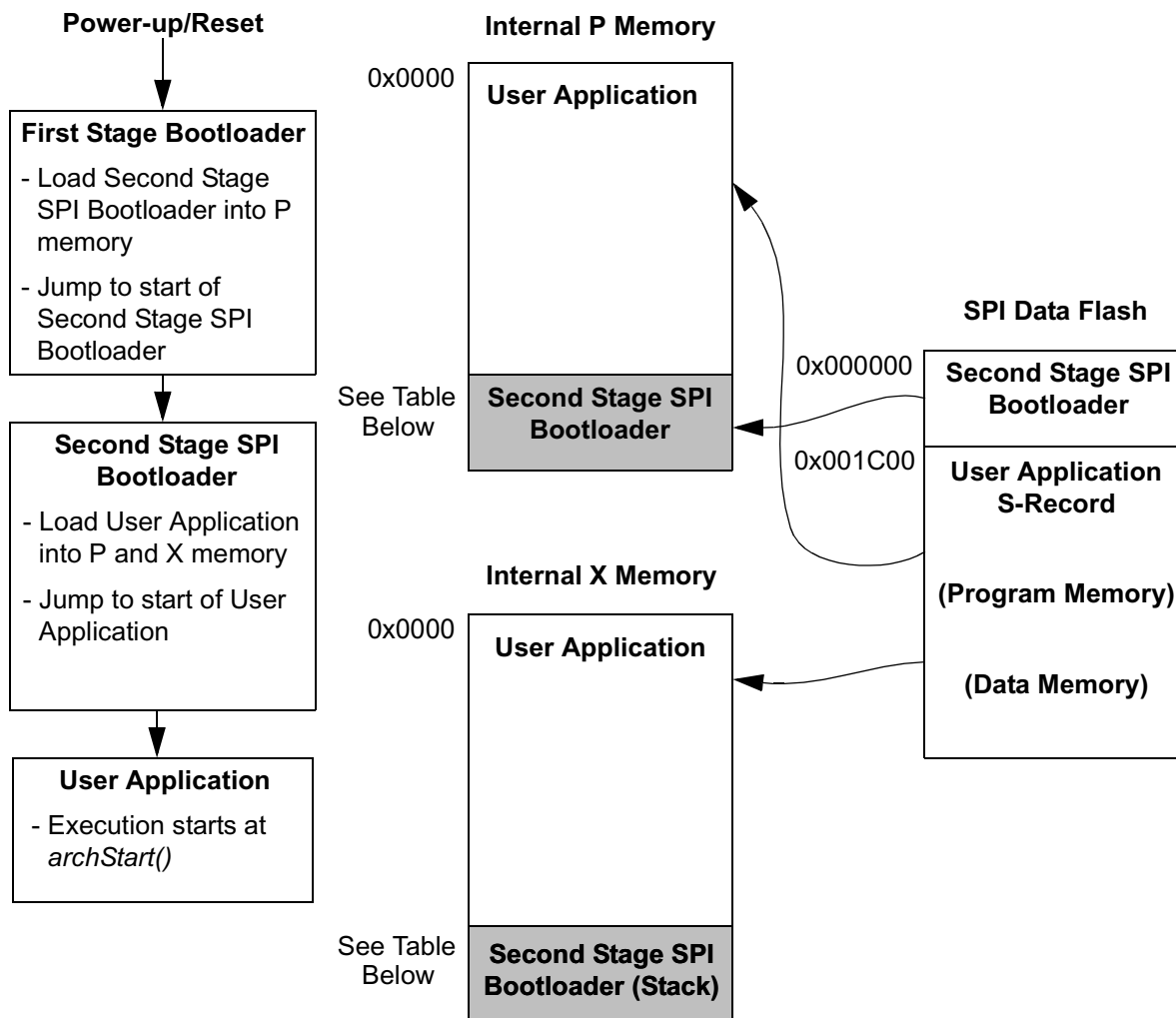
One disadvantage of using a Second Stage Bootloader is that it cannot occupy the same program and data space as the user's application, so the user's application must reserve program and data memory for a Second Stage Bootloader.

For the DSP5685x 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 DSP56858 target. This file must be modified for the DSP56853/4/5/7 targets so that the bootloader will reside in internal memory. The table in [Figure 3-6](#) shows where the bootloader should be located for each of the targets.



Target	Second Stage SPI Bootloader Program Memory		Second Stage SPI Bootloader Data Memory	
	Start	End	Start	End
DSP56853	0x2E30	0x2FFF	0x0FD0	0x0FFF
DSP56854	0x3E30	0x3FFF	0x3FD0	0x3FFF
DSP56855	0x5E30	0x5FFF	0x5FD0	0x5FFF
DSP56857	0x9E30	0x9FFF	0x5FD0	0x5FFF
DSP56858	0x9E30	0x9FFF	0x5FD0	0x5FFF

Figure 3-6. DSP5685x 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 DSP56858EVM 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 DSP56858EVM 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 **DSP56858 Evaluation Module User's Manual**, except :

- Set the Boot MODE (S4) 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 DSP56858EVM board.

3.3.2.1.1.2 Procedure for Programming the Second Stage SPI Bootloader into SPI Data Flash

- 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
 - Re-build 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 DSP56858EVM
  - Remove power from the DSP56858EVM
  - Re-apply power to the DSP56858EVM
  - 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 DSP5685x Boot ROM code.

### 3.3.4 DSP56858EVM 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 DSP5685x 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 DSP5685x clocks. By modifying the SDK default configuration parameters, a user can change the clock frequency for the DSP5685x core and peripherals.

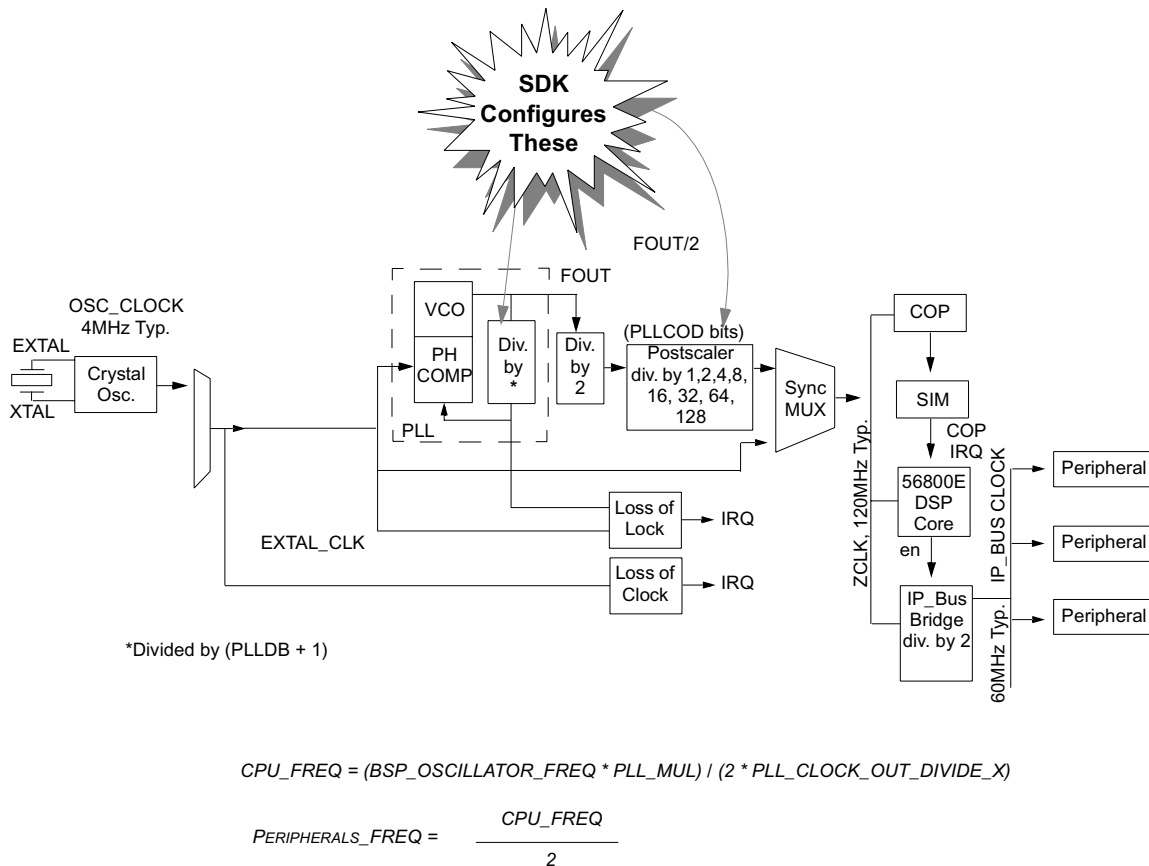


Figure 3-7. SDK Clock Configuration for the DSP5685x

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.



An application developer can use these configuration parameters to select desired CPU speeds. For example, to set a CPU speed to 9MHz, use the equation from [Figure 3-7](#), and redefine the default configuration parameters (from *config.h*) in the *appconfig.h* file as shown in [Code Example 3-57](#).

## Code Example 3-57. 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 3-57](#), *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* are used 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 DSP5685x

This section describes specifically how to implement interrupts on the DSP5685x 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 DSP5685x 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 DSP5685x 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 DSP5685x.

Note that the hardware fast interrupt offered by the DSP5685x 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 DSP5685x 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 DSP5685x (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 DSP5685x**

| 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 DSP5685x 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 DSP5685x 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 4.4.10](#) for details.

### 4.4.3 GPIO Driver

This driver does not use interrupts.

## 4.4.4 Host Interface Driver

When enabled, the Host Interface driver uses interrupt numbers 47 and 48, so the following interrupt priorities are automatically set to 1:

- GPR\_INT\_PRIORITY\_47
- GPR\_INT\_PRIORITY\_48

## 4.4.5 LED Driver

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

## 4.4.6 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

## 4.4.7 SCI Driver

The SCI driver, when enabled, uses interrupt numbers 42 to 46 for SCI0 and interrupt numbers 65 to 69 for SCI1, 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
- GPR\_INT\_PRIORITY\_65
- GPR\_INT\_PRIORITY\_66
- GPR\_INT\_PRIORITY\_67
- GPR\_INT\_PRIORITY\_68
- GPR\_INT\_PRIORITY\_69

## 4.4.8 Serial Data Flash Driver

The Serial Data Flash driver is implemented using the GPIO and SPI drivers; see [Section 4.4.3](#) and [Section 4.4.9](#) for details.

## 4.4.9 SPI Driver

The SPI driver, when enabled, uses interrupt numbers 40 and 41, automatically setting the following interrupt priorities to 1:

- GPR\_INT\_PRIORITY\_40
- GPR\_INT\_PRIORITY\_41

## 4.4.10 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

## 4.4.11 Timer Driver

The Timer driver is implemented using the Quad Timer driver, detailed in [Section 4.4.6](#).



## 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\dsp5685xevm\nos\bsp of the SDK where **dsp5685xevm** is replaced with the specific DSP being used (i.e. **dsp56858evm**).

While there are two major components, Interfaces and Drivers, this chapter focuses on the DSP5685x 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 **DSP56F85x 16-Bit Digital Signal Processor 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.

*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:<br>COP_RUN_IN_STOP - COP module will be active in the STOP mode<br>COP_RUN_IN_WAIT - COP module will be active in the WAIT mode<br>COP_ENABLE - Enables COP timeout processing<br>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 0x0FFF 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:** See [Code Example 5-3](#).

### 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 **DSP56858 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), and 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 **DSP56858 Evaluation Module User's Manual**, except:

- Set the Boot MODE (S4) 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 DSP56858EVM 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 DSP56858EVM
- Remove power from the DSP56858EVM
- Reapply power to the DSP56858EVM
- The LEDs should be blinking
- Press the IRQA Button to prevent the COP from resetting

## 5.2 DMA Driver

### 5.2.1 Introduction

The DSP5685x processor has six DMA Channels. This DMA device allows memory-to-memory, memory-to-peripheral, and peripheral-to-memory transfers.

### 5.2.2 Initialization

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

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

```
#define INCLUDE_DMA
```

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

To use the device-independent Application Programming Interface for the DMA Driver, insert the following line into the *appconfig.h* file:

```
#define INCLUDE_IO
#define INCLUDE_DMA
```

Defining the preprocessor variable, `INCLUDE_IO`, directs the SDK to include the device-independent Application Programming Interface in the executable image. If the preprocessor variable `INCLUDE_IO` is not defined, then the user application must use the device-dependent Application Programming Interface of the DMA driver.

You may also define additional preprocessor variables in the *appconfig.h* file to statically configure and initialize the DMA driver. The list of these variables is provided in [Table 5-2](#). Replace “x” with the DMA Channel being used (0 to 5).

**Table 5-2. DMA Configuration Items for *appconfig.h***

| DMA Defined Constants  | Explanation and Default Value                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DMAx_PERIPHERAL_SELECT | <p>Defines the peripheral the DMA Channel x module will transfer data to/from.</p> <p>Note that when defining DMAx_PERIPHERAL_SELECT to be a peripheral, DMAx_SRC_ADDR or DMAx_DEST_ADDR must be defined to use the appropriate peripheral register.</p> <p>Example: If you are using the Host Interface Transmitter, define:<br/> DMAx_PERIPHERAL_SELECT to be DMA_PERIPH_HOST_TX<br/> DMAx_DEST_ADDR to be DMA_HOST_TX_REG<br/> DMAx_DEST_DELTA to be DMA_DEST_ADDR_NO_CHANGE<br/> DMA_SRC_ADDR will be configured when you call the <i>write</i> function.</p> <p>Valid values are:<br/> DMA_PERIPH_HOST_TX<br/> DMA_PERIPH_HOST_RX<br/> DMA_PERIPH_ESSI1_TX2<br/> DMA_PERIPH_ESSI1_TX1<br/> DMA_PERIPH_ESSI1_TX0<br/> DMA_PERIPH_ESSI1_RX<br/> DMA_PERIPH_ESSI0_TX2<br/> DMA_PERIPH_ESSI0_TX1<br/> DMA_PERIPH_ESSI0_TX0<br/> DMA_PERIPH_ESSI0_RX<br/> DMA_PERIPH_SCI1_TX<br/> DMA_PERIPH_SCI1_RX<br/> DMA_PERIPH_SCI0_TX<br/> DMA_PERIPH_SCI0_RX<br/> DMA_PERIPH_SPI_TX<br/> DMA_PERIPH_SPI_RX<br/> DMA_PERIPH_MEM</p> <p>Default = DMA_PERIPH_MEM</p> |

**Table 5-2. DMA Configuration Items for *appconfig.h***

|                      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DMAx_SRC_ADDR        | <p>Source address of the DMA Channel x transfer. This is the address from which the DMA Channel will read. This address is a word address if DMAx_SAMPLE_SIZE is a word. This address is a byte address if DMAx_SAMPLE_SIZE is a byte.</p> <p>Valid values are:<br/> 0 &lt;= Source Addr &lt;= 0xfffff or<br/> DMA_ESSIO_RX_REG<br/> DMA_ESSI1_RX_REG<br/> DMA_HOST_RX_REG<br/> DMA_SCI0_RX_REG<br/> DMA_SCI1_RX_REG<br/> DMA_SPI_RX_REG<br/> NULL</p> <p>Default = NULL</p>                                                                                                        |
| DMAx_DEST_ADDR       | <p>Destination address of the DMA Channel x transfer. This is the address to which the DMA Channel will write. This address is a word address if DMAx_SAMPLE_SIZE is a word. This address is a byte address if DMAx_SAMPLE_SIZE is a byte.</p> <p>Valid values are:<br/> 0 &lt;= Destination Addr &lt;= 0xfffff or<br/> DMA_ESSIO_TX0_REG<br/> DMA_ESSIO_TX1_REG<br/> DMA_ESSIO_TX2_REG<br/> DMA_ESSI1_TX0_REG<br/> DMA_ESSI1_TX1_REG<br/> DMA_ESSI1_TX2_REG<br/> DMA_HOST_TX_REG<br/> DMA_SCI0_TX_REG<br/> DMA_SCI1_TX_REG<br/> DMA_SPI_TX_REG<br/> NULL</p> <p>Default = NULL</p> |
| DMAx_CIRC_QUEUE_SIZE | <p>Size of the memory buffer used as a circular queue for DMA Channel x transfers. If the size is 0, then the circular buffer is not used.</p> <p><b>Note:</b> The circular buffer size, if used, must be a multiple of the size of the buffer as specified in the <i>read/write</i> functions (NBytes) used in DMA Channel x transfers.</p> <p>Default = 0</p>                                                                                                                                                                                                                     |
| DMAx_SAMPLE_SIZE     | <p>Size of the sample being transferred using DMA Channel x. If a peripheral other than memory is being used, then the sample size must be a word. Only memory-to-memory transfers are allowed to have a sample size of a byte.</p> <p>Valid values are:<br/> DMA_DATA_SIZE_BYTE<br/> DMA_DATA_SIZE_WORD</p> <p>Default = DMA_DATA_SIZE_WORD</p>                                                                                                                                                                                                                                    |

**Table 5-2. DMA Configuration Items for *appconfig.h***

|                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DMAx_SRC_DELTA         | <p>Directs how the DMA Channel x peripheral is to modify the source address after a sample is transferred.</p> <p>Valid values are:<br/> DMA_SRC_ADDR_NO_CHANGE<br/> DMA_SRC_ADDR_INCREMENT<br/> DMA_SRC_ADDR_DECREMENT</p> <p>Default = DMA_SRC_ADDR_INCREMENT</p>                                                                                                                                                                                                                                                                  |
| DMAx_DEST_DELTA        | <p>Directs how the DMA Channel x peripheral is to modify the destination address after a sample is transferred.</p> <p>Valid values are:<br/> DMA_DEST_ADDR_NO_CHANGE<br/> DMA_DEST_ADDR_INCREMENT<br/> DMA_DEST_ADDR_DECREMENT</p> <p>Default = DMA_DEST_ADDR_INCREMENT</p>                                                                                                                                                                                                                                                         |
| DMAx_CALLBACK_FUNCTION | <p>A user function for the DMA driver to call when the DMA transfer is completed for Channel x. The user function must be of type <i>types_tCallback</i>:</p> <pre>typedef void (*types_tCallback)(void *);</pre> <p>Default = NULL</p>                                                                                                                                                                                                                                                                                              |
| DMAx_CALLBACK_ARG      | <p>A value that the user wants to pass to the DMAx_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.</p> <p>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 DMAx_CALLBACK_FUNCTION.</p> <p>Default = NULL</p> |



**Code Example:** [Code Example 5-4](#) illustrates how the *appconfig.h* file can be configured to statically define a callback function for DMA Channel 0 for the DMA Device-Independent Driver. Since nothing else is specified, the DMA Channel 0 peripheral will perform a memory-to-memory transfer.

---

**Code Example 5-4. Sample *appconfig.h* Code for Defining a Callback Function**


---

```
#define INCLUDE_DMA
#define INCLUDE_IO

extern void dmaCallback(void *pCallbackArg);

#define DMA0_CALLBACK_FUNCTION dmaCallback
```

## 5.2.3 API Definition

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

The DMA interface can be used at either of two levels: a device-independent interface and a device-dependent DMA driver interface. The device-independent interface invokes the device-dependent DMA driver 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 interface, but less portable. Your application may use either interface, depending on your specific goals for efficiency and portability. However, you should not mix the DMA 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 DMA driver function call, nor may you use the File Descriptor returned by the device-dependent driver *dmaOpen* call with any device-independent interface function call.

The following information may be found in the public header file *dma.h*. You must include *dma.h* in your application to utilize the DMA driver API.

### 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);
* 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);
*****/
/*****
* The device-dependent API function prototypes:
* types_tHandle dmaOpen(const char *pName, int OFlags);
* ssize_t dmaWrite(types_tHandle FileDesc, const void * pBuffer, size_t NBytes);
* ssize_t dmaRead(types_tHandle FileDesc, void * pBuffer, size_t NBytes);
* int dmaClose(types_tHandle FileDesc);
* UWord16 dmaIoctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams,
* const char * pName);
*****/

```

### Public Data Structure(s):

None

## 5.2.4 DMA Device-Independent API Specification

The following pages specify the Application Programming Interface (API) for the device-independent interface to the DMA driver.

To use the device-independent interface to the DMA driver, you must define both the `INCLUDE_DMA` 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_DMA
```

The user's application code then uses the DMA driver API as follows:

- An *open* call is made to initialize the DMA device
- *ioctl* calls are made to control the DMA device
- *write* calls are made to write data from the user's buffer to the DMA (transmit)
- *read* calls are made to read data from the DMA to the user's buffer (receive)
- After all DMA operations are completed, the DMA driver is closed via a *close* call

The following sections describe each function of the device-independent API for the DMA 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.

### 5.2.4.1 *open*

**Call(s):**

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

**Arguments:**
**Table 5-3. DMA Driver Arguments - *open***

|                 |            |                                                                                                                                    |
|-----------------|------------|------------------------------------------------------------------------------------------------------------------------------------|
| <i>pName</i>    | <i>in</i>  | DMA device name; see <a href="#">Table 5-4</a>                                                                                     |
| <i>OFlag</i>    | <i>in</i>  | Open Mode Flags:<br>- O_RDWR - Read and Write (default)<br>- O_BLOCK - Blocking Mode (default)<br>- O_NONBLOCK - Non-Blocking Mode |
| <i>FileDesc</i> | <i>out</i> | DMA device descriptor returned by <i>open</i> call and used in <i>read/write/close/ioctl</i> function calls                        |

**Table 5-4. DMA Device Names for DSP5685x Chips**

| Chip                                                     | Can Use                                                                                                                                                                          |
|----------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DSP56853<br>DSP56854<br>DSP56855<br>DSP56857<br>DSP56858 | BSP_DEVICE_NAME_DMA_CHAN_0<br>BSP_DEVICE_NAME_DMA_CHAN_1<br>BSP_DEVICE_NAME_DMA_CHAN_2<br>BSP_DEVICE_NAME_DMA_CHAN_3<br>BSP_DEVICE_NAME_DMA_CHAN_4<br>BSP_DEVICE_NAME_DMA_CHAN_5 |

**Description:** This function opens the DMA peripheral for operations for use with the device-independent interface.

This function calls *dmaOpen()*; please see the description of *dmaOpen()* for a description of this function.

**Returns:** The *open* call returns a type *types\_tHandle*. When the *open* call is successful, this *types\_tHandle* is a file descriptor for the DMA device and is used by the DMA's *read*, *write*, *close*, and *ioctl* functions. A “-1” is returned when the *open* call is not successful.

**Code Example:** [Figure 5-5](#) opens DMA Channel 0 for a memory-to-memory transfer using Non-Blocking mode and the device-independent API.

Code Example 5-5. Open DMA Device

---

- file *appconfig.h*

```
#define INCLUDE_BSP
#define INCLUDE_DMA
#define INCLUDE_IO

void dmaCallback(void *pCallbackArg);

#define DMA0_CALLBACK_FUNCTION dmaCallback
```

- user application:

```
#include "port.h"
#include "bsp.h"
#include "dma.h"

void dmaCallback(void *pCallbackArg)
{
}

void main(void)
{
 types_tHandle DmaFD;

 DmaFD = open(BSP_DEVICE_NAME_DMA_CHAN_0, O_NONBLOCK);
}
```

### 5.2.4.2 write

#### Call(s):

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

#### Arguments:

**Table 5-5. DMA Driver Arguments - write**

|                 |           |                                                                                                                                 |
|-----------------|-----------|---------------------------------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | DMA device descriptor returned by <i>open</i> call                                                                              |
| <i>pBuffer</i>  | <i>in</i> | Pointer to user buffer containing data to be written to the DMA; this will be used as the "Source" address for the DMA Transfer |
| <i>NBytes</i>   | <i>in</i> | Number of bytes to write                                                                                                        |

**Description:** This function is mapped to *dmaWrite()* via the *dmadvIOInterfaceVT* structure. Please see the description of *dmaWrite()* for a description of this function

**Returns:** See *dmaWrite*.

#### Code Example 5-6. Write to DMA Device

Using DMA Channel 0, copy data from Buffer1 to Buffer 2 using the *write()* function. The DMA Driver is opened in Non-Blocking mode and the callback function is used to signal when the data has been copied.

- file *appconfig.h*

```
#define INCLUDE_BSP
#define INCLUDE_DMA
#define INCLUDE_IO

void dmaCallback(void *pCallbackArg);

#define DMA0_CALLBACK_FUNCTION dmaCallback
```

- user application:

```
#include "port.h"
#include "bsp.h"
#include "dma.h"

UWord16 Buffer1[10];
UWord16 Buffer2[10];
volatile UWord16 TransferComplete = false;

void dmaCallback(void *pCallbackArg)
{
 TransferComplete = true;
}

void main(void)
{
 types_tHandle DmaFD;
 int i;
```



```
DmaFD = open(BSP_DEVICE_NAME_DMA_CHAN_0, O_NONBLOCK);

for (i = 0; i<10; i++)
{
 Buffer1[i] = i;
 Buffer2[i] = 0;
}

ioctl(DmaFD, DMA_SET_DEST_ADDR, (UWord32) Buffer2);

write(DmaFD, (void *)&Buffer1[0], sizeof(Buffer1));

while (TransferComplete == false)
{
}

}
```

## 5.2.4.3 read

Call(s):

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

Arguments:

**Table 5-6. DMA Driver Arguments - read**

|                 |           |                                                                                                                                    |
|-----------------|-----------|------------------------------------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | DMA device descriptor returned by <i>open</i> call                                                                                 |
| <i>pBuffer</i>  | <i>in</i> | Pointer to the user buffer into which the DMA will write data; this will be used as the "Destination" address for the DMA Transfer |
| <i>NBytes</i>   | <i>in</i> | Number of bytes to read                                                                                                            |

**Description:** This function is mapped to *dmaRead()* via the *dmadrvtIOInterfaceVT* structure. Please see the description of *dmaRead()* for a description of this function.

**Returns:** See *dmaRead*.

### Code Example 5-7. Read from DMA Device

Using DMA Channel 0, copy data from Buffer1 to Buffer 2 using the *read()* function. The DMA Driver is opened in Non-Blocking mode and the callback function is used to signal when the data has been copied.

- file *appconfig.h*

```
#define INCLUDE_BSP
#define INCLUDE_DMA
#define INCLUDE_IO

void dmaCallback(void *pCallbackArg);

#define DMA0_CALLBACK_FUNCTION dmaCallback
```

- user application:

```
#include "port.h"
#include "dma.h"
#include "dma.h"

UWord16 Buffer1[10];
UWord16 Buffer2[10];
volatile UWord16 TransferComplete = false;

void dmaCallback(void *pCallbackArg)
{
 TransferComplete = true;
}

void main(void)
{
 types_tHandle DmaFD;
 int i;
```





```
DmaFD = open(BSP_DEVICE_NAME_DMA_CHAN_0, O_NONBLOCK);

for (i = 0; i<10; i++)
{
 Buffer1[i] = i;
 Buffer2[i] = 0;
}

ioctl(DmaFD, DMA_SET_SRC_ADDR, (UWord32) Buffer1);
read(DmaFD, (void *)&Buffer2[0], sizeof(Buffer2));
while (TransferComplete == false)
{
}
}
```

## 5.2.4.4 close

### Call(s):

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

### Arguments:

**Table 5-7. DMA Driver Arguments - close**

|                 |           |                                                    |
|-----------------|-----------|----------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | DMA device descriptor returned by <i>open</i> call |
|-----------------|-----------|----------------------------------------------------|

**Description:** This function is mapped to *dmaClose()* via the *dmadrvtIOInterfaceVT* structure. Please see the description of *dmaClose()* 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 5-8. Close DMA Device

- file *appconfig.h*:

```
#define INCLUDE_BSP
#define INCLUDE_DMA
#define INCLUDE_IO

void dmaCallback(void *pCallbackArg);

#define DMA0_CALLBACK_FUNCTION dmaCallback
```

- user application:

```
#include "port.h"
#include "bsp.h"
#include "dma.h"

UWord16 Buffer1[10];
UWord16 Buffer2[10];
volatile UWord16 TransferComplete = false;

void dmaCallback(void *pCallbackArg)
{
 TransferComplete = true;
}

void main(void)
{
 types_tHandle DmaFD;
 int i;

 DmaFD = open(BSP_DEVICE_NAME_DMA_CHAN_0, O_NONBLOCK);

 for (i = 0; i<10; i++)
 {
 Buffer1[i] = i;
 Buffer2[i] = 0;
 }

 ioctl(DmaFD, DMA_SET_SRC_ADDR, (UWord32) Buffer1);
```



```
read(DmaFD, (void *)&Buffer2[0], sizeof(Buffer2));

while (TransferComplete == false)
{
}

close(DmaFD);
}
```

### 5.2.4.5 *ioctl*

**Call(s):**

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

**Arguments:**

**Table 5-8. DMA Driver Arguments - *ioctl***

|                 |    |                                                                         |
|-----------------|----|-------------------------------------------------------------------------|
| <i>FileDesc</i> | in | DMA device descriptor returned by <i>open</i> call                      |
| <i>Cmd</i>      | in | Commands found in <i>dma.h</i> which are used to control the DMA driver |
| <i>pParams</i>  | in | Pointer to <i>ioctl</i> command parameter                               |

**Table 5-9. DMA *ioctl* Commands**

| Cmd                  | <i>pParams</i>                 | Returns | Result                                                                                                                                                                                                                                                                                                                              |
|----------------------|--------------------------------|---------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DMA_DEVICE_DISABLE   | NULL                           | None    | This command disables the DMA and DMA interrupt. It should not be used unless an exception has occurred and the user wants to know how much data has already been transferred.                                                                                                                                                      |
| DMA_DEVICE_ENABLE    | NULL                           | None    | This command enables the DMA and DMA interrupt. The <i>read/write</i> functions enable the DMA automatically, so this command should not be needed the DMA must be re-enabled after calling the DMA_DEVICE_DISABLE command.                                                                                                         |
| DMA_CALLBACK         | void (*types_tCallback) (void) | None    | This command installs the DMA Callback function, which is called when the DMA peripheral has completed a transfer. If <i>pParams</i> is defined as NULL, then this command disables the callback function.                                                                                                                          |
| DMA_SET_CALLBACK_ARG | void * pCallbackArg            | None    | This command defines the callback argument to be used when the callback function is called.                                                                                                                                                                                                                                         |
| DMA_SET_SRC_ADDR     | UWord32 src_addr               | None    | This command sets the source address to be used by the DMA peripheral. The DMA peripheral receives its data from this source.<br><br>Valid values for <i>src_addr</i> are:<br>0 <= <i>src_addr</i> <= 0xfffff or<br>DMA_ESSIO_RX_REG<br>DMA_ESSI1_RX_REG<br>DMA_HOST_RX_REG<br>DMA_SCI0_RX_REG<br>DMA_SCI1_RX_REG<br>DMA_SPI_RX_REG |

**Table 5-9. DMA ioctl Commands (Continued)**

| Cmd                        | pParams                          | Returns | Result                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|----------------------------|----------------------------------|---------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DMA_SET_DEST_ADDR          | UWord32<br>dest_addr             | None    | <p>This command sets the destination address to be used by the DMA peripheral. The DMA peripheral writes its data to this destination.</p> <p>Valid values for dest_addr are:<br/>           0 &lt;= dest_addr &lt;= 0xfffff or<br/>           DMA_ESSIO_TX0_REG<br/>           DMA_ESSIO_TX1_REG<br/>           DMA_ESSIO_TX2_REG<br/>           DMA_ESSI1_TX0_REG<br/>           DMA_ESSI1_TX1_REG<br/>           DMA_ESSI1_TX2_REG<br/>           DMA_HOST_TX_REG<br/>           DMA_SCI0_TX_REG<br/>           DMA_SCI1_TX_REG<br/>           DMA_SPI_TX_REG</p> |
| DMA_SET_SRC_ADDR_DELTA     | dma_eSrcAddrDelta<br>src_delta   | None    | <p>This command configures how the DMA peripheral will modify the source address after a sample is transferred</p> <p>Valid values for src_delta are:<br/>           DMA_SRC_ADDR_NO_CHANGE<br/>           DMA_SRC_ADDR_INCREMENT<br/>           DMA_SRC_ADDR_DECREMENT</p>                                                                                                                                                                                                                                                                                          |
| DMA_SET_DEST_ADDR_DELTA    | dma_eDestAddrDelta<br>dest_delta | None    | <p>This command configures how the DMA peripheral will modify the destination address after a sample is transferred.</p> <p>Valid values for dest_delta are:<br/>           DMA_DEST_ADDR_NO_CHANGE<br/>           DMA_DEST_ADDR_INCREMENT<br/>           DMA_DEST_ADDR_DECREMENT</p>                                                                                                                                                                                                                                                                                |
| DMA_SET_SAMPLE_SIZE        | dma_eDataSize<br>sample_size     | None    | <p>This command sets the sample size to be used by the DMA peripheral</p> <p>Valid values for sample_size are:<br/>           DMA_DATA_SIZE_WORD<br/>           DMA_DATA_SIZE_BYTE</p>                                                                                                                                                                                                                                                                                                                                                                               |
| DMA_SET_RX_CIRC_QUEUE_SIZE | UWord16<br>queue_size            | None    | <p>This command sets the size of receive circular queue to be used by the DMA peripheral. If set to 0, then the receiver circular queue is disabled.</p>                                                                                                                                                                                                                                                                                                                                                                                                             |
| DMA_GET_STATUS             | NULL                             | Status  | <p>This command gets the status of the DMA driver.</p> <p>Valid return values are:<br/>           DMA_STATUS_BUSY<br/>           DMA_STATUS_IDLE</p>                                                                                                                                                                                                                                                                                                                                                                                                                 |

**Table 5-9. DMA *ioctl* Commands (Continued)**

| Cmd                | pParams                       | Returns        | Result                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|--------------------|-------------------------------|----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DMA_SET_PERIPHERAL | dma_ePeriphType<br>peripheral | None           | This command configures a DMA Channel to use the peripheral selected. It also configures the appropriate Source or Destination Address (based on the peripheral)<br><br>Valid values for the peripheral are:<br>DMA_PERIPH_HOST_TX<br>DMA_PERIPH_HOST_RX<br>DMA_PERIPH_ESSI1_TX2<br>DMA_PERIPH_ESSI1_TX1<br>DMA_PERIPH_ESSI1_TX0<br>DMA_PERIPH_ESSI1_RX<br>DMA_PERIPH_ESSI0_TX2<br>DMA_PERIPH_ESSI0_TX1<br>DMA_PERIPH_ESSI0_TX0<br>DMA_PERIPH_ESSI0_RX<br>DMA_PERIPH_SCI1_TX<br>DMA_PERIPH_SCI1_RX<br>DMA_PERIPH_SCI0_TX<br>DMA_PERIPH_SCI0_RX<br>DMA_PERIPH_SPI_TX<br>DMA_PERIPH_SPI_RX<br>DMA_PERIPH_MEM |
| DMA_READ_DEST_ADDR | NULL                          | dest<br>addr   | This command returns the 32-bit destination address of a DMA Channel                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| DMA_READ_SRC_ADDR  | NULL                          | source<br>addr | This command returns the 32-bit source address of a DMA Channel                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |

**Description:** The *ioctl* call changes DMA device modes or reports DMA driver status.

**Returns:** The *ioctl* call return value is dependent on the command passed in as an input parameter. [Table 5-9](#) lists the return for each command.

**Code Example:** [Figure 5-9](#) demonstrates how to disable the DMA Channel 0 Device.

---

**Code Example 5-9. Control DMA Device**


---

- file *appconfig.h*:

```

#define INCLUDE_BSP
#define INCLUDE_DMA
#define INCLUDE_IO

void dmaCallback(void *pCallbackArg);

#define DMA0_CALLBACK_FUNCTION dmaCallback

```

- user application:

```
#include "port.h"
#include "bsp.h"
#include "dma.h"

void dmaCallback(void *pCallbackArg)
{
}

void main(void)
{
 types_tHandle DmaFD;

 DmaFD = open(BSP_DEVICE_NAME_DMA_CHAN_0, O_NONBLOCK);

 ioctl(DmaFD, DMA_DEVICE_DISABLE, NULL);
}
```

## 5.2.5 DMA Device-Dependent API Specification

The following pages specify the Application Programming Interface (API) for the device-dependent interface to the DMA driver.

To use the device-dependent interface to the DMA driver, you must define the `INCLUDE_DMA` variable in the `appconfig.h` file of the SDK Embedded Project you created in CodeWarrior:

```
#define INCLUDE_DMA
```

The user's application code then uses the DMA driver API as follows:

- A *dmaOpen* call is made to initialize the DMA device
- *dmaIoctl* calls are made to control the DMA device
- *dmaWrite* calls are made to write data from the user's buffer to the DMA (transmit)
- *dmaRead* calls are made to read data from the DMA to the user's buffer (receive)
- After all operations are completed, the DMA driver is closed via a *dmaClose* call

The following sections describe each function of the device-dependent API for the DMA 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.



### 5.2.5.1 *dmaOpen*

**Call(s):**

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

**Arguments:**

**Table 5-10. DMA Driver Arguments - *dmaOpen***

|                 |            |                                                                                                                                    |
|-----------------|------------|------------------------------------------------------------------------------------------------------------------------------------|
| <i>pName</i>    | <i>in</i>  | The DMA device name; see <a href="#">Table 5-11</a>                                                                                |
| <i>OFlag</i>    | <i>in</i>  | Open Mode Flags:<br>- O_RDWR - Read and Write (default)<br>- O_BLOCK - Blocking Mode (default)<br>- O_NONBLOCK - Non-Blocking Mode |
| <i>FileDesc</i> | <i>out</i> | DMA device descriptor returned by the <i>dmaOpen</i> call and used in <i>dmaRead/dmaWrite/dmaClose/dmaIoctl</i> function calls     |

**Table 5-11. DMA Device Names for DSP5685x Chips**

| Chip                                                     | Can Use                                                                                                                                                                          |
|----------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DSP56853<br>DSP56854<br>DSP56855<br>DSP56857<br>DSP56858 | BSP_DEVICE_NAME_DMA_CHAN_0<br>BSP_DEVICE_NAME_DMA_CHAN_1<br>BSP_DEVICE_NAME_DMA_CHAN_2<br>BSP_DEVICE_NAME_DMA_CHAN_3<br>BSP_DEVICE_NAME_DMA_CHAN_4<br>BSP_DEVICE_NAME_DMA_CHAN_5 |

**Description:** The *dmaOpen* function opens a specific channel of the DMA peripheral for operations. The DMA device is always opened for *read* and for *write* calls. The *dmaOpen* function establishes the handle for the DMA driver. It initializes the internal memory buffers used by the DMA driver. It configures the driver for Blocking or Non-Blocking modes, as defined in the *OFlags* parameter.

**Returns:** The *dmaOpen* call returns a type *types\_tHandle*. When the *dmaOpen* call is successful, this *types\_tHandle* is a file descriptor for the DMA device and is used by the DMA's *dmaRead*, *dmaWrite*, and *dmaIoctl* functions. A “-1” is returned when the *dmaOpen* call is not successful.

**Code Example:** [Figure 5-10](#) opens DMA Channel 0 for a memory-to-memory transfer using Non-Blocking mode and the device-dependent API

#### Code Example 5-10. Open DMA Device

- file *appconfig.h*:

```
#define INCLUDE_BSP
#define INCLUDE_DMA

void dmaCallback(void *pCallbackArg);

#define DMA0_CALLBACK_FUNCTION dmaCallback
```



- user application:

```
#include "port.h"
#include "bsp.h"
#include "dma.h"

void dmaCallback(void *pCallbackArg)
{
}

void main(void)
{
 types_tHandle DmaFD;

 DmaFD = dmaOpen(BSP_DEVICE_NAME_DMA_CHAN_0, 0_NONBLOCK);
}
```

## 5.2.5.2 *dmaWrite*

### Call(s):

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

### Arguments:

**Table 5-12. DMA Driver Arguments - *dmaWrite***

|                 |           |                                                                                                                               |
|-----------------|-----------|-------------------------------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | DMA device descriptor returned by <i>dmaOpen</i> call                                                                         |
| <i>pBuffer</i>  | <i>in</i> | Pointer to user buffer containing data to be written to the DMA; it will be used as the "Source" address for the DMA Transfer |
| <i>NBytes</i>   | <i>in</i> | Number of bytes to write                                                                                                      |

**Description:** The *dmaWrite* function configures the source address of DMA Channel X to transfer *NBytes* from *pBuffer*. The destination address should be configured (i.e., DMA\_ESSI0\_TX1\_REG) prior to calling *dmaWrite()* using a *dmaIoctl(DMA\_SET\_DEST\_ADDR)* command or statically via *appconfig.h*.

If the sample size is defined to be DMA\_DATA\_SIZE\_BYTE, then *pBuffer* and *NBytes* are used as the "source address" and "transfer count". If the sample size is DMA\_DATA\_SIZE\_WORD, then *pBuffer* is typecast to be a word pointer and *NBytes* is divided by two to get a word count. In this case, *NBytes* should always be even. Note that if *NBytes* is odd when the sample size is DMA\_DATA\_SIZE\_WORD, then a byte will be lost.

In Blocking mode, *dmaWrite* returns after the DMA peripheral has read *NBytes* from *pBuffer*, at which point the user's application must call *dmaWrite* in order to start another transfer.

In Non-Blocking mode, *dmaWrite* starts the transfer operation and returns control to the application. It does not wait until all data is transferred via DMA. The user must define a DMA callback function or use the *dmaIoctl(DMA\_GET\_STATUS)* command to determine when the DMA transfer is complete.

The *dmaWrite* function also enables the DMA in order to get the transfer started.

**Returns:** The *dmaWrite* call returns a type *ssize\_t*. It is the actual size (in 8-bit bytes) of the data written. In Non-Blocking mode, the size will be 0, since the *dmaWrite* function returns before any data is written. In Blocking mode, the size is the actual size of the data written.

### Code Example 5-11. Write to DMA Device

Using DMA Channel 0, copy data from Buffer1 to Buffer 2 using the *dmaWrite()* function. The DMA Driver is opened in Non-Blocking mode and the callback function is used to signal when the data has been copied.

- file *appconfig.h*:

```
#define INCLUDE_BSP
#define INCLUDE_DMA
```

ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

```
void dmaCallback(void *pCallbackArg);
```

```
#define DMA0_CALLBACK_FUNCTION dmaCallback
```

- user application:

```
#include "port.h"
#include "bsp.h"
#include "dma.h"

UWord16 Buffer1[10];
UWord16 Buffer2[10];
volatile UWord16 TransferComplete = false;

void dmaCallback(void *pCallbackArg)
{
 TransferComplete = true;
}

void main(void)
{
 types_tHandle DmaFD;
 int i;

 DmaFD = dmaOpen(BSP_DEVICE_NAME_DMA_CHAN_0, O_NONBLOCK);

 for (i = 0; i<10; i++)
 {
 Buffer1[i] = i;
 Buffer2[i] = 0;
 }

 dmaIoctl(DmaFD, DMA_SET_DEST_ADDR, (UWord32) Buffer2,
 BSP_DEVICE_NAME_DMA_CHAN_0);

 dmaWrite(DmaFD, (void *)&Buffer1[0], sizeof(Buffer1));

 while (TransferComplete == false)
 {
 }
}
```

### 5.2.5.3 *dmaRead*

#### Call(s):

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

#### Arguments:

**Table 5-13. DMA Driver Arguments - *dmaRead***

|                 |           |                                                                                                                              |
|-----------------|-----------|------------------------------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | DMA device descriptor returned by <i>dmaOpen</i> call                                                                        |
| <i>pBuffer</i>  | <i>in</i> | Pointer to the user buffer into which DMA will write data; it will be used as the "Destination" address for the DMA Transfer |
| <i>NBytes</i>   | <i>in</i> | Number of bytes to read                                                                                                      |

#### Description:

The *dmaRead* function configures the destination address of DMA Channel X to transfer *NBytes* to *pBuffer*. The source address should be configured (i.e., DMA\_ESSI0\_RX\_REG) prior to calling *dmaRead()* using a *dmaIoctl(DMA\_SET\_SRC\_ADDR)* command or statically via *appconfig.h*.

If the sample size is defined to be DMA\_DATA\_SIZE\_BYTE, then *pBuffer* and *NBytes* are used as the "destination address" and "transfer count". If the sample size is DMA\_DATA\_SIZE\_WORD, then *pBuffer* is typecast to be a word pointer and *NBytes* is divided by two to get a word count. In this case, *NBytes* should always be even. Note that if *NBytes* is odd when the sample size is DMA\_DATA\_SIZE\_WORD, then a byte will be lost.

In Blocking mode, *dmaRead* returns after DMA has transferred *NBytes* to *pBuffer*, at which point the user's application can call *dmaRead()* in order to start another transfer.

In Non-Blocking mode, *dmaRead* starts the transfer operation and returns control to the application. It does not wait until all data is transferred via DMA. The user must define a DMA callback function or use the *dmaIoctl(DMA\_GET\_STATUS)* command to determine when the DMA transfer is complete.

If the user's application defines the size of the receiver circular queue to be greater than zero, then only one *dmaRead* call is required. After that, the DMA Callback function will be called each time *NBytes* is transferred into the circular queue.

**Note:** The size of the Circular Queue must be a multiple of *NBytes*.

The *dmaRead* function also enables the DMA in order to get the transfer started.

**Returns:** The *dmaRead* call returns a type *ssize\_t*, which is the actual size in bytes of the data read. In Non-Blocking mode, the size will be 0, since the *dmaRead* function returns before any data is read. In Blocking mode, the size is the actual size of the data read.

**Code Example 5-12. Read from DMA Device**

Using DMA Channel 0, copy data from Buffer1 to Buffer 2 using the *dmaRead()* function. The DMA Driver is opened in Non-Blocking mode and the callback function is used to signal when the data has been copied.

- file *appconfig.h*:

```
#define INCLUDE_BSP
#define INCLUDE_DMA

void dmaCallback(void *pCallbackArg);

#define DMA0_CALLBACK_FUNCTION dmaCallback
```

- user application:

```
#include "port.h"
#include "bsp.h"
#include "dma.h"

UWord16 Buffer1[10];
UWord16 Buffer2[10];
volatile UWord16 TransferComplete = false;

void dmaCallback(void *pCallbackArg)
{
 TransferComplete = true;
}

void main(void)
{
 types_tHandle DmaFD;
 int i;

 DmaFD = dmaOpen(BSP_DEVICE_NAME_DMA_CHAN_0, 0_NONBLOCK);

 for (i = 0; i<10; i++)
 {
 Buffer1[i] = i;
 Buffer2[i] = 0;
 }

 dmaIoctl(DmaFD, DMA_SET_SRC_ADDR, (UWord32) Buffer1,
 BSP_DEVICE_NAME_DMA_CHAN_0);

 dmaRead(DmaFD, (void *)&Buffer2[0], sizeof(Buffer2));

 while (TransferComplete == false)
 {
 }
}
```

### 5.2.5.4 *dmaClose*

**Call(s):**

```
int dmaClose (types_tHandle FileDesc);
```

**Arguments:**

**Table 5-14. DMA Driver Arguments - *dmaClose***

|                 |           |                                                       |
|-----------------|-----------|-------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | DMA device descriptor returned by <i>dmaOpen</i> call |
|-----------------|-----------|-------------------------------------------------------|

**Description:** The *dmaClose* function closes the DMA device by disabling the DMA channel.

**Returns:** The *dmaClose* call returns a type *int*. It returns “0” if successful, and “-1” if unsuccessful.

**Code Example:** [Figure 5-13](#) demonstrates how to close the DMA device.

#### Code Example 5-13. Close DMA Device

- file *appconfig.h*:

```
#define INCLUDE_BSP
#define INCLUDE_DMA

void dmaCallback(void *pCallbackArg);

#define DMA0_CALLBACK_FUNCTION dmaCallback
```

- user application:

```
#include "port.h"
#include "bsp.h"
#include "dma.h"

UWord16 Buffer1[10];
UWord16 Buffer2[10];
volatile UWord16 TransferComplete = false;

void dmaCallback(void *pCallbackArg)
{
 TransferComplete = true;
}

void main(void)
{
 types_tHandle DmaFD;
 int i;

 DmaFD = dmaOpen(BSP_DEVICE_NAME_DMA_CHAN_0, O_NONBLOCK);

 for (i = 0; i<10; i++)
 {
 Buffer1[i] = i;
 Buffer2[i] = 0;
 }

 dmaIoctl(DmaFD, DMA_SET_SRC_ADDR, (UWord32) Buffer1,
```



ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

BSP\_DEVICE\_NAME\_DMA\_CHAN\_0);

dmaRead(DmaFD, (void \*)&Buffer2[0], sizeof(Buffer2));

while (TransferComplete == false)

{  
{  
}

dmaClose(DmaFD);

}

ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005



## 5.2.5.5 *dmaIoctl*

### Call(s):

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

### Arguments:

**Table 5-15. DMA Driver Arguments - *dmaIoctl***

|                 |           |                                                                          |
|-----------------|-----------|--------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | DMA device descriptor returned by <i>dmaOpen</i> call                    |
| <i>Cmd</i>      | <i>in</i> | Commands found in <i>dma.h</i> which are used to control the DMA driver  |
| <i>pParams</i>  | <i>in</i> | Pointer to <i>dmaIoctl</i> command parameter                             |
| <i>pName</i>    | <i>in</i> | The DMA device name from <i>bsp.h</i> ; (i.e BSP_DEVICE_NAME_DMA_CHAN_0) |

**Table 5-9** contains the list of commands for the DMA *ioctl* function, which are identical to the *dmaIoctl* function.

**Description:** The *dmaIoctl* call changes DMA device modes or reports DMA driver status.

**Returns:** The *dmaIoctl* call returns a value that is dependent on the command passed in as an input parameter. **Table 5-9** lists these a return for each command.

**Code Example:** **Figure 5-14** demonstrates how to disable the DMA Channel 0 device.

### Code Example 5-14. Control DMA Device

- file *appconfig.h*:

```
#define INCLUDE_BSP
#define INCLUDE_DMA

void dmaCallback(void *pCallbackArg);

#define DMA0_CALLBACK_FUNCTION dmaCallback
```

- user application:

```
#include "port.h"
#include "bsp.h"
#include "dma.h"

void dmaCallback(void *pCallbackArg)
{
}

void main(void)
{
 types_tHandle DmaFD;

 DmaFD = dmaOpen(BSP_DEVICE_NAME_DMA_CHAN_0, 0_NONBLOCK);

 dmaIoctl(DmaFD, DMA_DEVICE_DISABLE, NULL, BSP_DEVICE_NAME_DMA_CHAN_0);
```



}

**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

## 5.2.6 DMA Driver Tests

DMA Driver Tests verify correct operation of the DMA Driver provided by the SDK.

### 5.2.6.1 DMA Memory-to-Memory Test

The DMA memory-to-memory test is an automated test that verifies that all six DMA channels can be used for memory-to-memory transfers when using the DMA Driver. This test also verifies static and dynamic initialization of the DMA Driver.

#### 5.2.6.1.1 Set-up for DMA Memory-to-Memory Test

##### EVM Jumper Settings:

Use default settings as shown in the **DSP56858 Evaluation Module User's Manual**.

#### 5.2.6.1.2 Procedure for DMA Memory-to-Memory Test

- Using CodeWarrior, open:  
...\\nos\\bsp\\test\\dma\\testdma.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 test results on CodeWarrior's console window

## 5.2.7 DMA Driver Demos

### 5.2.7.1 DMA Peripheral-to-Memory and Memory-to-Peripheral Tests

Please refer to the Codec DMA Driver application in [Section 6.3](#). The Codec DMA application uses the Codec DMA driver, implemented using the ESSI DMA Driver, which in turn uses the DMA Driver.

## 5.3 ESSI Driver

### 5.3.1 Introduction

The DSP56858 processor has two ESSI devices, ESSI0 and ESSI1. The SDK provides an ESSI driver with limited functionality. Presently, the ESSI driver supports only a mechanism for initializing the ESSI.

### 5.3.2 Initialization

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

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

```
#define INCLUDE_ESSI
```

Defining the preprocessor variable, INCLUDE\_ESSI, directs the SDK to include the ESSI driver in the executable image. If the preprocessor variable INCLUDE\_ESSI is not defined, then the ESSI 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 ESSI driver. The list of these variables is provided in [Table 5-16](#).

The ESSI's default configuration is defined in *config.h* and is shown in [Table 5-16](#). Most of the ESSI #defines in [Table 5-16](#) corresponds directly to a bit or group of bits in the ESSI control registers. Please refer to the **DSP5685x 16-Bit Digital Signal Processor User's Manual** for a detailed description of these bits.

Table 5-16. ESSI Configuration Items for *appconfig.h*

| ESSI #define       | Explanation and Default Value                                                                                                                                                                                                                                                             |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ESSI0_FIFO_SIZE    | <p>The default size (number of 16-bit samples) for both the RX FIFO and TX FIFO buffers used internally by the driver for ESSI0</p> <p>For more information, refer to ESSI0_RX_FIFO_SIZE and ESSI0_TX_FIFO_SIZE</p> <p>Default = 32</p>                                                   |
| ESSI0_RX_FIFO_SIZE | <p>The size (number of 16-bit samples) of the RX FIFO buffer used internally by the driver for the RX channel of ESSI0. The length of the RX FIFO buffer determines the maximum number of ESSI input samples that can be buffered within the driver.</p> <p>Default = ESSI0_FIFO_SIZE</p> |

Table 5-16. ESSI Configuration Items for *appconfig.h* (Continued)

| ESSI #define            | Explanation and Default Value                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ESSI0_TX_FIFO_SIZE      | <p>The size (number of 16-bit samples) of the TX FIFO buffer used internally by the driver for the TX channel of ESSI0. The length of the TX FIFO buffer determines the maximum number of ESSI output samples that can be buffered within the driver.</p> <p>Default = ESSI0_FIFO_SIZE</p>                                                                                                                                                                                                                                                                                                                                                                                                            |
| ESSI0_RX_CALLBACK_LEVEL | <p>The number of 16-bit samples that the ESSI must receive before calling the user function specified by ESSI0_RX_CALLBACK. The value specified must be less than ESSI0_RX_FIFO_SIZE.</p> <p>Increasing the ESSI0_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.</p> <p>Default = 8</p>                                                                                                                 |
| ESSI0_RX_CALLBACK       | <p>A user function for the ESSI0 driver to call when it has received at least ESSI0_RX_CALLBACK_LEVEL samples in its internal RX buffer. The user function must be of type <i>essi_tCallback</i>:</p> <pre>typedef void (*essi_tCallback) (void *pCallbackArg, int MaxNBytes);</pre> <p>The parameters to the callback function are:</p> <p><i>pCallbackArg</i> - The original value specified by the user for ESSI0_RX_CALLBACK_ARG</p> <p><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</p> <p>Default = NULL</p> |
| ESSI0_RX_CALLBACK_ARG   | <p>A value that the user would like passed to the ESSI0_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.</p> <p>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 ESSI0_RX_CALLBACK function.</p> <p>Default = NULL</p>                                                                                                                                                  |

Table 5-16. ESSI Configuration Items for *appconfig.h* (Continued)

| ESSI #define            | Explanation and Default Value                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|-------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ESSI0_TX_CALLBACK_LEVEL | <p>The minimum number of 16-bit samples that the ESSI0 must maintain in the transmit FIFO buffer. If the number of samples falls below ESSI0_TX_CALLBACK_LEVEL, the driver calls the user function specified by ESSI0_TX_CALLBACK. The value specified must be less than ESSI0_TX_FIFO_SIZE.</p> <p>Increasing the ESSI0_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.</p> <p>Default = 8</p> |
| ESSI0_TX_CALLBACK       | <p>A user function for the ESSI0 driver to call when the TX buffer has fallen below ESSI0_TX_CALLBACK_LEVEL samples. The user function must be of type <i>essi_tCallback</i>:</p> <pre>typedef void (*ssi_tCallback) (void *pCallbackArg, int MaxNBytes);</pre> <p>The parameters to the callback function are:</p> <p><i>pCallbackArg</i> - The original value specified by the user for ESSI0_TX_CALLBACK_ARG</p> <p><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</p> <p>Default = NULL</p>                                         |
| ESSI0_TX_CALLBACK_ARG   | <p>A value that the user would like passed to the ESSI0_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.</p> <p>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 ESSI0_TX_CALLBACK function.</p> <p>Default = NULL</p>                                                                                                                                        |
| ESSI0_NUM_TIME_SLOTS    | <p>The number of time slots to be used when ESSI0 is configured for network mode.</p> <p>Default = 2</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| ESSI0_HW_FIFO_DEPTH     | <p>The ESSI0 contains a Receive and Transmit hardware FIFO that can buffer up to 8 samples. This value defines how many samples the ESSI0 ISR will process.</p> <p>Default = 8</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |

**Table 5-16. ESSI Configuration Items for *appconfig.h* (Continued)**

| ESSI #define                     | Explanation and Default Value                                                                                                                                                                            |
|----------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ESSI0_RX_FIFO_FULL_WM            | When the ESSI0 RX FIFO contains this many samples, the ESSI0 Rx ISR will be called, if enabled.<br><br>Default = 8                                                                                       |
| ESSI0_TX_FIFO_EMPTY_WM           | When the ESSI0 TX FIFO contains this many empty samples, the ESSI0 Tx ISR will be called, if enabled.<br><br>Default = 8                                                                                 |
| ESSI0_PRESCALER                  | Bit 15 of ESSI0_RXTX_CONTROL_INITIAL_STATE<br>Valid values are:<br>ESSI_PRESCALER_1<br>ESSI_PRESCALER_8<br><br>Default = ESSI_PRESCALER_1                                                                |
| ESSI0_WORD_LENGTH                | Bits 13 and 14 of ESSI0_RXTX_CONTROL_INITIAL_STATE<br>Valid values are:<br>ESSI_WORD_LENGTH_8<br>ESSI_WORD_LENGTH_10<br>ESSI_WORD_LENGTH_12<br>ESSI_WORD_LENGTH_16<br><br>Default is ESSI_WORD_LENGTH_16 |
| ESSI0_DC                         | Used by ESSI0_FRAME_RATE_DIVIDER<br>Valid values are 1 to 32<br><br>Default = ESSI0_NUM_TIME_SLOTS                                                                                                       |
| ESSI0_PM                         | Used by ESSI0_PRESCALE_MODULUS<br>Valid values are:<br>1 to 256<br><br>Default = 58 (48K Sample Rate)                                                                                                    |
| ESSI0_FRAME_RATE_DIVIDER         | Bits 8 to 12 of ESSI0_RXTX_CONTROL_INITIAL_STATE<br><br>The default is ((ESSI0_DC-1) << 8)                                                                                                               |
| ESSI0_PRESCALE_MODULUS           | Bits 0 to 7 of ESSI0_RXTX_CONTROL_INITIAL_STATE<br><br>Default = (ESSI0_PM-1)                                                                                                                            |
| ESSI0_RXTX_CONTROL_INITIAL_STATE | Initial state of the ESSI Rx and Tx Control Registers for ESSI0<br>ESSI0_PRESCALER<br>ESSI0_WORD_LENGTH<br>ESSI0_FRAME_RATE_DIVIDER<br>ESSI0_PRESCALE_MODULUS                                            |
| ESSI0_RIE                        | Bit 15 of ESSI0_CONTROL2_INITIAL_STATE<br>Valid values are:<br>ESSI_RX_INTERRUPT_DISABLE<br>ESSI_RX_INTERRUPT_ENABLE<br><br>Default = ESSI_RX_INTERRUPT_ENABLE                                           |

Table 5-16. ESSI Configuration Items for *appconfig.h* (Continued)

| ESSI #define | Explanation and Default Value                                                                                                                                                |
|--------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ESSI0_TIE    | Bit 14 of ESSI0_CONTROL2_INITIAL_STATE<br>Valid values are:<br>ESSI_TX_INTERRUPT_DISABLE<br>ESSI_TX_INTERRUPT_ENABLE<br><br>Default = ESSI_TX_INTERRUPT_DISABLE              |
| ESSI0_RE     | Bit 13 of ESSI0_CONTROL2_INITIAL_STATE<br>Valid values are:<br>ESSI_RX_DISABLE<br>ESSI_RX_ENABLE<br><br>Default = ESSI_RX_DISABLE                                            |
| ESSI0_TE0    | Transmit Enable for Transmitter 0<br>Bit 12 of ESSI0_CONTROL2_INITIAL_STATE<br>Valid values are:<br>ESSI_TX0_DISABLE<br>ESSI_TX0_ENABLE<br><br>Default = ESSI_TX0_DISABLE    |
| ESSI0_TE1    | Transmitter Enable for Transmitter 1<br>Bit 11 of ESSI0_CONTROL2_INITIAL_STATE<br>Valid values are:<br>ESSI_TX1_DISABLE<br>ESSI_TX1_ENABLE<br><br>Default = ESSI_TX1_DISABLE |
| ESSI0_TE2    | Transmitter Enable for Transmitter 2<br>Bit 10 of ESSI0_CONTROL2_INITIAL_STATE<br>Valid values are:<br>ESSI_TX2_DISABLE<br>ESSI_TX2_ENABLE<br><br>Default = ESSI_TX2_DISABLE |
| ESSI0_SYN    | Bit 7 of ESSI0_CONTROL2_INITIAL_STATE<br>Valid values are:<br>ESSI_SYNC_MODE_DISABLED<br>ESSI_SYNC_MODE_ENABLED<br><br>The default is ESSI_SYNC_MODE_ENABLED                 |
| ESSI0_TSHFD  | Bit 6 of ESSI0_CONTROL2_INITIAL_STATE<br>Valid values are:<br>ESSI_TX_SHIFT_DIR_LSB_FIRST<br>ESSI_TX_SHIFT_DIR_MSB_FIRST<br><br>Default = ESSI_TX_SHIFT_DIR_MSB_FIRST        |



Table 5-16. ESSI Configuration Items for *appconfig.h* (Continued)

| ESSI #define                 | Explanation and Default Value                                                                                                                                                                                                   |
|------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ESSI0_TSCKP                  | Bit 5 of ESSI0_CONTROL2_INITIAL_STATE<br>Valid values are:<br>ESSI_TX_CLK_POL_FALLING_EDGE<br>ESSI_TX_CLK_POL_RISING_EDGE<br><br>Default = ESSI_TX_CLK_POL_RISING_EDGE                                                          |
| ESSI0_SSIEN                  | Bit 4 of ESSI0_CONTROL2_INITIAL_STATE<br>Valid values are:<br>ESSI_DISABLE<br>ESSI_ENABLE<br><br>Default = ESSI_ENABLE                                                                                                          |
| ESSI0_NET                    | Bit 3 of ESSI0_CONTROL2_INITIAL_STATE<br>Valid values are:<br>ESSI_NET_MODE_DISABLED<br>ESSI_NET_MODE_ENABLED<br><br>Default = ESSI_NET_MODE_ENABLED                                                                            |
| ESSI0_TFSI                   | Bit 2 of ESSI0_CONTROL2_INITIAL_STATE<br>Valid values are:<br>ESSI_TX_FRAME_SYNC_ACTIVE_LOW<br>ESSI_TX_FRAME_SYNC_ACTIVE_HI<br><br>Default = ESSI_TX_FRAME_SYNC_ACTIVE_HI                                                       |
| ESSI0_TFSL                   | Bit 1 of ESSI0_CONTROL2_INITIAL_STATE<br>Valid values are:<br>ESSI_TX_ONE_BIT_FRAME_SYNC<br>ESSI_TX_ONE_WORD_FRAME_SYNC<br><br>Default = ESSI_TX_ONE_BIT_FRAME_SYNC                                                             |
| ESSI0_TEFS                   | Bit 0 of ESSI0_CONTROL2_INITIAL_STATE<br>Valid values are:<br>ESSI_TX_FRAME_SYNC_EARLY<br>ESSI_TX_FRAME_SYNC_FIRST_BIT<br><br>Default = ESSI_TX_FRAME_SYNC_EARLY                                                                |
| ESSI0_CONTROL2_INITIAL_STATE | Initial state of the ESSI Control 2 Register<br>(ESSI0_RIE   ESSI0_TIE   ESSI0_RE   ESSI0_TE0   ESSI0_TE1   ESSI0_TE2   ESSI0_SYN   ESSI0_TSHFD   ESSI0_TSCKP   ESSI0_SSIEN   ESSI0_NET   ESSI0_TFSI   ESSI0_TFSL   ESSI0_TEFS) |
| ESSI0_DIV4DIS                | Bit 15 of ESSI0_CONTROL3_INITIAL_STATE<br>Valid values are:<br>ESSI_DIVIDE_MCU_CLOCK_1<br>ESSI_DIVIDE_MCU_CLOCK_4<br><br>Default = ESSI_DIVIDE_MCU_CLOCK_1                                                                      |

Table 5-16. ESSI Configuration Items for *appconfig.h* (Continued)

| ESSI #define | Explanation and Default Value                                                                                                                                              |
|--------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ESSI0_RSHFD  | Bit 14 of ESSI0_CONTROL3_INITIAL_STATE<br>Valid values are:<br>ESSI_RX_SHIFT_DIR_MSB_FIRST<br>ESSI_RX_SHIFT_DIR_LSB_FIRST<br><br>Default = ESSI_RX_SHIFT_DIR_MSB_FIRST     |
| ESSI0_RSCKP  | Bit 13 of ESSI0_CONTROL3_INITIAL_STATE<br>Valid values are:<br>ESSI_RX_CLK_POL_FALLING_EDGE<br>ESSI_RX_CLK_POL_RISING_EDGE<br><br>Default = ESSI_RX_CLK_POL_FALLING_EDGE   |
| ESSI0_RDMAE  | Bit 12 of ESSI0_CONTROL3_INITIAL_STATE<br>Valid values are:<br>ESSI_RX_DMA_REQUEST_DISABLED<br>ESSI_RX_DMA_REQUEST_ENABLED<br><br>Default = ESSI_RX_DMA_REQUEST_DISABLED   |
| ESSI0_TDMAE  | Bit 11 of ESSI0_CONTROL3_INITIAL_STATE<br>Valid values are:<br>ESSI_TX_DMA_REQUEST_DISABLED<br>ESSI_TX_DMA_REQUEST_ENABLED<br><br>Default = ESSI_TX_DMA_REQUEST_DISABLED   |
| ESSI0_RFSI   | Bit 10 of ESSI0_CONTROL3_INITIAL_STATE<br>Valid values are:<br>ESSI_RX_FRAME_SYNC_ACTIVE_HI<br>ESSI_RX_FRAME_SYNC_ACTIVE_LOW<br><br>Default = ESSI_RX_FRAME_SYNC_ACTIVE_HI |
| ESSI0_RFSL   | Bit 9 of ESSI0_CONTROL3_INITIAL_STATE<br>Valid values are:<br>ESSI_RX_ONE_WORD_FRAME_SYNC<br>ESSI_RX_ONE_BIT_FRAME_SYNC<br><br>Default = ESSI_RX_ONE_BIT_FRAME_SYNC        |
| ESSI0_REFS   | Bit 8 of ESSI0_CONTROL3_INITIAL_STATE<br>Valid values are:<br>ESSI_RX_FRAME_SYNC_EARLY<br>ESSI_RX_FRAME_SYNC_FIRST_BIT<br><br>Default = ESSI_RX_FRAME_SYNC_EARLY           |
| ESSI0_RFEN   | Bit 7 of ESSI0_CONTROL3_INITIAL_STATE<br>Valid values are:<br>ESSI_RX_FIFO_DISABLE<br>ESSI_RX_FIFO_ENABLE<br><br>Default = ESSI_RX_FIFO_ENABLE                             |

Table 5-16. ESSI Configuration Items for *appconfig.h* (Continued)

| ESSI #define                 | Explanation and Default Value                                                                                                                                                                                                        |
|------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ESSI0_TFEN                   | Bit 6 of ESSI0_CONTROL3_INITIAL_STATE<br>Valid values are:<br>ESSI_TX_FIFO_DISABLE<br>ESSI_TX_FIFO_ENABLE<br><br>Default = ESSI_TX_FIFO_ENABLE                                                                                       |
| ESSI0_RLIE                   | Bit 1 of ESSI0_CONTROL3_INITIAL_STATE<br>Valid values are:<br>ESSI_RX_LAST_SLOT_INT_DISABLE<br>ESSI_RX_LAST_SLOT_INT_ENABLE<br><br>Default = ESSI_RX_LAST_SLOT_INT_DISABLE                                                           |
| ESSI0_TLIE                   | Bit 0 of ESSI0_CONTROL3_INITIAL_STATE<br>Valid values are:<br>ESSI_TX_LAST_SLOT_INT_DISABLE<br>ESSI_TX_LAST_SLOT_INT_ENABLE<br><br>Default = ESSI_TX_LAST_SLOT_INT_DISABLE                                                           |
| ESSI0_CONTROL3_INITIAL_STATE | Initial state of the ESSI Control3 Register for ESSI0<br>(ESSI0_DIV4DIS   ESSI0_RSHFD   ESSI0_RSCKP  <br>ESSI0_RDMAE   ESSI0_TDMAE   ESSI0_RFSI   ESSI0_RFSL  <br>ESSI0_REFS   ESSI0_RFEN   ESSI0_TFEN   ESSI0_RLIE  <br>ESSI0_TLIE) |
| ESSI0_TXSF0                  | Bit 10 of SSI_CONTROL4_INITIAL_STATE<br>Valid values are:<br>ESSI_TX_STATUS_CTL_0_ON<br>ESSI_TX_STATUS_CTL_0_OFF<br><br>Default = ESSI_TX_STATUS_CTL_0_ON                                                                            |
| ESSI0_TXSF1                  | Bit 9 of SSI_CONTROL4_INITIAL_STATE<br>Valid values are:<br>ESSI_TX_STATUS_CTL_1_ON<br>ESSI_TX_STATUS_CTL_1_OFF<br><br>Default = ESSI_TX_STATUS_CTL_1_OFF                                                                            |
| ESSI0_TXSF2                  | Bit 8 of SSI_CONTROL4_INITIAL_STATE<br>Valid values are:<br>ESSI_TX_STATUS_CTL_2_ON<br>ESSI_TX_STATUS_CTL_2_OFF<br><br>Default = ESSI_TX_STATUS_CTL_2_OFF                                                                            |
| ESSI0_SSC1                   | Bit 6 of SSI_CONTROL4_INITIAL_STATE<br>Valid values are:<br>ESSI_SC1_IS_SERIAL_IO_FLAG<br>ESSI_SC1_IS_TX0_DRV_ENABLE<br><br>Default = ESSI_SC1_IS_SERIAL_IO_FLAG                                                                     |

Table 5-16. ESSI Configuration Items for *appconfig.h* (Continued)

| ESSI #define                     | Explanation and Default Value                                                                                                                                                     |
|----------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ESSI0_TXDIR                      | Bit 5 of SSI_CONTROL4_INITIAL_STATE<br>Valid values are:<br>ESSI_TX_CLK_INTERNAL<br>ESSI_TX_CLK_EXTERNAL<br><br>Default = ESSI_TX_CLK_EXTERNAL                                    |
| ESSI0_TXFSDIR                    | Bit 4 of SSI_CONTROL4_INITIAL_STATE<br>Valid values are:<br>ESSI_TX_FRM_SYNC_INTERNAL<br>ESSI_TX_FRM_SYNC_EXTERNAL<br><br>Default = ESSI_TX_FRM_SYNC_EXTERNAL                     |
| ESSI0_RXFSDIR                    | Bit 3 of SSI_CONTROL4_INITIAL_STATE<br>Valid values are:<br>ESSI_RX_FRM_SYNC_INTERNAL<br>ESSI_RX_FRM_SYNC_EXTERNAL<br><br>Default = ESSI_RX_FRM_SYNC_EXTERNAL                     |
| ESSI0_RXDIR                      | Bit 2 of SSI_CONTROL4_INITIAL_STATE<br>Valid values are:<br>ESSI_RX_CLK_INTERNAL<br>ESSI_RX_CLK_EXTERNAL<br><br>Default = ESSI_RX_CLK_EXTERNAL                                    |
| ESSI0_CONTROL4_INITIAL_STATE     | Initial state of the ESSI Control4 Register for ESSI0<br>(ESSI0_TXSF0   ESSI0_TXSF1   ESSI0_TXSF2   ESSI0_SSC1  <br>ESSI0_TXDIR   ESSI0_TXFSDIR   ESSI0_RXFSDIR  <br>ESSI0_RXDIR) |
| ESSI0_RFWM                       | Actual bits in the FIFO Control Status Register defining the Rx<br>FIFO Full Watermark for ESSI0.<br><br>Default = ESSI0_RX_FIFO_FULL_WM<<4                                       |
| ESSI0_TFWM                       | Actual bits in the FIFO Control Status Register defining the Tx<br>FIFO Empty Watermark for ESSI0.<br><br>Default = ESSI0_TX_FIFO_EMPTY_WM                                        |
| ESSI0_FIFO_CONTROL_INITIAL_STATE | Initial state of the ESSI FIFO Control Register for ESSI0<br>(ESSI0_RFWM   ESSI0_TFWM)                                                                                            |
| ESSI0_TX_SLOT_MASK_REG_A         | Default = 0xFFFFu                                                                                                                                                                 |
| ESSI0_TX_SLOT_MASK_REG_B         | Default = 0xFFFFu                                                                                                                                                                 |
| ESSI0_RX_SLOT_MASK_REG_A         | Default = 0xFFFFu                                                                                                                                                                 |
| ESSI0_RX_SLOT_MASK_REG_B         | Default = 0xFFFFu                                                                                                                                                                 |

Table 5-16. ESSI Configuration Items for *appconfig.h* (Continued)

| ESSI #define            | Explanation and Default Value                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ESSI1_FIFO_SIZE         | <p>The default size (number of 16-bit samples) for both the RX FIFO and TX FIFO buffers used internally by the driver for ESSI1.</p> <p>For more information, refer to ESSI1_RX_FIFO_SIZE and ESSI1_TX_FIFO_SIZE</p> <p>Default = 32</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| ESSI1_RX_FIFO_SIZE      | <p>The size (number of 16-bit samples) of the RX FIFO buffer used internally by the driver for the RX channel of ESSI1. The length of the RX FIFO buffer determines the maximum number of ESSI input samples that can be buffered within the driver.</p> <p>Default = ESSI1_FIFO_SIZE</p>                                                                                                                                                                                                                                                                                                                                                                                                             |
| ESSI1_TX_FIFO_SIZE      | <p>The size (number of 16-bit samples) of the TX FIFO buffer used internally by the driver for the TX channel of ESSI1. The length of the TX FIFO buffer determines the maximum number of ESSI output samples that can be buffered within the driver.</p> <p>Default = ESSI1_FIFO_SIZE</p>                                                                                                                                                                                                                                                                                                                                                                                                            |
| ESSI1_RX_CALLBACK_LEVEL | <p>The number of 16-bit samples that the ESSI must receive before calling the user function specified by ESSI1_RX_CALLBACK. The value specified must be less than ESSI1_RX_FIFO_SIZE.</p> <p>Increasing the ESSI1_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.</p> <p>Default = 8</p>                                                                                                                 |
| ESSI1_RX_CALLBACK       | <p>A user function for the ESSI1 driver to call when it has received at least ESSI1_RX_CALLBACK_LEVEL samples in its internal RX buffer. The user function must be of type <i>essi_tCallback</i>:</p> <pre>typedef void (*essi_tCallback) (void *pCallbackArg, int MaxNBytes);</pre> <p>The parameters to the callback function are:</p> <p><i>pCallbackArg</i> - The original value specified by the user for ESSI1_RX_CALLBACK_ARG</p> <p><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</p> <p>Default = NULL</p> |

Table 5-16. ESSI Configuration Items for *appconfig.h* (Continued)

| ESSI #define            | Explanation and Default Value                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|-------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ESSI1_RX_CALLBACK_ARG   | <p>A value that the user would like passed to the ESSI1_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.</p> <p>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 ESSI1_RX_CALLBACK function.</p> <p>Default = NULL</p>                                                                                                                                         |
| ESSI1_TX_CALLBACK_LEVEL | <p>The minimum number of 16-bit samples that the ESSI1 must maintain in the transmit FIFO buffer. If the number of samples falls below ESSI1_TX_CALLBACK_LEVEL, the driver calls the user function specified by ESSI1_TX_CALLBACK. The value specified must be less than ESSI1_TX_FIFO_SIZE.</p> <p>Increasing the ESSI1_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.</p> <p>Default = 8</p> |
| ESSI1_TX_CALLBACK       | <p>A user function for the ESSI1 driver to call when the TX buffer has fallen below ESSI1_TX_CALLBACK_LEVEL samples. The user function must be of type <i>essi_tCallback</i>:</p> <pre>typedef void (*ssi_tCallback) (void *pCallbackArg, int MaxNBytes);</pre> <p>The parameters to the callback function are:</p> <p><i>pCallbackArg</i> - The original value specified by the user for ESSI1_TX_CALLBACK_ARG</p> <p><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</p> <p>Default = NULL</p>                                          |

Table 5-16. ESSI Configuration Items for *appconfig.h* (Continued)

| ESSI #define           | Explanation and Default Value                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ESSI1_TX_CALLBACK_ARG  | <p>A value that the user would like passed to the ESSI1_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.</p> <p>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 ESSI1_TX_CALLBACK function.</p> <p>Default = NULL.</p> |
| ESSI1_NUM_TIME_SLOTS   | <p>The number of time slots to be used when ESSI1 is configured for network mode.</p> <p>Default = 2</p>                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| ESSI1_HW_FIFO_DEPTH    | <p>The ESSI1 contains a Receive and Transmit hardware FIFO that can buffer up to 8 samples. This value defines how many samples the ESSI1 ISR will process.</p> <p>Default = 8</p>                                                                                                                                                                                                                                                                                                                                                                    |
| ESSI1_RX_FIFO_FULL_WM  | <p>When the ESSI1 RX FIFO contains this many samples, the ESSI1 Rx ISR will be called, if enabled.</p> <p>Default = 8</p>                                                                                                                                                                                                                                                                                                                                                                                                                             |
| ESSI1_TX_FIFO_EMPTY_WM | <p>When the ESSI1 TX FIFO contains this many empty samples, the ESSI1 Tx ISR will be called, if enabled.</p> <p>Default = 8</p>                                                                                                                                                                                                                                                                                                                                                                                                                       |
| ESSI1_PRESCALER        | <p>Bit 15 of ESSI1_RXTX_CONTROL_INITIAL_STATE<br/>Valid values are:<br/>ESSI1_PRESCALER_1<br/>ESSI1_PRESCALER_8</p> <p>Default = ESSI1_PRESCALER_1</p>                                                                                                                                                                                                                                                                                                                                                                                                |
| ESSI1_WORD_LENGTH      | <p>Bits 13 and 14 of ESSI1_RXTX_CONTROL_INITIAL_STATE<br/>Valid values are:<br/>ESSI1_WORD_LENGTH_8<br/>ESSI1_WORD_LENGTH_10<br/>ESSI1_WORD_LENGTH_12<br/>ESSI1_WORD_LENGTH_16</p> <p>Default = ESSI1_WORD_LENGTH_16</p>                                                                                                                                                                                                                                                                                                                              |
| ESSI1_DC               | <p>Used by ESSI1_FRAME_RATE_DIVIDER<br/>Valid values are:<br/>1 to 32</p> <p>Default = ESSI1_NUM_TIME_SLOTS</p>                                                                                                                                                                                                                                                                                                                                                                                                                                       |

Table 5-16. ESSI Configuration Items for *appconfig.h* (Continued)

| ESSI #define                     | Explanation and Default Value                                                                                                                                                |
|----------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ESSI1_PM                         | Used by ESSI1_PRESCALE_MODULUS<br>Valid values are:<br>1 to 256<br><br>Default = 58 (48K Sample Rate)                                                                        |
| ESSI1_FRAME_RATE_DIVIDER         | Bits 8 to 12 of ESSI1_RXTX_CONTROL_INITIAL_STATE<br><br>Default = ((ESSI1_DC-1) << 8)                                                                                        |
| ESSI1_PRESCALE_MODULUS           | Bits 0 to 7 of ESSI1_RXTX_CONTROL_INITIAL_STATE<br><br>Default = (ESSI1_PM-1)                                                                                                |
| ESSI1_RXTX_CONTROL_INITIAL_STATE | Initial state of the ESSI Rx and Tx Control Registers for ESSI1<br>ESSI1_PRESCALER<br>ESSI1_WORD_LENGTH<br>ESSI1_FRAME_RATE_DIVIDER<br>ESSI1_PRESCALE_MODULUS                |
| ESSI1_RIE                        | Bit 15 of ESSI1_CONTROL2_INITIAL_STATE<br>Valid values are:<br>ESSI_RX_INTERRUPT_DISABLE<br>ESSI_RX_INTERRUPT_ENABLE<br><br>Default = ESSI_RX_INTERRUPT_ENABLE               |
| ESSI1_TIE                        | Bit 14 of ESSI1_CONTROL2_INITIAL_STATE<br>Valid values are:<br>ESSI_TX_INTERRUPT_DISABLE<br>ESSI_TX_INTERRUPT_ENABLE<br><br>Default = ESSI_TX_INTERRUPT_DISABLE              |
| ESSI1_RE                         | Bit 13 of ESSI1_CONTROL2_INITIAL_STATE<br>Valid values are:<br>ESSI_RX_DISABLE<br>ESSI_RX_ENABLE<br><br>Default = ESSI_RX_DISABLE                                            |
| ESSI1_TE0                        | Transmit Enable for Transmitter 0<br>Bit 12 of ESSI1_CONTROL2_INITIAL_STATE<br>Valid values are:<br>ESSI_TX0_DISABLE<br>ESSI_TX0_ENABLE<br><br>Default = ESSI_TX0_DISABLE    |
| ESSI1_TE1                        | Transmitter Enable for Transmitter 1<br>Bit 11 of ESSI1_CONTROL2_INITIAL_STATE<br>Valid values are:<br>ESSI_TX1_DISABLE<br>ESSI_TX1_ENABLE<br><br>Default = ESSI_TX1_DISABLE |



Table 5-16. ESSI Configuration Items for *appconfig.h* (Continued)

| ESSI #define | Explanation and Default Value                                                                                                                                                |
|--------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ESSI1_TE2    | Transmitter Enable for Transmitter 2<br>Bit 10 of ESSI1_CONTROL2_INITIAL_STATE<br>Valid values are:<br>ESSI_TX2_DISABLE<br>ESSI_TX2_ENABLE<br><br>Default = ESSI_TX2_DISABLE |
| ESSI1_SYN    | Bit 7 of ESSI1_CONTROL2_INITIAL_STATE<br>Valid values are:<br>ESSI_SYNC_MODE_DISABLED<br>ESSI_SYNC_MODE_ENABLED<br><br>Default = ESSI_SYNC_MODE_ENABLED                      |
| ESSI1_TSHFD  | Bit 6 of ESSI1_CONTROL2_INITIAL_STATE<br>Valid values are:<br>ESSI_TX_SHIFT_DIR_LSB_FIRST<br>ESSI_TX_SHIFT_DIR_MSB_FIRST<br><br>Default = ESSI_TX_SHIFT_DIR_MSB_FIRST        |
| ESSI1_TSCKP  | Bit 5 of ESSI1_CONTROL2_INITIAL_STATE<br>Valid values are:<br>ESSI_TX_CLK_POL_FALLING_EDGE<br>ESSI_TX_CLK_POL_RISING_EDGE<br><br>Default = ESSI_TX_CLK_POL_RISING_EDGE       |
| ESSI1_SSIEN  | Bit 4 of ESSI1_CONTROL2_INITIAL_STATE<br>Valid values are:<br>ESSI_DISABLE<br>ESSI_ENABLE<br><br>Default = ESSI_ENABLE                                                       |
| ESSI1_NET    | Bit 3 of ESSI1_CONTROL2_INITIAL_STATE<br>Valid values are:<br>ESSI_NET_MODE_DISABLED<br>ESSI_NET_MODE_ENABLED<br><br>Default = ESSI_NET_MODE_ENABLED                         |
| ESSI1_TFSI   | Bit 2 of ESSI1_CONTROL2_INITIAL_STATE<br>Valid values are:<br>ESSI_TX_FRAME_SYNC_ACTIVE_LOW<br>ESSI_TX_FRAME_SYNC_ACTIVE_HI<br><br>Default = ESSI_TX_FRAME_SYNC_ACTIVE_HI    |

Table 5-16. ESSI Configuration Items for *appconfig.h* (Continued)

| ESSI #define                 | Explanation and Default Value                                                                                                                                                                                                   |
|------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ESSI1_TFSL                   | Bit 1 of ESSI1_CONTROL2_INITIAL_STATE<br>Valid values are:<br>ESSI_TX_ONE_BIT_FRAME_SYNC<br>ESSI_TX_ONE_WORD_FRAME_SYNC<br><br>Default = ESSI_TX_ONE_BIT_FRAME_SYNC                                                             |
| ESSI1_TEFS                   | Bit 0 of ESSI1_CONTROL2_INITIAL_STATE<br>Valid values are:<br>ESSI_TX_FRAME_SYNC_EARLY<br>ESSI_TX_FRAME_SYNC_FIRST_BIT<br><br>Default = ESSI_TX_FRAME_SYNC_EARLY                                                                |
| ESSI1_CONTROL2_INITIAL_STATE | Initial state of the ESSI Control 2 Register<br>(ESSI1_RIE   ESSI1_TIE   ESSI1_RE   ESSI1_TE0   ESSI1_TE1   ESSI1_TE2   ESSI1_SYN   ESSI1_TSHFD   ESSI1_TSCKP   ESSI1_SSIEN   ESSI1_NET   ESSI1_TFSI   ESSI1_TFSL   ESSI1_TEFS) |
| ESSI1_DIV4DIS                | Bit 15 of ESSI1_CONTROL3_INITIAL_STATE<br>Valid values are:<br>ESSI_DIVIDE_MCU_CLOCK_1<br>ESSI_DIVIDE_MCU_CLOCK_4<br><br>Default = ESSI_DIVIDE_MCU_CLOCK_1                                                                      |
| ESSI1_RSHFD                  | Bit 14 of ESSI1_CONTROL3_INITIAL_STATE<br>Valid values are:<br>ESSI_RX_SHIFT_DIR_MSB_FIRST<br>ESSI_RX_SHIFT_DIR_LSB_FIRST<br><br>Default = ESSI_RX_SHIFT_DIR_MSB_FIRST                                                          |
| ESSI1_RSCKP                  | Bit 13 of ESSI1_CONTROL3_INITIAL_STATE<br>Valid values are:<br>ESSI_RX_CLK_POL_FALLING_EDGE<br>ESSI_RX_CLK_POL_RISING_EDGE<br><br>Default = ESSI_RX_CLK_POL_FALLING_EDGE                                                        |
| ESSI1_RDMAE                  | Bit 12 of ESSI1_CONTROL3_INITIAL_STATE<br>Valid values are:<br>ESSI_RX_DMA_REQUEST_DISABLED<br>ESSI_RX_DMA_REQUEST_ENABLED<br><br>Default = ESSI_RX_DMA_REQUEST_DISABLED                                                        |
| ESSI1_TDMAE                  | Bit 11 of ESSI1_CONTROL3_INITIAL_STATE<br>Valid values are:<br>ESSI_TX_DMA_REQUEST_DISABLED<br>ESSI_TX_DMA_REQUEST_ENABLED<br><br>Default = ESSI_TX_DMA_REQUEST_DISABLED                                                        |

Table 5-16. ESSI Configuration Items for *appconfig.h* (Continued)

| ESSI #define                 | Explanation and Default Value                                                                                                                                                                                                        |
|------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ESSI1_RFSI                   | Bit 10 of ESSI1_CONTROL3_INITIAL_STATE<br>Valid values are:<br>ESSI_RX_FRAME_SYNC_ACTIVE_HI<br>ESSI_RX_FRAME_SYNC_ACTIVE_LOW<br><br>Default = ESSI_RX_FRAME_SYNC_ACTIVE_HI                                                           |
| ESSI1_RFSL                   | Bit 9 of ESSI1_CONTROL3_INITIAL_STATE<br>Valid values are:<br>ESSI_RX_ONE_WORD_FRAME_SYNC<br>ESSI_RX_ONE_BIT_FRAME_SYNC<br><br>Default = ESSI_RX_ONE_BIT_FRAME_SYNC                                                                  |
| ESSI1_REFS                   | Bit 8 of ESSI1_CONTROL3_INITIAL_STATE<br>Valid values are:<br>ESSI_RX_FRAME_SYNC_EARLY<br>ESSI_RX_FRAME_SYNC_FIRST_BIT<br><br>Default = ESSI_RX_FRAME_SYNC_EARLY                                                                     |
| ESSI1_RFEN                   | Bit 7 of ESSI1_CONTROL3_INITIAL_STATE<br>Valid values are:<br>ESSI_RX_FIFO_DISABLE<br>ESSI_RX_FIFO_ENABLE<br><br>Default = ESSI_RX_FIFO_ENABLE                                                                                       |
| ESSI1_TFEN                   | Bit 6 of ESSI1_CONTROL3_INITIAL_STATE<br>Valid values are:<br>ESSI_TX_FIFO_DISABLE<br>ESSI_TX_FIFO_ENABLE<br><br>Default = ESSI_TX_FIFO_ENABLE                                                                                       |
| ESSI1_RLIE                   | Bit 1 of ESSI1_CONTROL3_INITIAL_STATE<br>Valid values are:<br>ESSI_RX_LAST_SLOT_INT_DISABLE<br>ESSI_RX_LAST_SLOT_INT_ENABLE<br><br>Default = ESSI_RX_LAST_SLOT_INT_DISABLE                                                           |
| ESSI1_TLIE                   | Bit 0 of ESSI1_CONTROL3_INITIAL_STATE<br>Valid values are:<br>ESSI_TX_LAST_SLOT_INT_DISABLE<br>ESSI_TX_LAST_SLOT_INT_ENABLE<br><br>Default = ESSI_TX_LAST_SLOT_INT_DISABLE                                                           |
| ESSI1_CONTROL3_INITIAL_STATE | Initial state of the ESSI Control3 Register for ESSI1<br>(ESSI1_DIV4DIS   ESSI1_RSHFD   ESSI1_RSCKP  <br>ESSI1_RDMAE   ESSI1_TDMAE   ESSI1_RFSI   ESSI1_RFSL  <br>ESSI1_REFS   ESSI1_RFEN   ESSI1_TFEN   ESSI1_RLIE  <br>ESSI1_TLIE) |

Table 5-16. ESSI Configuration Items for *appconfig.h* (Continued)

| ESSI #define  | Explanation and Default Value                                                                                                                                      |
|---------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ESSI1_TXSF0   | Bit 10 of ESSI1_CONTROL4_INITIAL_STATE<br>Valid values are:<br>ESSI_TX_STATUS_CTL_0_ON<br>ESSI_TX_STATUS_CTL_0_OFF<br><br>Default = ESSI_TX_STATUS_CTL_0_ON        |
| ESSI1_TXSF1   | Bit 9 of ESSI1_CONTROL4_INITIAL_STATE<br>Valid values are:<br>ESSI_TX_STATUS_CTL_1_ON<br>ESSI_TX_STATUS_CTL_1_OFF<br><br>Default = ESSI_TX_STATUS_CTL_1_OFF        |
| ESSI1_TXSF2   | Bit 8 of ESSI1_CONTROL4_INITIAL_STATE<br>Valid values are:<br>ESSI_TX_STATUS_CTL_2_ON<br>ESSI_TX_STATUS_CTL_2_OFF<br><br>Default = ESSI_TX_STATUS_CTL_2_OFF        |
| ESSI1_SSC1    | Bit 6 of ESSI1_CONTROL4_INITIAL_STATE<br>Valid values are:<br>ESSI_SC1_IS_SERIAL_IO_FLAG<br>ESSI_SC1_IS_TX0_DRV_ENABLE<br><br>Default = ESSI_SC1_IS_SERIAL_IO_FLAG |
| ESSI1_TXDIR   | Bit 5 of ESSI1_CONTROL4_INITIAL_STATE<br>Valid values are:<br>ESSI_TX_CLK_INTERNAL<br>ESSI_TX_CLK_EXTERNAL<br><br>Default = ESSI_TX_CLK_EXTERNAL                   |
| ESSI1_TXFSDIR | Bit 4 of ESSI1_CONTROL4_INITIAL_STATE<br>Valid values are:<br>ESSI_TX_FRM_SYNC_INTERNAL<br>ESSI_TX_FRM_SYNC_EXTERNAL<br><br>Default = ESSI_TX_FRM_SYNC_EXTERNAL    |
| ESSI1_RXFSDIR | Bit 3 of ESSI1_CONTROL4_INITIAL_STATE<br>Valid values are:<br>ESSI_RX_FRM_SYNC_INTERNAL<br>ESSI_RX_FRM_SYNC_EXTERNAL<br><br>Default = ESSI_RX_FRM_SYNC_EXTERNAL    |
| ESSI1_RXDIR   | Bit 2 of ESSI1_CONTROL4_INITIAL_STATE<br>Valid values are:<br>ESSI_RX_CLK_INTERNAL<br>ESSI_RX_CLK_EXTERNAL<br><br>Default = ESSI_RX_CLK_EXTERNAL                   |

**Table 5-16. ESSI Configuration Items for *appconfig.h* (Continued)**

| ESSI #define                     | Explanation and Default Value                                                                                                                                                    |
|----------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ESSI1_CONTROL4_INITIAL_STATE     | Initial state of the ESSI Control4 Register for ESSI1<br>ESSI1_TXSF0<br>ESSI1_TXSF1<br>ESSI1_TXSF2<br>ESSI1_SSC1<br>ESSI1_TXDIR<br>ESSI1_TXFSDIR<br>ESSI1_RXFSDIR<br>ESSI1_RXDIR |
| ESSI1_RFWM                       | Actual bits in the FIFO Control Status Register defining the Rx FIFO Full Watermark for ESSI1.<br><br>Default = ESSI1_RX_FIFO_FULL_WM<<4                                         |
| ESSI1_TFWM                       | Actual bits in the FIFO Control Status Register defining the Tx FIFO Empty Watermark for ESSI1.<br><br>Default = ESSI1_TX_FIFO_EMPTY_WM                                          |
| ESSI1_FIFO_CONTROL_INITIAL_STATE | Initial state of the ESSI FIFO Control Register for ESSI1<br>ESSI1_RFWM<br>ESSI1_TFWM                                                                                            |
| ESSI1_TX_SLOT_MASK_REG_A         | Default = 0xFFFFu                                                                                                                                                                |
| ESSI1_TX_SLOT_MASK_REG_B         | Default = 0xFFFFu                                                                                                                                                                |
| ESSI1_RX_SLOT_MASK_REG_A         | Default = 0xFFFFu                                                                                                                                                                |
| ESSI1_RX_SLOT_MASK_REG_B         | Default = 0xFFFFu                                                                                                                                                                |

### 5.3.3 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 ESSI driver in an SDK Embedded Project, insert the following line in the *appconfig.h* file:

```
#define INCLUDE_ESSI
```

The following information may be found in the public header file *essi.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);
*****/
/*****

```



ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

```
* types_tHandle essiOpen(const char *pName, int OFlags);
* ssize_t essiWrite(types_tHandle FileDesc, const void * pBuffer, size_t NBytes);
* ssize_t essiRead(types_tHandle FileDesc, void * pBuffer, size_t NBytes);
* int essiClose(types_tHandle FileDesc);
* UWord16 essiIoctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams,
* const char * pName);
*****/
```

**Public Data Structure(s):**

None

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



### 5.3.4 ESSI 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.

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.

### 5.3.4.1 *open*

#### Call(s):

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

#### Arguments:

**Table 5-17. ESSI Driver Arguments - *open***

|                 |            |                                                                                                  |
|-----------------|------------|--------------------------------------------------------------------------------------------------|
| <i>pName</i>    | <i>in</i>  | The ESSI device name; see <a href="#">Table 5-18</a>                                             |
| <i>OFlag</i>    | <i>in</i>  | Open Mode Flags; specify O_RDWR<br>Use O_NONBLOCK flags for Non-Blocking mode                    |
| <i>FileDesc</i> | <i>out</i> | ESSI device descriptor returned by <i>open</i> call and used in <i>read/write</i> function calls |

**Table 5-18. ESSI Device Names for DSP5685x Chips**

| Chip                                                     | Can Use                                          |
|----------------------------------------------------------|--------------------------------------------------|
| DSP56853<br>DSP56854<br>DSP56855<br>DSP56857<br>DSP56858 | BSP_DEVICE_NAME_ESSI_0<br>BSP_DEVICE_NAME_ESSI_1 |

**Description:** The *open* call enables the ESSI peripheral for operations and returns a file descriptor used by the ESSI driver's *read*, *write*, *close*, and *ioctl* function calls. The ESSI 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 ESSI device and is used by the ESSI's *read*, *write*, and *ioctl* functions. A "-1" is returned when the *open* call is not successful.

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



**5.3.4.2 write****Call(s):**

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

**Arguments:****Table 5-19. ESSI Driver Arguments - write**

|                 |           |                                                                                                           |
|-----------------|-----------|-----------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | ESSI device descriptor returned by <i>open</i> call                                                       |
| <i>pBuffer</i>  | <i>in</i> | Pointer to user buffer; the <i>write</i> 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 ESSI device.

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

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

### 5.3.4.3 *read*

**Call(s):**

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

**Arguments:**

**Table 5-20. ESSI Driver Arguments - *read***

|                 |           |                                                                                                          |
|-----------------|-----------|----------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | ESSI device descriptor returned by <i>open</i> call                                                      |
| <i>pBuffer</i>  | <i>in</i> | Pointer to user buffer; the <i>read</i> 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 ESSI device into the user buffer.

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

**Code Example:** None



### 5.3.4.4 *close*

**Call(s):**

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

**Arguments:**

**Table 5-21. ESSI Driver Arguments - *close***

|                 |           |                                                     |
|-----------------|-----------|-----------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | ESSI device descriptor returned by <i>open</i> call |
|-----------------|-----------|-----------------------------------------------------|

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

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

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

### 5.3.4.5 *ioctl*

**Call(s):**

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

**Arguments:**
**Table 5-22. ESSI Driver Arguments - *ioctl***

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

**Table 5-23. ESSI *ioctl* Commands**

| Cmd                                   | <i>pParams</i> | Return | Result                                                                                                                              |
|---------------------------------------|----------------|--------|-------------------------------------------------------------------------------------------------------------------------------------|
| ESSI_DEVICE_RESET                     | NULL           | None   | Sets new ESSI configuration and resets device; cancels all non-blocking operations                                                  |
| ESSI_DEVICE_ENABLE                    | NULL           | None   | Enables device                                                                                                                      |
| ESSI_DEVICE_DISABLE                   | NULL           | None   | Disables device                                                                                                                     |
| ESSI_DEVICE_SET_RX_CAL<br>LBACK_LEVEL | UWord16 length | None   | Set number of words/bytes for Read callback; after this number of words is received, the Receive callback is called                 |
| ESSI_DEVICE_SET_TX_CAL<br>LBACK_LEVEL | UWord16 length | None   | Sets 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 ESSI device modes or reports ESSI driver status.

**Returns:** The *ioctl* call returns a type *UWord16*. This *UWord16* is dependent on the command passed in as an input parameter. **Table 5-23** lists these returns based on the specific command.

**Code Example:** **Code Example 5-15** sets up the ESSI to write data out the serial port.

**Code Example 5-15. ESSI Driver Usage**

```

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

#include "essi.h"

void main(void)

```



```
{
 types_tHandle EssiDevice;
 UWord16 Data;

 /* open essi driver */
 EssiDevice = open(BSP_DEVICE_NAME_ESSI_0, O_RDWR);

 Data = 0x0030;

 /* Send one sample via ESSI */
 write(EssiDevice, (void *)&Data, sizeof(UWord16));

 /* close essi driver */
 close(EssiDevice);
}
/*****/
```

### 5.3.5 ESSI 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.

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.

### 5.3.5.1 *essiOpen*

**Call(s):**

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

**Arguments:**

**Table 5-24. ESSI Driver Arguments - *essiOpen***

|                 |            |                                                                                                              |
|-----------------|------------|--------------------------------------------------------------------------------------------------------------|
| <i>pName</i>    | <i>in</i>  | The ESSI device name; see <a href="#">Table 5-25</a>                                                         |
| <i>OFlag</i>    | <i>in</i>  | Open Mode Flags; specify O_RDWR<br>Use O_NONBLOCK flags for Non-Blocking mode                                |
| <i>FileDesc</i> | <i>out</i> | ESSI device descriptor returned by <i>essiOpen</i> call and used in <i>essiRead/essiWrite</i> function calls |

**Table 5-25. ESSI Device Names for DSP5685x Chips**

| Chip                                                     | Can Use                                          |
|----------------------------------------------------------|--------------------------------------------------|
| DSP56853<br>DSP56854<br>DSP56855<br>DSP56857<br>DSP56858 | BSP_DEVICE_NAME_ESSI_0<br>BSP_DEVICE_NAME_ESSI_1 |

**Description:** The *essiOpen* call enables the ESSI peripheral for operations and returns a file descriptor used by the ESSI driver's *essiRead*, *essiWrite*, *essiClose*, and *essiIoctl* function calls. The ESSI device must be opened for *essiRead/essiWrite* calls.

**Returns:** The *essiOpen* call returns a type *types\_tHandle*. When the *essiOpen* call is successful, this *types\_tHandle* is a file descriptor for the ESSI device and is used by the ESSI *essiRead*, *essiWrite*, and *essiIoctl* functions. If the *essiOpen* call is not successful, "-1" is returned.

**Code Example:** See [Code Example 5-16](#).

### 5.3.5.2 *essiWrite*

**Call(s):**

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

**Arguments:**

**Table 5-26. ESSI Driver Arguments - *essiWrite***

|                 |           |                                                                                                          |
|-----------------|-----------|----------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | ESSI device descriptor returned by <i>essiOpen</i> call                                                  |
| <i>pBuffer</i>  | <i>in</i> | Pointer to user buffer; the <i>write</i> 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 *essiWrite* call writes data from the user buffer to the ESSI device.

**Returns:** The *essiWrite* call returns a type *ssize\_t*, which is the number of transferred bytes.

**Code Example:** See [Code Example 5-16](#).





### 5.3.5.3 *essiRead*

Call(s):

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

Arguments:

Table 5-27. ESSI Driver Arguments - *essiRead*

|                 |           |                                                                                                          |
|-----------------|-----------|----------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | ESSI device descriptor returned by <i>essiOpen</i> call                                                  |
| <i>pBuffer</i>  | <i>in</i> | Pointer to user buffer; the <i>read</i> 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 *essiRead* call reads data from ESSI device into the user buffer.

**Returns:** The *essiRead* call returns a type *ssize\_t*, which is the number of received bytes.

**Code Example:** None

### 5.3.5.4 *essiClose*

**Call(s):**

```
int essiClose (types_tHandle FileDesc);
```

**Arguments:**

**Table 5-28. ESSI Driver Arguments - *essiClose***

|                 |           |                                                         |
|-----------------|-----------|---------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | ESSI device descriptor returned by <i>essiOpen</i> call |
|-----------------|-----------|---------------------------------------------------------|

**Description:** The *essiClose* call disables the ESSI peripheral.

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

**Code Example:** See [Code Example 5-16](#).

### 5.3.5.5 *essioctl*

#### Call(s):

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

#### Arguments:

**Table 5-29. ESSI Driver Arguments - *essioctl***

|                 |           |                                                                            |
|-----------------|-----------|----------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | ESSI device descriptor returned by <i>essiOpen</i> call                    |
| <i>Cmd</i>      | <i>in</i> | Commands found in <i>essi.h</i> which are used to manage the ESSI driver   |
| <i>pParams</i>  | <i>in</i> | Pointer to <i>essioctl</i> command parameter                               |
| <i>pName</i>    | <i>in</i> | The ESSI device name from <i>bsp.h</i> ; typically, BSP_DEVICE_NAME_ESSI_0 |

**Table 5-30. ESSI *essioctl* Commands**

| Cmd                               | <i>pParams</i> | Return | Result                                                                                                                              |
|-----------------------------------|----------------|--------|-------------------------------------------------------------------------------------------------------------------------------------|
| ESSI_DEVICE_RESET                 | NULL           | None   | Sets new ESSI configuration and resets device; cancels all non-blocking operations                                                  |
| ESSI_DEVICE_ENABLE                | NULL           | None   | Disables device                                                                                                                     |
| ESSI_DEVICE_DISABLE               | NULL           | None   | Enables device                                                                                                                      |
| ESSI_DEVICE_SET_RX_CALLBACK_LEVEL | UWord16 length | None   | Sets number of words/bytes for Read callback; after this number of words is received, the Receive callback is called                |
| ESSI_DEVICE_SET_TX_CALLBACK_LEVEL | UWord16 length | None   | Sets 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 *essiIoctl* call changes ESSI device modes or reports ESSI driver status.

**Returns:** The *essioctl* call returns a type *UWord16*. This *UWord16* is dependent on the command passed in as an input parameter. [Table 5-30](#) lists these returns based on the specific command.

**Code Example:** [Code Example 5-16](#) sets up the ESSI to write data out the serial port.

#### Code Example 5-16. ESSI Driver Usage

```

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

#include "essi.h"

```



```
void main(void)
{
 types_tHandleEssiDevice;
 UWord16 Data;

 /* open essi driver */
 EssiDevice = essiOpen(BSP_DEVICE_NAME_ESSI_0, 0_RDWR);

 Data = 0x0030;

 /* Send one byte via ESSI */
 essiWrite(EssiDevice, (void *)&Data, sizeof(UWord16));

 /* close essi driver */
 essiClose(EssiDevice);
}
/*****/
```



### 5.3.6 ESSI 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 ESSI Driver.

## 5.4 ESSI DMA Driver

### 5.4.1 Introduction

The DSP5685x processor has two ESSI devices, ESSI0 and ESSI1. Each ESSI device has a receiver and three transmitters. Each of these can be used with DMA. The SDK provides a driver for the ESSI that takes advantage of on-chip DMA.

### 5.4.2 Initialization

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

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

```
#define INCLUDE_ESSIDMA
```

Defining the preprocessor variable, INCLUDE\_ESSIDMA, directs the SDK to include the ESSI DMA driver in the executable image. If the preprocessor variable INCLUDE\_ESSIDMA is not defined, then the ESSI DMA driver will not be included in the executable.

To use the device-independent Application Programming Interface for the ESSI DMA Driver, insert the following line into the *appconfig.h* file:

```
#define INCLUDE_IO
#define INCLUDE_ESSIDMA
```

Defining the preprocessor variable, INCLUDE\_IO, directs the SDK to include the device-independent Application Programming Interface in the executable image. If the preprocessor variable INCLUDE\_IO is not defined, then the user application must use the device-dependent Application Programming Interface of the ESSI DMA driver.

You may also define additional preprocessor variables in the *appconfig.h* file to statically configure and initialize the ESSI DMA driver. The list of these variables is provided in [Table 5-31](#). Replace “x” with the ESSI Device being used, either 0 or 1.

The ESSI DMA Driver’s default configuration is defined in *config.h* and is shown in [Table 5-31](#). Most of the ESSI DMA defined constants in [Table 5-31](#) correspond directly to a bit or group of bits in the ESSI control registers. Please refer to the **DSP5685x 16-Bit Digital Signal Processor User’s Manual** for a detailed description of these bits.

**Table 5-31. ESSI DMA Configuration Items for *appconfig.h***

| ESSI DMA Defined Constants | Explanation and Default Value                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ESSIDMAx_RX_DMA_CHANNEL    | <p>DMA Channel to use for the ESSIx Receiver</p> <p>Valid values are:<br/>0 to 5</p> <p>Default = 0</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| ESSIDMAx_TX0_DMA_CHANNEL   | <p>DMA Channel to use for the ESSIx Transmitter 0</p> <p>Valid values are:<br/>0 to 5</p> <p>Default = 1</p>                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| ESSIDMAx_TX1_DMA_CHANNEL   | <p>DMA Channel to use for the ESSIx Transmitter 1</p> <p>Valid values are:<br/>0 to 5</p> <p>Default = 2</p>                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| ESSIDMAx_TX2_DMA_CHANNEL   | <p>DMA Channel to use for the ESSIx Transmitter 2</p> <p>Valid values are:<br/>0 to 5</p> <p>Default = 3</p>                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| ESSIDMAx_RX_CALLBACK       | <p>A user function for the ESSIDMA driver to call when the DMA for the receiver of ESSIx has completed its transfer. The user function must be of type <i>types_tCallback</i>:</p> <pre>typedef void (*types_tCallback) (void *pCallbackArg);</pre> <p>The parameters to the callback function are:</p> <p><i>pCallbackArg</i> - The original value specified by the user for ESSIDMAx_RX_CALLBACK_ARG</p> <p>Default = NULL</p>                                                                                                                       |
| ESSIDMAx_RX_CALLBACK_ARG   | <p>A value that the user wants to pass to the ESSIDMAx_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.</p> <p>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 ESSIDMAx_RX_CALLBACK function.</p> <p>Default = NULL</p> |

**Table 5-31. ESSI DMA Configuration Items for *appconfig.h* (Continued)**

| ESSI DMA Defined Constants      | Explanation and Default Value                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|---------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ESSIDMAx_RX_CALLBACK_EXCEPTION  | <p>A user function for the ESSIDMA driver to call when an exception has been detected. The user function must be of type <i>types_tErrorCallback</i>:</p> <pre>typedef void (*types_tErrorCallback) (UWord16 Error);</pre> <p>The parameters to the callback function are:</p> <p><i>Error</i> - Contains the exception</p> <p>Default = NULL</p>                                                                                                                                                                                                        |
| ESSIDMAx_TX0_CALLBACK           | <p>A user function for the ESSIDMA driver to call when the DMA for transmitter 0 of ESSIx has completed its transfer. The user function must be of type <i>types_tCallback</i>:</p> <pre>typedef void (*types_tCallback) (void *pCallbackArg);</pre> <p>The parameters to the callback function are:</p> <p><i>pCallbackArg</i> - The original value specified by the user for ESSIDMAx_TX0_CALLBACK_ARG</p> <p>Default = NULL</p>                                                                                                                       |
| ESSIDMAx_TX0_CALLBACK_ARG       | <p>A value that the user wants to pass to the ESSIDMAx_TX0_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.</p> <p>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 ESSIDMAx_TX0_CALLBACK function.</p> <p>Default = NULL</p> |
| ESSIDMAx_TX0_CALLBACK_EXCEPTION | <p>A user function for the ESSIDMA driver to call when an exception has been detected. The user function must be of type <i>types_tErrorCallback</i>:</p> <pre>typedef void (*types_tErrorCallback) (UWord16 Error);</pre> <p>The parameters to the callback function are:</p> <p><i>Error</i> - Contains the exception</p> <p>Default = NULL</p>                                                                                                                                                                                                        |



Table 5-31. ESSI DMA Configuration Items for *appconfig.h* (Continued)

| ESSI DMA Defined Constants      | Explanation and Default Value                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|---------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ESSIDMAx_TX1_CALLBACK           | <p>A user function for the ESSIDMA driver to call when the DMA for transmitter 1 of ESSIx has completed its transfer. The user function must be of type <i>types_tCallback</i>:</p> <pre>typedef void (*types_tCallback) (void *pCallbackArg);</pre> <p>The parameters to the callback function are:</p> <p><i>pCallbackArg</i> - The original value specified by the user for ESSIDMAx_TX1_CALLBACK_ARG</p> <p>Default = NULL</p>                                                                                                                       |
| ESSIDMAx_TX1_CALLBACK_ARG       | <p>A value that the user wants to pass to the ESSIDMAx_TX1_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.</p> <p>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 ESSIDMAx_TX1_CALLBACK function.</p> <p>Default = NULL</p> |
| ESSIDMAx_TX1_CALLBACK_EXCEPTION | <p>A user function for the ESSIDMA driver to call when an exception has been detected. The user function must be of type <i>types_tErrorCallback</i>:</p> <pre>typedef void (*types_tErrorCallback) (UWord16 Error);</pre> <p>The parameters to the callback function are:</p> <p><i>Error</i> - Contains the exception</p> <p>Default = NULL</p>                                                                                                                                                                                                        |
| ESSIDMAx_TX2_CALLBACK           | <p>A user function for the ESSIDMA driver to call when the DMA for transmitter 2 of ESSIx has completed its transfer. The user function must be of type <i>types_tCallback</i>:</p> <pre>typedef void (*types_tCallback) (void *pCallbackArg);</pre> <p>The parameters to the callback function are:</p> <p><i>pCallbackArg</i> - The original value specified by the user for ESSIDMAx_TX2_CALLBACK_ARG</p> <p>Default = NULL</p>                                                                                                                       |

**Table 5-31. ESSI DMA Configuration Items for *appconfig.h* (Continued)**

| ESSI DMA Defined Constants      | Explanation and Default Value                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|---------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ESSIDMAx_TX2_CALLBACK_ARG       | <p>A value that the user would like passed to the ESSIDMAx_TX2_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.</p> <p>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 ESSIDMAx_TX2_CALLBACK function.</p> <p>Default = NULL</p> |
| ESSIDMAx_TX2_CALLBACK_EXCEPTION | <p>A user function for the ESSIDMA driver to call when an exception has been detected. The user function must be of type <i>types_tErrorCallback</i>:</p> <pre>typedef void (*types_tErrorCallback) (UWord16 Error);</pre> <p>The parameters to the callback function are:</p> <p><i>Error</i> - Contains the exception</p> <p>Default = NULL</p>                                                                                                                                                                                                            |
| ESSIDMAx_FRAME_RATE             | <p>Number of Frames syncs per second for ESSIx. This is only used when the ESSIx is configured as a master.</p> <p>Default = 8000</p>                                                                                                                                                                                                                                                                                                                                                                                                                        |
| ESSIDMAx_NUM_TIME_SLOTS         | <p>The number of time slots to be used when ESSIx is configured for network mode.</p> <p>Valid values are:<br/>1 to 32</p> <p>Default = 2</p>                                                                                                                                                                                                                                                                                                                                                                                                                |
| ESSIDMAx_HW_FIFO_DEPTH          | <p>The ESSIx contains Receive and Transmit hardware FIFO that can buffer up to 8 samples. This value defines how many samples the ESSIx ISR or DMA will process.</p> <p>Valid values are:<br/>0 to 8</p> <p>Default = 0</p>                                                                                                                                                                                                                                                                                                                                  |
| ESSIDMAx_RX_FIFO_FULL_WM        | <p>When the ESSIx RX FIFO contains this many samples, the ESSIx Rx ISR or DMA will be called, if enabled.</p> <p>Valid values are:<br/>0 to 8</p> <p>Default = 0</p>                                                                                                                                                                                                                                                                                                                                                                                         |

**Table 5-31. ESSI DMA Configuration Items for *appconfig.h* (Continued)**

| ESSI DMA Defined Constants | Explanation and Default Value                                                                                                                                                                                                                                                                     |
|----------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ESSIDMAx_TX_FIFO_EMPTY_WM  | <p>When the ESSIx TX FIFO contains this many empty samples, the ESSIx Tx ISR or DMA will be called, if enabled.</p> <p>Valid values are:<br/>0 to 8</p> <p>Default = 0</p>                                                                                                                        |
| ESSIDMAx_PRESCALER         | <p>This bit controls a fixed divide-by-eight prescaler in series with the variable prescaler. It extends the range of the prescaler for those cases where a slower bit clock is desired.</p> <p>Valid values are:<br/>ESSI_PRESCALER_1<br/>ESSI_PRESCALER_8</p> <p>Default = ESSI_PRESCALER_1</p> |
| ESSIDMAx_WORD_LENGTH       | <p>Defines the length of a sample for ESSIx</p> <p>Valid values are:<br/>ESSI_WORD_LENGTH_8<br/>ESSI_WORD_LENGTH_10<br/>ESSI_WORD_LENGTH_12<br/>ESSI_WORD_LENGTH_16</p> <p>Default = ESSI_WORD_LENGTH_16</p>                                                                                      |
| ESSIDMAx_SYN               | <p>Synchronizes the ESSIx Receiver and Transmitters</p> <p>Valid values are:<br/>ESSI_SYNC_MODE_DISABLED<br/>ESSI_SYNC_MODE_ENABLED</p> <p>Default = ESSI_SYNC_MODE_ENABLED</p>                                                                                                                   |
| ESSIDMAx_TSHFD             | <p>Defines the ESSIx Transmitter's Shift Direction</p> <p>Valid values are:<br/>ESSI_TX_SHIFT_DIR_LSB_FIRST<br/>ESSI_TX_SHIFT_DIR_MSB_FIRST</p> <p>Default = ESSI_TX_SHIFT_DIR_MSB_FIRST</p>                                                                                                      |
| ESSIDMAx_TSCKP             | <p>Defines ESSIx Transmitter's Clock Polarity</p> <p>Valid values are:<br/>ESSI_TX_CLK_POL_FALLING_EDGE<br/>ESSI_TX_CLK_POL_RISING_EDGE</p> <p>Default = ESSI_TX_CLK_POL_RISING_EDGE</p>                                                                                                          |

Table 5-31. ESSI DMA Configuration Items for *appconfig.h* (Continued)

| ESSI DMA Defined Constants | Explanation and Default Value                                                                                                                                                                                                         |
|----------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ESSIDMAx_NET               | <p>Enables or disables network mode for ESSIx</p> <p>Valid values are:<br/>ESSI_NET_MODE_DISABLED<br/>ESSI_NET_MODE_ENABLED</p> <p>Default = ESSI_NET_MODE_ENABLED</p>                                                                |
| ESSIDMAx_TFSI              | <p>Defines whether the ESSIx Transmitter's Frame Sync is high or low</p> <p>Valid values are:<br/>ESSI_TX_FRAME_SYNC_ACTIVE_LOW<br/>ESSI_TX_FRAME_SYNC_ACTIVE_HI</p> <p>Default = ESSI_TX_FRAME_SYNC_ACTIVE_HI</p>                    |
| ESSIDMAx_TFSL              | <p>Defines whether the ESSIx Transmitter's Frame Sync is one-bit or one-word long</p> <p>Valid values are:<br/>ESSI_TX_ONE_BIT_FRAME_SYNC<br/>ESSI_TX_ONE_WORD_FRAME_SYNC</p> <p>Default = ESSI_TX_ONE_BIT_FRAME_SYNC</p>             |
| ESSIDMAx_TEFS              | <p>Defines whether the ESSIx Transmitter's Frame Sync occurs before or during the first data bit</p> <p>Valid values are:<br/>ESSI_TX_FRAME_SYNC_EARLY<br/>ESSI_TX_FRAME_SYNC_FIRST_BIT</p> <p>Default = ESSI_TX_FRAME_SYNC_EARLY</p> |
| ESSIDMAx_DIV4DIS           | <p>Defines whether the ESSIx clock is divided by one or 4</p> <p>Valid values are:<br/>ESSI_DIVIDE_MCU_CLOCK_1<br/>ESSI_DIVIDE_MCU_CLOCK_4</p> <p>Default = ESSI_DIVIDE_MCU_CLOCK_1</p>                                               |
| ESSIDMAx_RSHFD             | <p>Defines the ESSIx Receiver's Shift Direction</p> <p>Valid values are:<br/>ESSI_RX_SHIFT_DIR_MSB_FIRST<br/>ESSI_RX_SHIFT_DIR_LSB_FIRST</p> <p>Default = ESSI_RX_SHIFT_DIR_MSB_FIRST</p>                                             |

**Table 5-31. ESSI DMA Configuration Items for *appconfig.h* (Continued)**

| ESSI DMA Defined Constants | Explanation and Default Value                                                                                                                                                                                                                            |
|----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ESSIDMAx_RSCKP             | <p>Defines the ESSIx Receiver's Clock Polarity</p> <p>Valid values are:<br/>           ESSI_RX_CLK_POL_FALLING_EDGE<br/>           ESSI_RX_CLK_POL_RISING_EDGE</p> <p>Default = ESSI_RX_CLK_POL_FALLING_EDGE</p>                                         |
| ESSIDMAx_RFSI              | <p>Defines whether the ESSIx Receiver's Frame Sync is high or low</p> <p>Valid values are:<br/>           ESSI_RX_FRAME_SYNC_ACTIVE_HI<br/>           ESSI_RX_FRAME_SYNC_ACTIVE_LOW</p> <p>Default = ESSI_RX_FRAME_SYNC_ACTIVE_HI</p>                    |
| ESSIDMAx_RFSL              | <p>Defines whether the ESSIx Receiver's Frame Sync is one-bit or one-word long</p> <p>Valid values are:<br/>           ESSI_RX_ONE_WORD_FRAME_SYNC<br/>           ESSI_RX_ONE_BIT_FRAME_SYNC</p> <p>Default = ESSI_RX_ONE_BIT_FRAME_SYNC</p>             |
| ESSIDMAx_REFS              | <p>Defines whether the ESSIx Receiver's Frame Sync occurs before or during the first data bit</p> <p>Valid values are:<br/>           ESSI_RX_FRAME_SYNC_EARLY<br/>           ESSI_RX_FRAME_SYNC_FIRST_BIT</p> <p>Default = ESSI_RX_FRAME_SYNC_EARLY</p> |
| ESSIDMAx_RFEN              | <p>Enables or Disables the ESSIx Receiver's FIFO</p> <p>Valid values are:<br/>           ESSI_RX_FIFO_DISABLE<br/>           ESSI_RX_FIFO_ENABLE</p> <p>Default = ESSI_RX_FIFO_ENABLE</p>                                                                |
| ESSIDMAx_TFEN              | <p>Enables or Disables the ESSIx Transmitter's FIFO</p> <p>Valid values are:<br/>           ESSI_TX_FIFO_DISABLE<br/>           ESSI_TX_FIFO_ENABLE</p> <p>Default = ESSI_TX_FIFO_ENABLE</p>                                                             |

**Table 5-31. ESSI DMA Configuration Items for *appconfig.h* (Continued)**

| ESSI DMA Defined Constants  | Explanation and Default Value                                                                                                                                                                                                                                                  |
|-----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ESSIDMAx_SSC1               | <p>Defines the role of ESSIx SC1</p> <p>Valid values are:<br/>           ESSI_SC1_IS_SERIAL_IO_FLAG<br/>           ESSI_SC1_IS_TX0_DRV_ENABLE</p> <p>Default = ESSI_SC1_IS_SERIAL_IO_FLAG</p>                                                                                  |
| ESSIDMAx_TXDIR              | <p>Defines the direction of the ESSIx Transmit Clock</p> <p>Valid values are:<br/>           ESSI_TX_CLK_INTERNAL<br/>           ESSI_TX_CLK_EXTERNAL</p> <p>Default = ESSI_TX_CLK_EXTERNAL</p>                                                                                |
| ESSIDMAx_TXFSDIR            | <p>Defines the source of the ESSIx Transmit Frame Sync</p> <p>Valid values are:<br/>           ESSI_TX_FRM_SYNC_INTERNAL<br/>           ESSI_TX_FRM_SYNC_EXTERNAL</p> <p>Default = ESSI_TX_FRM_SYNC_EXTERNAL</p>                                                               |
| ESSIDMAx_RXFSDIR            | <p>Defines the source of the ESSIx Receive Frame Sync</p> <p>Valid values are:<br/>           ESSI_RX_FRM_SYNC_INTERNAL<br/>           ESSI_RX_FRM_SYNC_EXTERNAL</p> <p>Default = ESSI_RX_FRM_SYNC_EXTERNAL</p>                                                                |
| ESSIDMAx_RXDIR              | <p>Defines the direction of the ESSIx Receive Clock</p> <p>Valid values are:<br/>           ESSI_RX_CLK_INTERNAL<br/>           ESSI_RX_CLK_EXTERNAL</p> <p>Default = ESSI_RX_CLK_EXTERNAL</p>                                                                                 |
| ESSIDMAx_TX_SLOT_MASK_REG_A | <p>Defines whether the ESSIx Transmitter will transmit during a time slot or tri-state. This value is valid for time slots 0 to 15. A Logic One indicates transmit during the time slot. A Logic Zero indicates tri-state during that time slot.</p> <p>Default = 0xFFFFu</p>  |
| ESSIDMAx_TX_SLOT_MASK_REG_B | <p>Defines whether the ESSIx Transmitter will transmit during a time slot or tri-state. This value is valid for time slots 16 to 31. A Logic One indicates transmit during the time slot. A Logic Zero indicates tri-state during that time slot.</p> <p>Default = 0xFFFFu</p> |

**Table 5-31. ESSI DMA Configuration Items for *appconfig.h* (Continued)**

| ESSI DMA Defined Constants  | Explanation and Default Value                                                                                                                                                                                                                                         |
|-----------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ESSIDMAx_RX_SLOT_MASK_REG_A | Defines whether the ESSIx Receiver will receive during a time slot or ignore the data. This value is valid for time slots 0 to 15. A Logic One indicates receive during the time slot. A Logic Zero indicates ignore during that time slot.<br><br>Default = 0xFFFFu  |
| ESSIDMAx_RX_SLOT_MASK_REG_B | Defines whether the ESSIx Receiver will receive during a time slot or ignore the data. This value is valid for time slots 16 to 31. A Logic One indicates receive during the time slot. A Logic Zero indicates ignore during that time slot.<br><br>Default = 0xFFFFu |

**Code Example:** [Code Example 5-4](#) statically configures and initializes the ESSI DMA driver to use the device-independent API and to use callback functions.

**Code Example 5-17. Sample *appconfig.h* code**

```

#define INCLUDE_BSP /* include Board Support Package */
#define INCLUDE_IO /* include device-independent API (optional) */
#define INCLUDE_ESSIDMA /* include ESSI DMA driver (required) */

extern void Essi0RXISR (void *pCallbackArg);
extern void Essi0TX0ISR (void *pCallbackArg);
extern void Essi0TX1ISR (void *pCallbackArg);
extern void Essi0TX2ISR (void *pCallbackArg);

#define ESSIDMA0_RX_CALLBACK Essi0RXISR
#define ESSIDMA0_TX0_CALLBACK Essi0TX0ISR
#define ESSIDMA0_TX1_CALLBACK Essi0TX1ISR
#define ESSIDMA0_TX2_CALLBACK Essi0TX2ISR

```

### 5.4.3 API Definition

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

The ESSI DMA interface can be used at either of two levels: a device-independent interface or a device-dependent interface. The device-independent interface invokes the 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 interface, but is less portable. Your application may use either interface, depending on your specific goals for efficiency and portability. However, you should not mix the ESSI DMA 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 driver function call, nor may you use the File Descriptor returned by the device-dependent driver *essidmaOpen* call with any device-independent interface function call.

The following information may be found in the public header file *essidma.h*. You must include *essidma.h* in your application to utilize the ESSI DMA driver API.

#### 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);
 * 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);
 *****/
/*****
 * The device-dependent API function prototypes:
 * types_tHandle essidmaOpen(const char *pName, int OFlags);
 * ssize_t essidmaWrite(types_tHandle FileDesc, const void * pBuffer,
 * size_t NBytes);
 * ssize_t essidmaRead(types_tHandle FileDesc, void * pBuffer, size_t NBytes);
 * int essidmaClose(types_tHandle FileDesc);
 * UWord16 essidmaIoctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams,
 * const char * pName);
 *****/

```

#### Public Data Structure(s):

None



## 5.4.4 ESSI DMA Device-Independent API Specification

The following pages specify the Application Programming Interface (API) for the device-independent interface to the ESSI DMA driver.

To use the device-independent interface to the ESSI DMA driver, you must define both the `INCLUDE_ESSIDMA` 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_ESSIDMA
```

The user's application code then uses the ESSI DMA driver API as follows:

- An *open* call is made to initialize the ESSI device
- *ioctl* calls are made to control the ESSI device
- *write* calls are made to write data from the user buffer to the ESSI (transmit)
- *read* calls are made to read data from the ESSI to the user buffer (receive)
- After all ESSI operations are completed, the ESSI DMA driver is closed via a *close* call

Follow this sequence when using the ESSI DMA Driver in Non-Blocking mode:

- Open the ESSI DMA Device
- Call the *read* and/or *write* functions to set up the DMA buffers
- Call *ioctl(ESSIDMA\_DEVICE\_ENABLE)* to enable the ESSI
- By using this sequence, the ESSI Receiver and Transmitters will be synchronized
- If you enable the ESSI prior to setting up the DMA buffers, then the Receiver and Transmitters may not be synchronized

If you open the ESSI DMA Driver in Blocking mode, you will need to enable the ESSI prior to calling the first *read* or *write* function because the *read* or *write* function will not return until all of the data has been transferred. If the ESSI has not been enabled, then no data will be transferred and your program will be stuck in the *read* or *write* function. It really doesn't make sense to use the ESSI DMA Driver in Blocking mode if you plan on using a combination of the Receiver and Transmitters. For example, if configured for blocking mode and reading data from the Receiver, while waiting for all of the data to be received from the Receiver, the Transmitters won't have data to transmit.

The following sections describe each function of the device-independent API for the ESSI DMA 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.

### 5.4.4.1 *open*

**Call(s):**

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

**Arguments:**
**Table 5-32. ESSI DMA Driver Arguments - *open***

|                 |            |                                                                                                                                    |
|-----------------|------------|------------------------------------------------------------------------------------------------------------------------------------|
| <i>pName</i>    | <i>in</i>  | The ESSI DMA device name; see <a href="#">Table 5-33</a>                                                                           |
| <i>OFlag</i>    | <i>in</i>  | Open Mode Flags;<br>- O_RDWR - Read and Write (default)<br>- O_BLOCK - Blocking Mode (default)<br>- O_NONBLOCK - Non-Blocking Mode |
| <i>FileDesc</i> | <i>out</i> | ESSI DMA device descriptor returned by <i>open</i> call and used in <i>read/write/close/ioctl</i> function calls                   |

**Table 5-33. ESSI DMA Device Names for DSP5685x Chips**

| Chip                                                                                        | Can Use                                                                                                                                                                                                                                                                    |
|---------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>DSP56853</b><br><b>DSP56854</b><br><b>DSP56855</b><br><b>DSP56857</b><br><b>DSP56858</b> | BSP_DEVICE_NAME_ESSI_0_RX_DMA<br>BSP_DEVICE_NAME_ESSI_0_TX0_DMA<br>BSP_DEVICE_NAME_ESSI_0_TX1_DMA<br>BSP_DEVICE_NAME_ESSI_0_TX2_DMA<br>BSP_DEVICE_NAME_ESSI_1_RX_DMA<br>BSP_DEVICE_NAME_ESSI_1_TX0_DMA<br>BSP_DEVICE_NAME_ESSI_1_TX1_DMA<br>BSP_DEVICE_NAME_ESSI_1_TX2_DMA |

**Description:** This function opens the ESSI DMA peripheral for operations for use with the device-independent interface.

This function calls *essidmaOpen()*, so please see the description of *essidmaOpen()* for a description of this function.

**Returns:** The *open* call returns a type *types\_tHandle*. When the *open* call is successful, this *types\_tHandle* is a file descriptor for the ESSI DMA device and is used by the ESSI's *read*, *write*, *close*, and *ioctl* functions. If the *open* call is not successful, "-1" is returned.

**Code Example:** See [Code Example 5-18](#).

## 5.4.4.2 *write*

### Call(s):

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

### Arguments:

**Table 5-34. ESSI DMA Driver Arguments - *write***

|                 |           |                                                                                                                                      |
|-----------------|-----------|--------------------------------------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | ESSI DMA device descriptor returned by <i>open</i> call                                                                              |
| <i>pBuffer</i>  | <i>in</i> | Array containing the samples to be transmitted                                                                                       |
| <i>NBytes</i>   | <i>in</i> | Size (in 8-bit bytes) of the data to be written out of the ESSI DMA device; note that the number of 16-bit words is <i>NBytes</i> /2 |

**Description:** This function is mapped to *essidmaWrite()* via the *essidmadrvtIOInterfaceVT* structure. Please see the description of *essidmaWrite()* for a description of this function.

**Returns:** See *essidmaWrite*

**Code Example:** See [Code Example 5-18](#).

### 5.4.4.3 *read*

**Call(s):**

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

**Arguments:**

**Table 5-35. ESSI DMA Driver Arguments - *read***

|                 |           |                                                                                                                                  |
|-----------------|-----------|----------------------------------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | ESSI DMA device descriptor returned by <i>open</i> call                                                                          |
| <i>pBuffer</i>  | <i>in</i> | Array in which to store the received samples                                                                                     |
| <i>NBytes</i>   | <i>in</i> | Size (in 8-bit bytes) of the data to be read from the ESSI DMA device; note that the number of 16-bit words is <i>NBytes</i> /2. |

**Description:** This function is mapped to *essidmaRead()* via the *essidmadrIOInterfaceVT* structure. Please see the description of *essidmaRead()* for a description of this function.

**Returns:** See *essidmaRead*.

**Code Example:** See [Code Example 5-18](#)



### 5.4.4.4 *close*

**Call(s):**

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

**Arguments:**

**Table 5-36. ESSI DMA Driver Arguments - *close***

|                 |           |                                                         |
|-----------------|-----------|---------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | ESSI DMA device descriptor returned by <i>open</i> call |
|-----------------|-----------|---------------------------------------------------------|

**Description:** This function is mapped to *essidmaClose()* via the *essidmadrvtIOInterfaceVT* structure. Please see the description of *essidmaClose()* for a description of this function.

**Returns:** The function returns “0” if successful and “-1” if unsuccessful.

**Code Example:** See [Code Example 5-18](#).

### 5.4.4.5 *ioctl*

**Call(s):**

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

**Arguments:**

**Table 5-37. ESSI DMA Driver Arguments - *ioctl***

|                 |           |                                                                                  |
|-----------------|-----------|----------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | ESSI DMA device descriptor returned by <i>open</i> call                          |
| <i>Cmd</i>      | <i>in</i> | Commands found in <i>essidma.h</i> which are used to control the ESSI DMA driver |
| <i>pParams</i>  | <i>in</i> | Pointer to <i>ioctl</i> command parameter                                        |

**Table 5-38. ESSI DMA *ioctl* Commands**

| Cmd                         | <i>pParams</i>                                   | Return | Result                                                                                                                                                                                                                                                                                                                                                        |
|-----------------------------|--------------------------------------------------|--------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ESSIDMA_DEVICE_ENABLE       | NULL                                             | None   | This command enables the ESSI DMA device. Note that a <i>read()</i> or <i>write()</i> should be called prior to using this command in order to configure the DMA device to read from or write to a valid buffer. This command allows the user to synchronize the transfer of data by setting up all of the DMA buffers prior to enabling the ESSI peripheral. |
| ESSIDMA_DEVICE_DISABLE      | NULL                                             | None   | This command disables the ESSI DMA device                                                                                                                                                                                                                                                                                                                     |
| ESSIDMA_SET_CALLBACK        | void (*types_tCallback) (void *)                 | None   | This command installs a Callback function to be called by the ESSI DMA driver when a DMA transfer has completed                                                                                                                                                                                                                                               |
| ESSIDMA_SET_CALLBACK_ARG    | void * pCallbackArg                              | None   | This command defines the callback argument to be used when the callback function is called                                                                                                                                                                                                                                                                    |
| ESSIDMA_SET_CALLBACK_EXCEPT | void (*types_tErrorCallback) (UWord16 Exception) | None   | This command installs the Exception Callback function                                                                                                                                                                                                                                                                                                         |

**Description:** This function is used to dynamically control the ESSI DMA device.

**Returns:** The *ioctl* call returns a type *UWord16*. This *UWord16* is dependent on the command passed in as an input parameter. **Table 5-38** lists the return for each command.

**Code Example:** **Code Example 5-18** opens the ESSI1 DMA Receiver and Transmitter0 devices for Non-Blocking mode. This example uses callback functions which can be defined statically or

dynamically. If you place “DEFINE\_STATICALLY” in *appconfig.h*, then the callback functions will be defined statically. Otherwise, they will be defined dynamically using *ioctl()* commands.

#### Code Example 5-18. ESSI DMA Driver Device-Independent API Example

---

- file *appconfig.h*:

```
#define INCLUDE_BSP
#define INCLUDE_ESSIDMA
#define INCLUDE_IO

#define DEFINE_STATICALLY // comment this out if you want to configure the
 // callback functions dynamically

#ifdef DEFINE_STATICALLY
 void RxCallback(void * pArg);
 void TxCallback(void * pArg);

 #define ESSIDMA1_RX_CALLBACK RxCallback
 #define ESSIDMA1_TX0_CALLBACK TxCallback
#else
 #endif
```

- user application:

```
#include "port.h"
#include "bsp.h"
#include "essidma.h"

void RxCallback(void * pArg);
void TxCallback(void * pArg);

void RxCallback(void * pArg)
{
}

void TxCallback(void * pArg)
{
}

void main(void)
{
 types_tHandle EssiRx, EssiTx;
 int RxBuffer[10], TxBuffer[10];

 // Open ESSI Receiver and Transmitter 0

 EssiRx = open(BSP_DEVICE_NAME_ESSI_1_RX_DMA, O_NONBLOCK);
 EssiTx = open(BSP_DEVICE_NAME_ESSI_1_TX0_DMA, O_NONBLOCK);

 #ifndef DEFINE_STATICALLY
```

```

 // Define Rx and Tx callback functions

 ioctl(EssiRx, ESSIDMA_SET_CALLBACK, RxCallback);
 ioctl(EssiTx, ESSIDMA_SET_CALLBACK, TxCallback);

 #endif

 // Read data from ESSI receiver

 read(EssiRx, (void *)&RxBuffer[0], sizeof(RxBuffer));

 // Write data to ESSI transmitter 0

 write(EssiTx, (void *)&TxBuffer[0], sizeof(TxBuffer));

 // Enable the ESSI Device

 ioctl(EssiRx, ESSIDMA_DEVICE_ENABLE, NULL);

 while(1)
 {
 // Wait for callbacks to be called
 // Read more data
 // Transmit more data
 // ...
 }

 // Disable the ESSI Device

 ioctl(EssiRx, ESSIDMA_DEVICE_DISABLE, NULL);

 // close ESSI RX and TX DMA drivers

 close(EssiRx);
 close(EssiTx);
}

```



## 5.4.5 ESSI DMA Device-Dependent API Specification

The following pages specify the Application Programming Interface (API) for the device-dependent interface to the ESSI DMA driver.

To use the device-dependent interface to the ESSI DMA driver, you must define the `INCLUDE_ESSIDMA` variable in the *appconfig.h* file of the SDK Embedded Project you created in CodeWarrior:

```
#define INCLUDE_ESSIDMA
```

The user's application code then uses the ESSI DMA driver API as follows:

- A *essidmaOpen* call is made to initialize the ESSI device
- *essidmaIoctl* calls are made to control the ESSI device
- *essidmaWrite* calls are made to write data from the user buffer to the ESSI (transmit)
- *essidmaRead* calls are made to read data from the ESSI to the user buffer (receive)
- After all ESSI operations are completed, the ESSI DMA driver is closed via an *essidmaClose* call

Follow this sequence when using the ESSI DMA Driver in Non-Blocking mode:

- Open the ESSI DMA Device
- Call the *essidmaRead* and/or *essidmaWrite* functions to set up the DMA buffers
- Call *essidmaIoctl(ESSIDMA\_DEVICE\_ENABLE)* to enable the ESSI
- By using this sequence, the ESSI Receiver and Transmitters will be synchronized
- If you enable the ESSI prior to setting up the DMA buffers, then the Receiver and Transmitters may not be synchronized

If you open the ESSI DMA Driver in Blocking mode, you will need to enable the ESSI prior to calling the first *essidmaRead* or *essidmaWrite* function because the *essidmaRead* or *essidmaWrite* function will not return until all of the data has been transferred. If the ESSI has not been enabled, then no data will be transferred and your program will be stuck in the *essidmaRead* or *essidmaWrite* function. It really doesn't make sense to use the ESSI DMA Driver in blocking mode if you plan on using a combination of the Receiver and Transmitters. For example, if configured for blocking mode and reading data from the Receiver, while waiting for all of the data to be received from the Receiver, the Transmitters won't have data to transmit.

The following sections describe each function of the device-dependent API for the ESSI DMA 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.

### 5.4.5.1 *essidmaOpen*

**Call(s):**

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

**Arguments:**

**Table 5-39. ESSI DMA Driver Arguments - *essidmaOpen***

|                 |            |                                                                                                                                                     |
|-----------------|------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>pName</i>    | <i>in</i>  | The ESSI DMA device name; see <a href="#">Table 5-40</a>                                                                                            |
| <i>OFlag</i>    | <i>in</i>  | Open Mode Flags;<br>- O_RDWR - Read and Write (default)<br>- O_BLOCK - Blocking Mode (default)<br>- O_NONBLOCK - Non-Blocking Mode                  |
| <i>FileDesc</i> | <i>out</i> | ESSI DMA device descriptor returned by <i>essidmaOpen</i> call and used in <i>essidmaRead/essidmaWrite/essidmaClose/essidmaIoctl</i> function calls |

**Table 5-40. ESSI DMA Device Names for DSP5685x Chips**

| Chip                                                                                        | Can Use                                                                                                                                                                                                                                                                    |
|---------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>DSP56853</b><br><b>DSP56854</b><br><b>DSP56855</b><br><b>DSP56857</b><br><b>DSP56858</b> | BSP_DEVICE_NAME_ESSI_0_RX_DMA<br>BSP_DEVICE_NAME_ESSI_0_TX0_DMA<br>BSP_DEVICE_NAME_ESSI_0_TX1_DMA<br>BSP_DEVICE_NAME_ESSI_0_TX2_DMA<br>BSP_DEVICE_NAME_ESSI_1_RX_DMA<br>BSP_DEVICE_NAME_ESSI_1_TX0_DMA<br>BSP_DEVICE_NAME_ESSI_1_TX1_DMA<br>BSP_DEVICE_NAME_ESSI_1_TX2_DMA |

**Description:** This function opens the ESSI DMA peripheral for operations for use with the device-dependent interface.

If ESSI 0 is being opened, *essidmaOpen* configures GPIO C to be a peripheral.

If ESSI 1 is being opened, *essidmaOpen* configures GPIO D to be a peripheral.

This function opens a DMA device via *dmaOpen()* to be used by this driver. The DMA channel to be used and its configuration is defined in *config.h* and can be redefined by the user in *appconfig.h*. The user should not change *config.h*.

This function does not enable the ESSI since additional set-up is required prior to DMA transfer. The user must use an *essidmaIoctl(ESSIDMA\_DEVICE\_ENABLE)* command to do this.

**Returns:** The *essidmaOpen* call returns a type *types\_tHandle*. When the *essidmaOpen* call is successful, this *types\_tHandle* is a file descriptor for the ESSI DMA device and is used by the ESSI *essidmaRead*, *essidmaWrite*, *essidmaClose*, and *essidmaIoctl* functions. When the *essidmaOpen* call is not successful, "-1" is returned.

**Code Example:** See [Code Example 5-19](#).

### 5.4.5.2 *essidmaWrite*

**Call(s):**

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

**Arguments:**

**Table 5-41. ESSI DMA Driver Arguments - *essidmaWrite***

|                 |           |                                                                                                                                            |
|-----------------|-----------|--------------------------------------------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | ESSI DMA device descriptor returned by <i>essidmaOpen</i> call                                                                             |
| <i>pBuffer</i>  | <i>in</i> | Array containing the samples to be transmitted                                                                                             |
| <i>NBytes</i>   | <i>in</i> | Size (in 8-bit bytes) of the data to be written out of the ESSI DMA device;<br><b>Note:</b> The number of 16-bit words is <i>NBytes</i> /2 |

**Description:** This function uses the DMA Driver to write user data to the ESSI Transmitter defined in the *essidmaOpen()* call. This function is only valid for ESSI DMA Transmitter devices.

If the specific ESSI Transmitter (TX0, TX1, or TX2) is not enabled, this function enables the Transmitter. This means that when the “overall” ESSI is enabled, data will be transmitted out of the selected transmitter. This function does not enable the ESSI. The user must use *essidmaIoctl(ESSIDMA\_DEVICE\_ENABLE)* to enable the ESSI. This allows the user to synchronize all of the DMA buffers for the ESSI peripheral by enabling the ESSI after configuring the DMA for each ESSI Transmitter and Receiver.

This function calls *dmaWrite()* to configure the DMA to transfer *NBytes* from *pBuffer* to ESSI TX0, TX1, or TX2. If an Exception Callback is defined (i.e., *ESSIDMA\_CALLBACK\_EXCEPTION*), this function checks for an exception and, if an exception exists, calls the Exception Callback.

If the ESSI DMA Driver is configured for Blocking mode, then this function returns after the DMA Driver has transferred *NBytes* from *pBuffer*, at which point the user's application can call *essidmaWrite()* or disable the ESSI.

If the ESSI DMA Driver is configured for Non-Blocking mode, then this function returns immediately. The ESSI DMA Driver will call the ESSI Callback function after the DMA Driver has read *NBytes* from *pBuffer*, at which point the user's application can call *essidmaWrite()* or disable the ESSI.

If the ESSI is configured for Network mode, then the user application must interleave the data in *pBuffer* for each time slot.

**Returns:** The *essidmaWrite* call returns a type *ssize\_t*, which is the number of bytes of data written. In Non-Blocking mode, the size will be 0, since the *dmaWrite* function returns before any data is written. In Blocking mode, the size is the number of bytes of data written.

**Code Example:** See [Code Example 5-19](#).

### 5.4.5.3 *essidmaRead*

**Call(s):**

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

**Arguments:**

**Table 5-42. ESSI DMA Driver Arguments - *essidmaRead***

|                 |           |                                                                                                                                      |
|-----------------|-----------|--------------------------------------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | ESSI DMA device descriptor returned by <i>essidmaOpen</i> call                                                                       |
| <i>pBuffer</i>  | <i>in</i> | Array in which to store the received samples                                                                                         |
| <i>NBytes</i>   | <i>in</i> | Size (in 8-bit bytes) of the data to be read from the ESSI DMA device<br><b>Note:</b> The number of 16-bit words is <i>NBytes</i> /2 |

**Description:** This function uses the DMA Driver to read data from the ESSI Receiver and store it in the user's buffer. This function is only valid for ESSI DMA Receiver devices.

If the ESSI Receiver is not enabled, this function enables it. This means that when the “overall” ESSI is enabled, data will be received from the receiver. This function does not enable the ESSI. The user must use *essidmaIoctl(ESSIDMA\_DEVICE\_ENABLE)* to enable the ESSI. This allows the user to synchronize all of the DMA buffers for the ESSI peripheral by enabling the ESSI after configuring the DMA for each ESSI Transmitter and Receiver.

This function calls *dmaRead()* to configure the DMA Driver to transfer *NBytes* from the ESSI Receiver to *pBuffer*. If an Exception Callback is defined (i.e., *ESSIDMA\_CALLBACK\_EXCEPTION*), this function checks for an Exception and, if an exception exists, calls the callback.

If the ESSI DMA Driver is configured for Blocking mode, then this function returns after the DMA Driver has transferred *NBytes* to *pBuffer*, at which point the user's application can call *essidmaRead()* to get more data or disable the ESSI.

If the ESSI DMA Driver is configured for Non-Blocking mode, then this function returns immediately. The ESSI DMA Driver will call the ESSI Callback function after the DMA Driver has transferred *NBytes* to *pBuffer*, at which point the user's application can call *essidmaRead()* to get more data or disable the ESSI.

If the ESSI is configured for Network mode, then the user application must de-interleave the data in *pBuffer* for each time slot.

**Returns:** The *essidmaRead* call returns a type *ssize\_t*. It is the number of bytes of data read. In Non-Blocking mode, the size will be 0, since the *dmaRead* function returns before any data is read. In Blocking mode, the size is the number of bytes of data read.

**Code Example:** See [Code Example 5-19](#)

### 5.4.5.4 *essidmaClose*

Call(s):

```
int essidmaClose (types_tHandle FileDesc);
```

Arguments:

Table 5-43. ESSI DMA Driver Arguments - *essidmaClose*

|                 |           |                                                                |
|-----------------|-----------|----------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | ESSI DMA device descriptor returned by <i>essidmaOpen</i> call |
|-----------------|-----------|----------------------------------------------------------------|

**Description:** This function disables a specific ESSI DMA device (i.e., Receiver, Transmitter 0, Transmitter 1, or Transmitter 2). If all of the specific DMA devices have been disabled for a specific ESSI, then this function will disable the ESSI also.

**Returns:** If successful, this function returns “0”; if unsuccessful, it returns “-1”.

**Code Example:** See [Code Example 5-19](#).

### 5.4.5.5 *essidmaIoctl*

#### Call(s):

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

#### Arguments:

**Table 5-44. ESSI DMA Driver Arguments - *essidmaIoctl***

|                 |           |                                                                                       |
|-----------------|-----------|---------------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | ESSI DMA device descriptor returned by <i>essidmaOpen</i> call                        |
| <i>Cmd</i>      | <i>in</i> | Commands found in <i>essidma.h</i> , which are used to manage the ESSI DMA driver     |
| <i>pParams</i>  | <i>in</i> | Pointer to <i>essidmaIoctl</i> command parameter                                      |
| <i>pName</i>    | <i>in</i> | The ESSI DMA device name from <i>bsp.h</i> ; typically, BSP_DEVICE_NAME_ESSI_0_RX_DMA |

**Table 5-38** contains the list of commands for the ESSI DMA *ioctl* function, which are identical to commands for the *essidmaIoctl* function

**Description:** This function is used to dynamically control the ESSI DMA device.

**Returns:** The *essidmaIoctl* call returns a value that is dependent on the command passed in as an input parameter. **Table 5-38** lists a return for each command.

**Code Example:** **Code Example 5-19** opens the ESSI1 DMA Receiver and Transmitter0 Devices for Non-Blocking mode. This example uses callback functions which can be defined statically or dynamically. If you place “DEFINE\_STATICALLY” in *appconfig.h*, then the callback functions will be defined statically. Otherwise, they will be defined dynamically using *ioctl()* commands.

#### Code Example 5-19. ESSI DMA Driver Device-Dependent API Example

- file *appconfig.h*:

```
#define INCLUDE_BSP
#define INCLUDE_ESSIDMA

#define DEFINE_STATICALLY // comment this out if you want to configure the
 // callback functions dynamically

#ifdef DEFINE_STATICALLY
 void RxCallback(void * pArg);
 void TxCallback(void * pArg);

 #define ESSIDMA1_RX_CALLBACK RxCallback
 #define ESSIDMA1_TX0_CALLBACK TxCallback
#endif
```



- user application:

```
#include "port.h"
#include "bsp.h"
#include "essidma.h"

void RxCallback(void * pArg);
void TxCallback(void * pArg);

void RxCallback(void * pArg)
{
}

void TxCallback(void * pArg)
{
}

void main(void)
{
 types_tHandle EssiRx, EssiTx;
 int RxBuffer[10], TxBuffer[10];

 // Open ESSI Receiver and Transmitter 0

 EssiRx = essidmaOpen(BSP_DEVICE_NAME_ESSI_1_RX_DMA, 0_NONBLOCK);
 EssiTx = essidmaOpen(BSP_DEVICE_NAME_ESSI_1_TX0_DMA, 0_NONBLOCK);

 #ifndef DEFINE_STATICALLY

 // Define Rx and Tx callback functions

 essidmaIoctl(EssiRx, ESSIDMA_SET_CALLBACK, RxCallback,
 BSP_DEVICE_NAME_ESSI_1_RX_DMA);
 essidmaIoctl(EssiTx, ESSIDMA_SET_CALLBACK, TxCallback,
 BSP_DEVICE_NAME_ESSI_1_TX0_DMA);

 #endif

 // Read data from ESSI receiver

 essidmaRead(EssiRx, (void *)&RxBuffer[0], sizeof(RxBuffer));

 // Write data to ESSI transmitter 0

 essidmaWrite(EssiTx, (void *)&TxBuffer[0], sizeof(TxBuffer));

 // Enable the ESSI Device

 essidmaIoctl(EssiRx, ESSIDMA_DEVICE_ENABLE, NULL,
 BSP_DEVICE_NAME_ESSI_1_RX_DMA);

 while(1)
 {
 // Wait for callbacks to be called
 }
}
```



ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

```
// Read more data
// Transmit more data
// ...
}

// Disable the ESSI Device

essidmaIoctl(EssiRx, ESSIDMA_DEVICE_DISABLE, NULL,
 BSP_DEVICE_NAME_ESSI_1_RX_DMA));

// close ESSI RX and TX DMA drivers

essidmaClose(EssiRx);
essidmaClose(EssiTx);
}
```



## 5.4.6 ESSI DMA Driver Tests and Demos

Please refer to the Codec DMA Driver application in [Section 6.3](#). The Codec DMA application uses the Codec DMA driver, which is implemented using the ESSI DMA Driver.

## 5.5 GPIO Driver

### 5.5.1 Introduction

The General Purpose I/O (GPIO) interface manipulates external signals routed through general purpose pins. Typically, each pin 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 **DSP5685x 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 DSP56858:

The DSP56858 has no dedicated GPIO and has 47 multiplexed GPIOs on Ports A, B, C, D, E, F, G, and H.

### 5.5.2 Initialization

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

### 5.5.3 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



## 5.5.4 GPIO 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.

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, NULL);
```

**Arguments:**
**Table 5-45. GPIO Driver Arguments - *open***

|                 |            |                                                                                             |
|-----------------|------------|---------------------------------------------------------------------------------------------|
| <i>pName</i>    | <i>in</i>  | The GPIO device name; see <a href="#">Table 5-46</a>                                        |
| <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 5-46. GPIO Ports Available for the DSP5685x Chip**

| Chip     | Can Use                                                                                                                                                                                                      |
|----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DSP56858 | BSP_DEVICE_NAME_GPIO_A<br>BSP_DEVICE_NAME_GPIO_B<br>BSP_DEVICE_NAME_GPIO_C<br>BSP_DEVICE_NAME_GPIO_D<br>BSP_DEVICE_NAME_GPIO_E<br>BSP_DEVICE_NAME_GPIO_F<br>BSP_DEVICE_NAME_GPIO_G<br>BSP_DEVICE_NAME_GPIO_H |

**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 which 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 5-20](#).



### 5.5.4.2 *close*

Call(s):

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

Arguments:

Table 5-47. 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 5-20](#).

### 5.5.4.3 *ioctl*

#### Call(s):

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

#### Arguments:

**Table 5-48. 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; it's used to pass on a particular pin(s) on a port in which to perform one of the commands in <a href="#">Table 5-49</a> . The <i>gpioPin</i> macro defined in <a href="#">Code Example 5-20</a> is used to obtain a mask for that particular pin and port. |

**Table 5-49. 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<br>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 DSP5685x, [Code Example 5-20](#) toggles Port A, pin 0, between a logic high and logic low.

#### Code Example 5-20. 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);
*****/

```



## 5.5.5 GPIO 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.

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 *gpioOpen*

Call(s):

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

**Table 5-50. GPIO Driver Arguments - *gpioOpen***

|              |           |                                                      |
|--------------|-----------|------------------------------------------------------|
| <i>pName</i> | <i>in</i> | The GPIO device name; see <a href="#">Table 5-46</a> |
| <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.

5.5.5.2 *gpioClose*

Call(s):

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

Table 5-51. GPIO Driver Arguments - *close*

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

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

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



5.5.5.3 *gploctI*

Call(s):

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

Table 5-52. GPIO Driver Arguments

|                   |    |                                                                                                                                                                                                                                                                                                                                                                  |
|-------------------|----|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i>   | in | GPIO device descriptor returned by <i>open</i> call and used in <i>ioctI</i> function calls                                                                                                                                                                                                                                                                      |
| <i>Cmd</i>        | in | Commands found in <i>gpio.h</i> which are used to modify GPIO pins; see <a href="#">Table 5-53</a>                                                                                                                                                                                                                                                               |
| <i>Params</i>     | in | <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; it's 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 <a href="#">Code Example 5-21</a> is used to obtain a mask for that particular pin and port. |
| <i>gpioDevice</i> | in | The GPIO device name; see <a href="#">Table 5-46</a>                                                                                                                                                                                                                                                                                                             |

**Table 5-53. *gpioIoctl* Commands**

| Cmd                   | <i>pParams</i>          | Return | Result                                                                             |
|-----------------------|-------------------------|--------|------------------------------------------------------------------------------------|
| GPIO_SETAS_GPIO       | gpioPin(Port,Pin) macro | None   | When peripheral is disabled, DDR determines direction of data flow in PER register |
| GPIO_SETAS_PERIPHERAL | gpioPin(Port,Pin) macro | None   | A peripheral masters the GPIO pin, in PER register                                 |
| GPIO_SETAS_INPUT      | gpioPin(Port,Pin) macro | None   | Sets a GPIO pin as an input, in DDR register                                       |
| GPIO_SETAS_OUTPUT     | gpioPin(Port,Pin) macro | None   | Sets a GPIO pin as an output, in DDR register                                      |
| GPIO_DISABLE_PULLUP   | gpioPin(Port,Pin) macro | None   | Disables pull-up, in PUR register                                                  |
| GPIO_ENABLE_PULLUP    | gpioPin(Port,Pin) macro | None   | Enables pull-up, in PUR register                                                   |
| GPIO_SET              | gpioPin(Port,Pin) macro | None   | Sets a GPIO signal                                                                 |
| GPIO_CLEAR            | gpioPin(Port,Pin) macro | None   | Clears a GPIO signal                                                               |
| GPIO_TOGGLE           | gpioPin(Port,Pin) macro | None   | Toggles a GPIO signal                                                              |
| GPIO_READ             | gpioPin(Port,Pin) macro | 0 or 1 | Reads the value of an input pin<br>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 5-21](#).

#### Code Example 5-21. 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);
}
/*****/

```

### 5.5.6 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.

## 5.6 Host Interface Driver

### 5.6.1 Introduction

The DSP5685x processor has one Host Interface (HI). This HI device allows full-duplex, asynchronous serial communication between the DSP, a host computer, or other devices.

### 5.6.2 Initialization

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

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

```
#define INCLUDE_HI
```

Defining the preprocessor variable, INCLUDE\_HI, directs the SDK to include the HI driver in the executable image. If the preprocessor variable INCLUDE\_HI is not defined, then the HI 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 HI driver. The list of these variables is provided in [Table 5-54](#).

**Table 5-54. HI Configuration Items for *appconfig.h***

| HI #define               | Explanation and Default Value                                                                                                                                                                                                                                 |
|--------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| HI_SEND_BUFFER_LENGTH    | Fifo buffer size for write operation for HI<br><br>Default = 8                                                                                                                                                                                                |
| HI_RECEIVE_BUFFER_LENGTH | Fifo buffer size for read operation for HI<br><br>Default = 8                                                                                                                                                                                                 |
| HI_TX_CALLBACK_FUNCTION  | A user function for the HI driver to call when the TX buffer has no more samples to transmit. The user function must be of type <i>tpHiDataCallback</i> :<br><pre>typedef void (*tpHiDataCallback)(void);</pre> Default = NULL                                |
| HI_RX_CALLBACK_FUNCTION  | A user function for the HI driver to call when it has received at least HI_READ_WATERMARK samples in its internal RX buffer. The user function must be of type <i>tpHiDataCallback</i> :<br><pre>typedef void (*tpHiDataCallback)(void);</pre> Default = NULL |

Table 5-54. HI Configuration Items for *appconfig.h* (Continued)

| HI #define                 | Explanation and Default Value                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|----------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| HI_ERROR_CALLBACK_FUNCTION | <p>A user function for the HI driver to call when it has detected an exception condition. The user function must be of type <i>tpHiErrorCallback</i></p> <pre>typedef void (*tpHiErrorCallback)(UWord16 Exception);</pre> <p>The parameters to the callback function are:<br/> <i>Exception</i> - The actual exception that occurred<br/> 0x0001 - HI_EXCEPTION_RX_BUFFER_OVERFLOW</p> <p>Default = NULL</p>                                                                                                                                                                      |
| HI_READ_WATERMARK          | <p>The number of bytes that the HI must receive before calling the user function specified by HI_RX_CALLBACK_FUNCTION. The value specified must be less than HI_RECEIVE_BUFFER_LENGTH.</p> <p>Increasing the HI_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.</p> <p>Default = 0</p> |
| HI_SAMPLE_SIZE             | <p>The size of the Host Interface sample.</p> <p>If HI_BYTE is selected, then the type of data used by the read and write functions is char. The read function reads from the most significant byte of the word-wide host interface receive register and writes to the least significant byte of the word-wide host interface transmit register.</p> <p>If HI_WORD is selected, then the type of data used by the read and write functions is Word16.</p> <p>Default = HI_BYTE</p>                                                                                                |
| HI_HOST_STROBE_MODE        | <p>The mode for strobing the data.</p> <p>If HI_HOST_SINGLE_DATA_STROBE is selected, then a single data strobe is generated for both read and writes.</p> <p>If HI_HOST_DUAL_DATA_STROBE is selected, then separate read and write strobes are generated.</p> <p>Default = HI_HOST_SINGLE_DATA_STROBE</p>                                                                                                                                                                                                                                                                         |
| HI_HOST_REQ_MODE_SELECT    | <p>Determines the host request pins.</p> <p>If HI_HOST_REQ_MODE_SELECT_HREQ_HACK is selected, then HREQ and HACK are active.</p> <p>If HI_HOST_REQ_MODE_SELECT_HTRQ_HRRQ is selected, then HTRQ and HRRQ are active.</p> <p>Default = HI_HOST_REQ_MODE_SELECT_HREQ_HACK</p>                                                                                                                                                                                                                                                                                                       |

### 5.6.3 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 HI driver in an SDK Embedded Project, insert the following line in the *appconfig.h* file:

```
#define INCLUDE_HI
```

The following information may be found in the public header file *hi.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 hiOpen(const char *pName, int OFlags);
* ssize_t hiWrite(types_tHandle FileDesc, const void * pBuffer, size_t NBytes);
* ssize_t hiRead(types_tHandle FileDesc, void * pBuffer, size_t NBytes);
* int hiClose(types_tHandle FileDesc);
* UWord16 hiIoctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams,
* const char * pName);
*****/

```

#### Public Data Structure(s):

None





## 5.6.4 HI 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.

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 5-55. HI Driver Arguments - *open***

|                 |            |                                                                                                      |
|-----------------|------------|------------------------------------------------------------------------------------------------------|
| <i>pName</i>    | <i>in</i>  | The HI device name; see <a href="#">Table 5-56</a>                                                   |
| <i>OFlag</i>    | <i>in</i>  | Open Mode Flags; specify O_RDWR<br>Use O_NONBLOCK flags for Non-Blocking mode                        |
| <i>FileDesc</i> | <i>out</i> | HI device descriptor returned by <i>open</i> call and used in <i>read/write/ioctl</i> function calls |

**Table 5-56. HI Device Names for DSP5685x Chips**

| Chip                                         | Can Use            |
|----------------------------------------------|--------------------|
| DSP56853<br>DSP56854<br>DSP56857<br>DSP56858 | BSP_DEVICE_NAME_HI |
| DSP56855                                     | N/A                |

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

The *open* call initializes the internal memory buffers used by the HI driver. It configures the driver for blocking or non-blocking modes, as defined in the *OFlag* parameter. Finally, it enables the HI Receiver interrupts. It does not enable the HI transmitter interrupts, as this is done when the *write* function is called. The default mode for *open*, as defined in *config.h* and *config.c*, is the following:

Default Host Tx Buffer Size = 8

Default Host Rx Buffer Size = 8

Default Host Req Mode = HREQ and HACK selected

Default Host Strobe Mode = Single Strobe

Default Host Rx Callback = NULL

Default Host Tx Callback = NULL

Default Host Exception Callback = NULL

Default Host Read Watermark = 0

Default Host Sample Size = Byte Wide



**Returns:** The *open* call returns a type *types\_tHandle*. When the *open* call is successful, this *types\_tHandle* is a file descriptor for the HI device and is used by the HI's *read*, *write*, and *ioctl* functions. A “-1” is returned when the *open* call is not successful.

**Code Example:** See [Code Example 5-22](#).

### 5.6.4.2 *write*

#### Call(s):

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

#### Arguments:

**Table 5-57. HI Driver Arguments - *write***

|                 |           |                                                                                                                                                                                                                                                             |
|-----------------|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | HI device descriptor returned by <i>open</i> call                                                                                                                                                                                                           |
| <i>pBuffer</i>  | <i>in</i> | Pointer to user buffer. The <i>write</i> function type casts <i>pBuffer</i> to be a ( <i>char *</i> ) if HI_SAMPLE_SIZE is HI_BYTE (default). The <i>write</i> function type casts <i>pBuffer</i> to be a ( <i>Word16 *</i> ) if HI_SAMPLE_SIZE is HI_WORD. |
| <i>NBytes</i>   | <i>in</i> | Number of bytes to write                                                                                                                                                                                                                                    |

**Description:** The *write* call writes data from the user buffer to the HI device. The *write* function can be configured to handle a user's buffer as bytes or words using the HI\_SAMPLE\_SIZE #define in *appconfig.h*. The default is bytes. When configured for bytes, the byte of data is written to the least significant byte of the 16-bit transmit register.

In blocking mode, *write* waits while all required data is transferred.

In non-blocking mode, *write* starts the transfer operation and returns control to the application. It does not wait while all data is transferred via HI.

The *write* function also enables the transmitter interrupt in order to get the transmitter started.

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

**Code Example:** See [Code Example 5-22](#).

### 5.6.4.3 *read*

**Call(s):**

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

**Arguments:**

**Table 5-58. HI Driver Arguments - *read***

|                 |           |                                                                                                                                                                                                                                         |
|-----------------|-----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | HI device descriptor returned by <i>open</i> call                                                                                                                                                                                       |
| <i>pBuffer</i>  | <i>in</i> | Pointer to user buffer. The <i>read</i> function type casts <i>pBuffer</i> to be a (char *) if HI_SAMPLE_SIZE is HI_BYTE (default). The <i>read</i> function type casts <i>pBuffer</i> to be a (Word16 *) if HI_SAMPLE_SIZE is HI_WORD. |
| <i>Size</i>     | <i>in</i> | Number of bytes to read                                                                                                                                                                                                                 |

**Description:** The *read* call reads data from the HI device into the user buffer. The *read* function can be configured to handle a user's buffer as bytes or words using the HI\_SAMPLE\_SIZE #define in *appconfig.h*. The default is bytes. When configured for bytes, the byte of data is read from the most significant byte of the 16-bit receive register.

In Blocking mode, *read* waits until all data has been received.

In Non-Blocking mode, *read* returns the actual data that has already been received from the HI device.

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

**Code Example:** None



5.6.4.4 close

Call(s):

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

Arguments:

Table 5-59. HI Driver Arguments - close

|          |    |                                            |
|----------|----|--------------------------------------------|
| FileDesc | in | HI device descriptor returned by open call |
|----------|----|--------------------------------------------|

Description: The close call disables the HI peripheral by disabling all HI interrupts.

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

Code Example: See Code Example 5-22.

### 5.6.4.5 *ioctl*

**Call(s):**

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

**Arguments:**

**Table 5-60. HI Driver Arguments - *ioctl***

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

**Table 5-61. HI *ioctl* Commands**

| Cmd                   | <i>pParams</i>                                 | Return | Result                                                                                                                                                                                                                           |
|-----------------------|------------------------------------------------|--------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| HI_DEVICE_RESET       | NULL                                           | None   | This command disables and then enables the Host Interface driver. See HI_DEVICE_ENABLE and HI_DEVICE_DISABLE                                                                                                                     |
| HI_DEVICE_DISABLE     | NULL                                           | None   | This command disables all Host Interface interrupts and clears any pending receiver interrupt.                                                                                                                                   |
| HI_DEVICE_ENABLE      | NULL                                           | None   | This command clears the transmit and receive buffers and re-enables the Host Interface Receiver interrupt.                                                                                                                       |
| HI_CALLBACK_RX        | void (*hiDataCallback)<br>(void)               | None   | This command installs the Read Completed Callback function. This function is called by the HI Driver when the receiver has received a number of bytes as defined by HI_READ_WATERMARK.                                           |
| HI_CALLBACK_TX        | void (*hiDataCallback)<br>(void)               | None   | This command installs the Write Completed Callback function. This function is called when the HI Driver has no more data to transmit.                                                                                            |
| HI_CALLBACK_EXCEPTION | void (*hiErrorCallback)<br>(UWord16 Exception) | None   | This command installs the Exception Callback function. This function is called when an exception is detected by the HI Driver. Use HI_GET_EXCEPTION to determine the exception.                                                  |
| HI_SET_RX_WATERMARK   | UWord16 length                                 | None   | This command sets the number of bytes (if Sample Size is in bytes) or words (if Sample Size is in words) that must be received prior to calling the Read Completed Callback function. The default for this is HI_READ_WATERMARK. |

Table 5-61. HI *ioctl* Commands (Continued)

| Cmd                 | pParams      | Return               | Result                                                                                                                                                                                                                              |
|---------------------|--------------|----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| HI_GET_STATUS       | NULL         | Status               | This command gets the status of the Host Interface driver. Valid return values are:<br>HI_STATUS_WRITE_INPROGRESS<br>HI_STATUS_EXCEPTION_EXIST                                                                                      |
| HI_GET_EXCEPTION    | NULL         | Exception            | This command gets the exception status of the Host Interface driver. Valid returns values are:<br>HI_EXCEPTION_RX_BUFFER_OVERFLOW                                                                                                   |
| HI_CMD_READ_CLEAR   | NULL         | None                 | This command disables the Host Interface Receiver interrupt, clears the Receiver's Buffer, and then re-enables the Host Interface Receiver interrupt.                                                                               |
| HI_CMD_WRITE_CLEAR  | NULL         | None                 | This command disables the Host Interface Transmitter interrupt and the clears the Transmitter's Buffer. Note that the Transmitter's interrupt is re-enabled during the next <i>write</i> call.                                      |
| HI_SET_HOST_FLAG    | Host Flag(s) | None                 | This command sets the Host Flag(s) specified by <i>pParams</i> . Note that only Host Flags 2 and 3 can be written to by the DSP. Valid parameters are:<br>HI_HOST_FLAG_2 or HI_HOST_FLAG_3 or<br>(HI_HOST_FLAG2   HI_HOST_FLAG_3)   |
| HI_CLEAR_HOST_FLAG  | Host Flag(s) | None                 | This command clears the Host Flag(s) specified by <i>pParams</i> . Note that only Host Flags 2 and 3 can be written to by the DSP. Valid parameters are:<br>HI_HOST_FLAG_2 or HI_HOST_FLAG_3 or<br>(HI_HOST_FLAG2   HI_HOST_FLAG_3) |
| HI_READ_HOST_FLAG_1 | None         | State of Host Flag 1 | This command reads Host Flag 1. Note that this flag cannot be written to by the DSP. This command returns "True" if Host Flag 1 is set and "False" if Host Flag 1 is clear.                                                         |
| HI_READ_HOST_FLAG_0 | None         | State of Host Flag 0 | This command reads Host Flag 0. Note that this flag cannot be written to by the DSP. This command returns "True" if Host Flag 0 is set and "False" if Host Flag 0 is clear.                                                         |

**Description:** The *ioctl* call changes HI device modes or reports HI driver status.

**Returns:** The *ioctl* call returns a type *UWord16*. This *UWord16* is dependent on the command passed in as an input parameter. [Table 5-61](#) lists these returns based on the specific command.

**Code Example:** [Code Example 5-22](#) sets up the HI to write data out the host interface.



**Code Example 5-22. HI Driver Usage**

```

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

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

 /* open host interface driver */
 HiDevice = open(BSP_DEVICE_NAME_HI, O_RDWR);

 Data = 0x0030;

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

 /* close host interface driver */
 close(HiDevice);
}
*****/

```

## 5.6.5 HI Device-Dependent 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.

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 *hiOpen*

**Call(s):**

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

**Arguments:**

**Table 5-62. HI Driver Arguments - *hiOpen***

|                 |            |                                                                                                              |
|-----------------|------------|--------------------------------------------------------------------------------------------------------------|
| <i>pName</i>    | <i>in</i>  | The HI device name; see <a href="#">Table 5-63</a>                                                           |
| <i>OFlag</i>    | <i>in</i>  | Open Mode Flags; specify O_RDWR<br>Use O_NONBLOCK flags for Non-Blocking mode                                |
| <i>FileDesc</i> | <i>out</i> | HI device descriptor returned by <i>hiOpen</i> call and used in <i>hiRead/hiWrite/hiIoctl</i> function calls |

**Table 5-63. HI Device Names for DSP5685x Chips**

| Chip                                         | Can Use            |
|----------------------------------------------|--------------------|
| DSP56853<br>DSP56854<br>DSP56857<br>DSP56858 | BSP_DEVICE_NAME_HI |
| DSP56855                                     | N/A                |

**Description:** The *hiOpen* call enables the HI peripheral for operations and returns a file descriptor used by the HI driver's *hiRead*, *hiWrite*, *hiClose*, and *hiIoctl* function calls. The HI device must be opened for *hiRead/hiWrite/hiIoctl* calls.

The *hiOpen* call initializes the internal memory buffers used by the HI driver. It configures the driver for blocking or non-blocking modes, as defined in the *OFlags* parameter. Finally, it enables the HI Receiver interrupts. It does not enable the HI transmitter interrupts, as this is done when the *hiWrite* function is called. The default mode for open, as defined in *config.h* and *config.c*, is the following:

Default Host Tx Buffer Size = 8

Default Host Rx Buffer Size = 8

Default Host Req Mode = HREQ and HACK selected

Default Host Strobe Mode = Single Strobe

Default Host Rx Callback = NULL

Default Host Tx Callback = NULL

Default Host Exception Callback = NULL

Default Host Read Watermark = 0

Default Host Sample Size = Byte Wide



**Returns:** The *hiOpen* call returns a type *types\_tHandle*. When the *hiOpen* call is successful, this *types\_tHandle* is a file descriptor for the HI device and is used by the *hiRead*, *hiWrite*, and *hiIoctl* functions. A "-1" is returned when the *hiOpen* call is not successful.

**Code Example:** See [Code Example 5-23](#).

### 5.6.5.2 *hiWrite*

#### Call(s):

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

#### Arguments:

**Table 5-64. HI Driver Arguments - *hiWrite***

|                 |           |                                                                                                                                                                                                                                                                 |
|-----------------|-----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | HI device descriptor returned by <i>hiOpen</i> call                                                                                                                                                                                                             |
| <i>pBuffer</i>  | <i>in</i> | Pointer to user buffer. The <i>hiWrite</i> function type casts <i>pBuffer</i> to be a ( <i>char *</i> ) if HI_SAMPLE_SIZE is HI_BYTE (default). The <i>hiWrite</i> function type casts <i>pBuffer</i> to be a ( <i>Word16 *</i> ) if HI_SAMPLE_SIZE is HI_WORD. |
| <i>NBytes</i>   | <i>in</i> | Number of bytes to write                                                                                                                                                                                                                                        |

**Description:** The *hiWrite* call writes data from the user buffer to the HI device. The *hiWrite* function can be configured to handle a user's buffer as bytes or words using the HI\_SAMPLE\_SIZE #define in *appconfig.h*. The default is bytes. When configured for bytes, the byte of data is written to the least significant byte of the 16-bit transmit register.

In blocking mode, *hiWrite* waits while all required data is transferred.

In non-blocking mode, *hiWrite* starts the transfer operation and returns control to the application. It does not wait while all data is transferred via HI.

The *hiWrite* function also enables the transmitter interrupt in order to get the transmitter started.

**Returns:** The *hiWrite* call returns a type *ssize\_t*, which is the number of transferred bytes.

**Code Example:** See [Code Example 5-23](#).

### 5.6.5.3 *hiRead*

**Call(s):**

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

**Arguments:**

**Table 5-65. HI Driver Arguments - *hiRead***

|                 |           |                                                                                                                                                                                                                                                               |
|-----------------|-----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | HI device descriptor returned by <i>hiOpen</i> call                                                                                                                                                                                                           |
| <i>pBuffer</i>  | <i>in</i> | Pointer to user buffer. The <i>hiRead</i> function type casts <i>pBuffer</i> to be a ( <i>char *</i> ) if HI_SAMPLE_SIZE is HI_BYTE (default). The <i>hiRead</i> function type casts <i>pBuffer</i> to be a ( <i>Word16 *</i> ) if HI_SAMPLE_SIZE is HI_WORD. |
| <i>Size</i>     | <i>in</i> | Number of bytes to read                                                                                                                                                                                                                                       |

**Description:** The *hiRead* call reads data from HI device into the user buffer. The *hiRead* function can be configured to handle a user's buffer as bytes or words using the HI\_SAMPLE\_SIZE #define in *appconfig.h*. The default is bytes. When configured for bytes, the byte of data is read from the most significant byte of the 16-bit receive register.

In Blocking mode, *hiRead* waits until all data has been received.

In Non-Blocking mode, *hiRead* returns the actual data that has already been received from the HI device.

**Returns:** The *hiRead* call returns a type *ssize\_t*, which is the number of received bytes.

**Code Example:** None



### 5.6.5.4 *hiClose*

**Call(s):**

```
int hiClose (types_tHandle FileDesc);
```

**Arguments:**

**Table 5-66. HI Driver Arguments - *hiClose***

|                 |           |                                                     |
|-----------------|-----------|-----------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | HI device descriptor returned by <i>hiOpen</i> call |
|-----------------|-----------|-----------------------------------------------------|

**Description:** The *hiClose* call disables the HI peripheral by disabling all HI interrupts.

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

**Code Example:** See [Code Example 5-23](#).

### 5.6.5.5 *hiioctl*

#### Call(s):

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

#### Arguments:

**Table 5-67. HI Driver Arguments - *hiioctl***

|                 |           |                                                                      |
|-----------------|-----------|----------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | HI device descriptor returned by <i>hiOpen</i> call                  |
| <i>Cmd</i>      | <i>in</i> | Commands found in <i>hi.h</i> which are used to manage the HI driver |
| <i>pParams</i>  | <i>in</i> | Pointer to <i>hiioctl</i> command parameter                          |
| <i>pName</i>    | <i>in</i> | The HI device name from <i>bsp.h</i> ; typically, BSP_DEVICE_NAME_HI |

**Table 5-68. HI *hiioctl* Commands**

| Cmd                   | <i>pParams</i>                              | Return | Result                                                                                                                                                                                 |
|-----------------------|---------------------------------------------|--------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| HI_DEVICE_RESET       | NULL                                        | None   | This command disables and then enables the Host Interface driver. See: HI_DEVICE_ENABLE and HI_DEVICE_DISABLE                                                                          |
| HI_DEVICE_DISABLE     | NULL                                        | None   | This command disables all Host Interface interrupts and clears any pending receiver interrupt.                                                                                         |
| HI_DEVICE_ENABLE      | NULL                                        | None   | This command clears the transmit and receive buffers and re-enables the Host Interface Receiver interrupt.                                                                             |
| HI_CALLBACK_RX        | void (*hiDataCallback) (void)               | None   | This command installs the Read Completed Callback function. This function is called by the HI Driver when the receiver has received a number of bytes as defined by HI_READ_WATERMARK. |
| HI_CALLBACK_TX        | void (*hiDataCallback) (void)               | None   | This command installs the Write Completed Callback function. This function is called when the HI Driver has no more data to transmit.                                                  |
| HI_CALLBACK_EXCEPTION | void (*hiErrorCallback) (UWord16 Exception) | None   | This command installs the Exception Callback function. This function is called when an exception is detected by the HI Driver. Use HI_GET_EXCEPTION to determine the exception.        |



Table 5-68. HI *hiloctl* Commands (Continued)

| Cmd                 | <i>pParams</i> | Return               | Result                                                                                                                                                                                                                              |
|---------------------|----------------|----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| HI_SET_RX_WATERMARK | UWord16 length | None                 | This command sets the number of bytes (if Sample Size is in bytes) or words (if Sample Size is in words) that must be received prior to calling the Read Completed Callback function. The default for this is HI_READ_WATERMARK.    |
| HI_GET_STATUS       | NULL           | Status               | This command gets the status of the Host Interface driver. Valid return values are:<br>HI_STATUS_WRITE_INPROGRESS<br>HI_STATUS_EXCEPTION_EXIST                                                                                      |
| HI_GET_EXCEPTION    | NULL           | Exception            | This command gets the exception status of the Host Interface driver. Valid returns values are:<br>HI_EXCEPTION_RX_BUFFER_OVERFLOW                                                                                                   |
| HI_CMD_READ_CLEAR   | NULL           | None                 | This command disables the Host Interface Receiver interrupt, clears the Receiver's Buffer, and then re-enables the Host Interface Receiver interrupt.                                                                               |
| HI_CMD_WRITE_CLEAR  | NULL           | None                 | This command disables the Host Interface Transmitter interrupt and the clears the Transmitter's Buffer. Note that the Transmitter's interrupt is re-enabled during the next <i>write</i> call.                                      |
| HI_SET_HOST_FLAG    | Host Flag(s)   | None                 | This command sets the Host Flag(s) specified by <i>pParams</i> . Note that only Host Flags 2 and 3 can be written to by the DSP. Valid parameters are:<br>HI_HOST_FLAG_2 or HI_HOST_FLAG_3 or<br>(HI_HOST_FLAG2   HI_HOST_FLAG_3)   |
| HI_CLEAR_HOST_FLAG  | Host Flag(s)   | None                 | This command clears the Host Flag(s) specified by <i>pParams</i> . Note that only Host Flags 2 and 3 can be written to by the DSP. Valid parameters are:<br>HI_HOST_FLAG_2 or HI_HOST_FLAG_3 or<br>(HI_HOST_FLAG2   HI_HOST_FLAG_3) |
| HI_READ_HOST_FLAG_1 | None           | State of Host Flag 1 | This command reads Host Flag 1. Note that this flag cannot be written to by the DSP. This command returns "True" if Host Flag 1 is set and "False" if Host Flag 1 is clear.                                                         |
| HI_READ_HOST_FLAG_0 | None           | State of Host Flag 0 | This command reads Host Flag 0. Note that this flag cannot be written to by the DSP. This command returns "True" if Host Flag 0 is set and "False" if Host Flag 0 is clear.                                                         |

**Description:** The *hiioctl* call changes HI device modes or gives HI driver status.

**Returns:** The *hiioctl* call returns a type *UWord16*. This *UWord16* is dependent on the command passed in as an input parameter. [Table 5-68](#) lists these returns based on the specific command.

**Code Example:** [Code Example 5-23](#) sets up the HI to write data out the host interface.

#### Code Example 5-23. HI Driver Usage

---

```

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

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

 /* open host interface driver */
 HiDevice = hiOpen(BSP_DEVICE_NAME_HI, O_RDWR);

 Data = 0x0030;

 /* Send one byte via HI */
 hiWrite(HiDevice, (void *)&Data, sizeof(char));

 /* close host interface driver */
 hiClose(HiDevice);
}
*****/

```

## 5.6.6 HI Driver Tests

The HI Driver Tests verify correct operation of the HI Driver provided by the SDK. Currently, there are no stand-alone tests for verifying the HI Driver.

## 5.6.7 HI Driver Demos

### 5.6.7.1 Host Interface (HI) Demo

The HI Demo interacts with a desktop/laptop PC via the PC parallel port connection using the DomainTec EPP Rev D card. The HI demo tests the data transfer between the Host Processor and the DSP. The data transfer is handled by two data channels, one used to transfer data from the host to the DSP and the other one to transfer data in the opposite direction, from the DSP to the host.

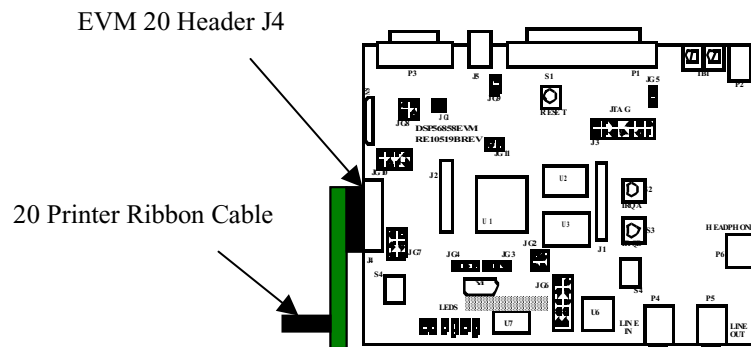
#### 5.6.7.1.1 Set-up for Host Interface Demo

##### EVM Jumper Settings:

Use default settings as shown in the **DSP56858 Evaluation Module User's Manual**.

##### EPP Card:

- Connect the Host Interface of the EPP Card (EPP Rev D) to the Host Interface of the DSP56858EVM (J4) as shown in [Figure 5-1](#)
- Connect the Ribbon Cable to the EPP Card (EPP Rev D) as shown in [Figure 5-1](#)



### 5.6.7.1.2 Host Interface Demo

The Host Interface Demo requires a PC application that communicates to the EVM using the parallel port. CodeWarrior also requires the use of the parallel port for downloading and debugging the application on the EVM. The following procedures describes how one PC or two PCs can be used to execute the Host Interface Demo. If you are using one PC, see

[Section 5.6.7.1.3](#). If you are using two PCs, see [Section 5.6.7.1.4](#).

### 5.6.7.1.3 Procedure for Host Interface Demo Using One PC

- Using CodeWarrior, open:  
`...\nos\applications\bsp\HostDemo\HostDemo.mcp`
- Build the project; this will generate an S-Record
- Program the Host Interface Demo S-Record into the Data Flash of the DSP56858EVM
  - See the Second Stage SPI Bootloader Demo in [Section 3.3.2.1.1](#) to learn how to program the Data Flash
  - Instead of using the S-Record for the Timers application, use the S-Record for the Host Interface Demo
- Start the Host Interface Demo on the DSP by removing and then reapplying power to the DSP56858EVM
- Continue with the Host Interface Demo as described in [Section 5.6.7.1.5](#)

### 5.6.7.1.4 Procedure for Host Interface Demo if Using Two PCs

- Connect the parallel port of the PC running CodeWarrior to the DSP56858EVM
- Using CodeWarrior, open:  
`...\nos\applications\bsp\HostDemo\HostDemo.mcp`
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the demo
- Continue with the Host Interface Demo as described in [Section 5.6.7.1.5](#)

### 5.6.7.1.5 Procedure for Host Interface Demo

- Connect the Ribbon Cable to the parallel port of the PC that will execute the Host Interface Test PC Application
- Run the *HostPortTest.exe* PC demo application, found at:  
`...\src\x86\win32\applications\HostPortInterface\DomainTecEPP\HostPortTest.exe`
- After executing *HostPortTest.exe*, you will see this menu:

```

Motorola DSP5685x Host Interface test code
Options
[0] Set the LPT Mode (EPP, Bidirectional supported)
[1] Toggle EVM LEDs <Turns ON and OFF EVM LEDs>
[2] Write a word
[3] Read a word
[4] Write a block of data
[5] Read a block of data
[6] Loopback mode
[7] Toggle Host Flags
[8] Quit

```

- Selecting option “[0] Set the LPT Mode (EPP, Bidirectional supported)” will allow you to change the PC Printer mode
  - Make sure that the PC System BIOS is configured to the Mode selected

- Selecting option “[1] Toggle EVM LEDS <Turns ON and OFF EVM LEDs>” will toggle all but the first red LED
  - If they are off, they will turn on
  - If they are on, they will turn off
- Selecting option “[2] Write a word” will prompt the user to write a character to the DSP
- Selecting option “[3] Read a word” will read the character written previously to the DSP and display it on the screen
- Selecting option “[4] Write a block of data” will write 127 bytes to the DSP
- Selecting option “[5] Read a block of data” will read the previously written 127 bytes from the DSP and compare them for accuracy
- Selecting option “[6] Loopback mode” will write and read 127 bytes, one byte at a time
- Selecting option “[7] Toggle Host Flags” will toggle the Host Flag0 (HF0), and Host Flag1 (HF1)
  - All LEDs will turn on for HF0, and all LEDs will turn off for HF1
- Selecting option “[8] Quit” will stop the *HostPortTest.exe* application code

### NOTE:

- The EPP Rev D board is provided by Domain Technology (DomainTec.com). It is provided ONLY for testing DSP5685x Host Port Interface
- The drivers have been tested only with Windows 2000
- The Bidirectional Mode rate achieved is 56Kbytes/second; for EPP mode, a rate of 277Kbytes/second is achieved
- Motorola is not responsible for Domain Technology EPP Rev D, and its software drivers. Consult with Domain Technology about any support related to this EPP Rev D board and software.

## 5.7 Programmable Chip Select (PCS) Driver

The DSP5685x 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.

### 5.7.1 Initialization

The Chip Selects and EMI (BCR) may be initialized automatically by the SDK start-up code by adding 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 DSP56858EVM. Chip Select 0 is mapped to both bytes of Program space address range \$000000-\$7FFFFFFF; Chip Select 1 is mapped to the lower byte of Data space address range \$000000-\$7FFFFFFF; and Chip Select 2 is mapped to the upper byte of Data space address range \$000000-\$7FFFFFFF. 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_RWSS PCS_0_WAIT_STATES
#define PCS2_RWSH 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_RWSS PCS_0_WAIT_STATES
#define PCS3_RWSH 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
```





## 5.7.2 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.



### 5.7.2.1 *pcsInit()*

**Call(s):**

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

**Arguments:**

**Table 5-69. PCS Argument Values**

| Argument           | Possible value                                                                                                                                                                                                                                                                                          |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>ChipSelect</i>  | Identifies the Chip Select to be initialized<br>PCS_CS0<br>PCS_CS1<br>PCS_CS2<br>PCS_CS3                                                                                                                                                                                                                |
| <i>BaseAddress</i> | Start address, which must be a multiple of the selected block size.<br>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<br>PCS_BLKSIZE_4K<br>PCS_BLKSIZE_8K<br>PCS_BLKSIZE_16K<br>PCS_BLKSIZE_32K<br>PCS_BLKSIZE_64K<br>PCS_BLKSIZE_128K<br>PCS_BLKSIZE_256K<br>PCS_BLKSIZE_512K<br>PCS_BLKSIZE_1M<br>PCS_BLKSIZE_2M<br>PCS_BLKSIZE_4M<br>PCS_BLKSIZE_8M<br>PCS_BLKSIZE_16M |
| <i>SpaceEnable</i> | Selects the type of memory (program, data, or both) used for asserting the appropriate strobe signal<br>PCS_DISABLE<br>PCS_DATA<br>PCS_PROGRAM<br>Note: To specify both data and program:<br>(PCS_DATA PCS_PROGRAM)                                                                                     |
| <i>RWEnable</i>    | Identifies the type of access required in order to assert the Chip Select<br>PCS_READ_WRITE_DISABLE<br>PCS_WRITE_ONLY<br>PCS_READ_ONLY<br>PCS_READ_WRITE_ENABLE                                                                                                                                         |

**Table 5-69. PCS Argument Values**

|                                  |                                                                                                                                                                                                                                                    |
|----------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>ByteEnable</i>                | Selects which Byte(s) to access when accessing the device controlled by the Chip Select<br>PCS_NO_BYTE_ENABLE<br>PCS_LOWER_BYTE_ENABLE<br>PCS_UPPER_BYTE_ENABLE<br>PCS_BOTH_BYTES_ENABLE                                                           |
| <i>ReadWaitStates</i>            | Specifies the number of additional system clocks to delay for a read access to the selected memory<br>PCS_0_WAIT_STATES<br>PCS_1_WAIT_STATE<br>PCS_2_WAIT_STATES<br>... PCS_(3-29)_WAIT_STATES<br>PCS_30_WAIT_STATES                               |
| <i>WriteWaitStates</i>           | Specifies the number of additional system clocks to delay for a write access to the selected memory<br>PCS_0_WAIT_STATES<br>PCS_1_WAIT_STATE<br>PCS_2_WAIT_STATES<br>... PCS_(3-29)_WAIT_STATES<br>PCS_30_WAIT_STATES                              |
| <i>WriteWaitStatesSetupDelay</i> | Specifies the number of system clocks to delay prior to asserting write signal<br>PCS_0_WAIT_STATES<br>PCS_1_WAIT_STATE<br>PCS_2_WAIT_STATES<br>... PCS_(3-29)_WAIT_STATES<br>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<br>PCS_0_WAIT_STATES<br>PCS_1_WAIT_STATE<br>PCS_2_WAIT_STATES<br>... PCS_(3-29)_WAIT_STATES<br>PCS_30_WAIT_STATES |
| <i>ReadWaitStatesSetupDelay</i>  | Specifies the number of system clocks to delay prior to asserting read signal<br>PCS_0_WAIT_STATES<br>PCS_1_WAIT_STATE<br>PCS_2_WAIT_STATES<br>... PCS_(3-29)_WAIT_STATES<br>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<br>PCS_0_WAIT_STATES<br>PCS_1_WAIT_STATE<br>PCS_2_WAIT_STATES<br>... PCS_(3-29)_WAIT_STATES<br>PCS_30_WAIT_STATES  |

**Table 5-69. PCS Argument Values**

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

**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 5-24](#) be added to the *appconfig.h* file.

**Code Example 5-24. *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_RSSH 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 5-25](#) 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 5-25. 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);
}
```

## 5.7.2.2 *pcsInitEMI()*

### Call(s):

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

### Arguments:

**Table 5-70. 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.<br>PCS_DRV_DISABLED<br>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.<br>PCS_0_WAIT_STATES<br>PCS_1_WAIT_STATE<br>PCS_2_WAIT_STATES<br>... PCS_(3-6)_WAIT_STATES<br>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.<br>PCS_0_WAIT_STATES<br>PCS_1_WAIT_STATE<br>PCS_2_WAIT_STATES<br>... PCS_(3-29)_WAIT_STATES<br>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<br>PCS_0_WAIT_STATES<br>PCS_1_WAIT_STATE<br>PCS_2_WAIT_STATES<br>... PCS_(3-29)_WAIT_STATES<br>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 5-26](#) be added to the *appconfig.h* file.

**Code Example 5-26. *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 5-27](#) will enable the Drive Bit and set the base wait states to 0.

**Code Example 5-27. *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);
}
```

## 5.8 Quad Timer Driver

### 5.8.1 Introduction

The DSP5685x 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 14** of the **DSP5685x 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.

### 5.8.2 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 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 **Chapter 5** in the **SDK Programmer's Guide** for the Timer Services interface definition and specification.

If the Quad Timer is being used in the absence of the Timer Services, the default is to configure all four timers for direct use with the Quad Timer driver as shown in [Code Example 5-28](#).

#### Code Example 5-28. 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 should be overridden by the user specifying the exact timers required by the application. This will minimize the code size of the driver for the case where all timers are not required. The list of available timers is in [Table 5-71](#). For 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 */
```

**Table 5-71. 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 |

### 5.8.3 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 5-72. Quad Timer Data Structure Members**

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

Table 5-72. 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<br>Possible values are:<br><i>qtPastCompare</i> ,<br><i>qtUntilCompare</i>                                                                                                                                                                   | Count length of control register                      |
| CountDirection       | qt_eCountDirection | Normal or reverse direction<br>Possible values are:<br><i>qtUp</i> ,<br><i>qtDown</i>                                                                                                                                                                                                                              | Count direction of control register                   |
| OutputMode           | qt_eOutputMode     | Output mode<br>Possible values are:<br><i>qtAssertWhileActive</i> ,<br><i>qtAssertOnCompare</i> ,<br><i>qtDeassertOnCompare</i> ,<br><i>qtToggleOnCompare</i> ,<br><i>qtToggleUsingAlternateCompare</i> ,<br><i>qtDeassertOnSecondary</i> ,<br><i>qtDeassertOnCounterRollover</i> ,<br><i>qtAssertOnGatedClock</i> | Output mode of control register                       |
| OutputPolarity       | qt_ePolarity       | Output polarity<br>Possible values are:<br><i>qtNormal</i> ,<br><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:<br><i>qtDisabled</i> ,<br><i>qtRisingEdge</i> ,<br><i>qtFallingEdge</i> ,<br><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                                    |
| InitialLoadValue     | UInt16             | Initial load value                                                                                                                                                                                                                                                                                                 | Counter register                                      |

**Table 5-72. Quad Timer Data Structure Members (Continued)**

| Parameter           | Type         | Description                 | Quad Timer Register Reference |
|---------------------|--------------|-----------------------------|-------------------------------|
| 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 14** of the **DSP5685x 16-Bit Digital Signal Processor User's Manual**.

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

- Callback on compare
- Callback on overflow
- 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 5-29](#), where the user has defined a callback function, *CallbackOnCompare0*, with input argument 0.

### Code Example 5-29. 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,
 /* CompareValue2 = */ 0,
```



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



## 5.8.4 Quad Timer 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.

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.

### 5.8.4.1 *open*

**Call(s):**

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

**Arguments:**

**Table 5-73. Quad Driver Arguments - *open***

|                 |            |                                                                                                                                                                     |
|-----------------|------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>pName</i>    | <i>in</i>  | The quad timer device name:<br>BSP_DEVICE_NAME_QUAD_TIMER_A_0<br>BSP_DEVICE_NAME_QUAD_TIMER_A_1<br>BSP_DEVICE_NAME_QUAD_TIMER_A_2<br>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 and 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 5-29](#).

### 5.8.4.2 *close*

**Call(s):**

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

**Arguments:**

**Table 5-74. 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 5-29](#).



### 5.8.4.3 *ioctl*

Call(s):

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

**Table 5-75. Quad Timer Driver Arguments - *ioctl***

|                 |    |                                                                                                   |
|-----------------|----|---------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | in | Quad Timer device descriptor returned by <i>open</i> call and used in <i>ioctl</i> function calls |
| <i>Cmd</i>      | in | Commands found in <i>quadtimer.h</i> which are used to manage the Quad Timer driver               |
| <i>pParams</i>  | in | Pointer to <i>ioctl</i> command parameter; see <a href="#">Table 5-76</a>                         |

**Table 5-76. 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:<br>True = force to Logic One<br>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 5-76. 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 *<br>(addr of<br>output value) | None    | Returns Quad Timer Input Clock Frequency in<br>the input pointer |
| QT_SET_INPUT_CAPTURE_MODE | qt_eCapture<br>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 5-76** lists these returns based on the specific command.

**Code Example 5-29** sets up Quad Timer A Channel Zero to turn the Green LED on and off.

**Code Example 5-30. 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;

}

/\*\*\*\*\*/



## 5.8.5 Quad Timer 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.

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.

### 5.8.5.1 *qtOpen*

Call(s):

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

**Table 5-77. Quad Timer Driver Arguments - *open***

|                |     |                                                                                                                                                             |
|----------------|-----|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>pName</i>   | in  | The Quad Timer device name; see <a href="#">Table 5-50</a>                                                                                                  |
| <i>OFlag</i>   | in  | General parameter to configure the Quad Timer driver; however, this parameter is not currently used                                                         |
| <i>pParams</i> | int | 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 5-30](#).



## 5.8.5.2 *qtClose*

Call(s):

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

**Table 5-78. Quad Timer Driver Arguments - *close***

|                 |    |                                                                                                       |
|-----------------|----|-------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | in | Quad Timer device descriptor returned by <i>qtOpen</i> call and used in <i>qtIoctl</i> 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 5-30](#).

### 5.8.5.3 *qtIoctl*

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

**Table 5-79. Quad Timer Driver Arguments - *ioctl***

|                      |       |                                                                                                                                                              |
|----------------------|-------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i>      | in    | Quad Timer device descriptor returned by <i>open</i> call and used in <i>ioctl</i> function calls                                                            |
| <i>Cmd</i>           | in    | Commands found in <i>quadtimer.h</i> which are used to manage the Quad Timer driver; see <a href="#">Section 5.8.4.3</a> for list of <i>qtIoctl</i> Commands |
| <i>pParams</i>       | in    | Pointer to <i>qtIoctl</i> command parameter; see <a href="#">Table 5-76</a>                                                                                  |
| <i>bspDeviceName</i> | void* | 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*. This *UWord16* is dependent on the command passed in as an input parameter. [Table 5-76](#) lists these returns based on the specific command.

**Code Example:** [Code Example 5-30](#) shows an example of the use of the device-dependent API Quad Timer driver.

#### Code Example 5-31. Device-Dependent API 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);
}
*****/

```



## 5.8.6 Quad Timer Driver Tests

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

## 5.8.7 Quad Timer Driver Demos

### 5.8.7.1 Quad Timer Flashing LEDs Demo

The Quad Timer flashing LEDs demonstrate the 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.

#### 5.8.7.1.1 Set-up for Quad Timer Flashing LEDs Demo

##### EVM Jumper Settings:

Use default settings as shown in the **DSP56858 Evaluation Module User's Manual**.

#### 5.8.7.1.2 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 ...)

## 5.9 SCI Driver

### 5.9.1 Introduction

The DSP5685x processor has two SCI devices, SCI0 and SCI1. This SCI device allows full-duplex, asynchronous serial communication between the DSP, a host computer, or other devices.

### 5.9.2 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 5-80](#).

**Table 5-80. SCI Configuration Items for *appconfig.h***

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

Table 5-80. SCI Configuration Items for *appconfig.h* (Continued)

| SCI #define                  | Explanation and Default Value                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SCI0_TX_CALLBACK_FUNCTION    | <p>A user function for the SCI driver to call when the SCI0 TX buffer has no more samples to transmit. The user function must be of type <i>tpSciDataCallback</i>:</p> <pre>typedef void (*tpSciDataCallback)(void);</pre> <p>Default = NULL</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| SCI0_RX_CALLBACK_FUNCTION    | <p>A user function for the SCI driver to call when SCI0 has received at least SCI0_READ_WATERMARK samples in its internal RX buffer. The user function must be of type <i>tpSciDataCallback</i></p> <pre>typedef void (*tpSciDataCallback)(void);</pre> <p>Default = NULL</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| SCI0_ERROR_CALLBACK_FUNCTION | <p>A user function for the SCI driver to call when SCI0 has detected an exception condition. The user function must be of type <i>tpSciErrorCallback</i></p> <pre>typedef void (*tpSciErrorCallback)(UWord16 Exception);</pre> <p>The parameters to the callback function are:</p> <p><i>Exception</i> - The actual exception that occurred</p> <ul style="list-style-type: none"> <li>0x0800 - SCI_EXCEPTION_OVERRUN_ERROR</li> <li>0x0400 - SCI_EXCEPTION_NOISE_ERROR</li> <li>0x0200 - SCI_EXCEPTION_FRAME_ERROR</li> <li>0x0100 - SCI_EXCEPTION_PARITY_ERROR</li> <li>0x0008 - SCI_EXCEPTION_BUFFER_OVERFLOW</li> <li>0x0004 - SCI_EXCEPTION_ADDRESS_MARK</li> <li>0x0002 - SCI_EXCEPTION_BREAK_SYMBOL</li> </ul> <p>Default = NULL</p> |
| SCI0_READ_WATERMARK          | <p>The number of bytes that SCI0 must receive before calling the user function specified by SCI0_RX_CALLBACK_FUNCTION. The value specified must be less than SCI0_RECEIVE_BUFFER_LENGTH.</p> <p>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.</p> <p>Default = 0.</p>                                                                                                                                                      |
| SCI1_LOOP_MODE               | <p>Loopback mode of SCI1 peripheral</p> <p>Valid values are:</p> <ul style="list-style-type: none"> <li>SCI_CNTL_LOOPBACK_DISABLED (default)</li> <li>SCI_CNTL_LOOPBACK_ENABLED</li> <li>SCI_CNTL_LOOPBACK_SINGLE_WIRE_ENABLED</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |

Table 5-80. SCI Configuration Items for *appconfig.h* (Continued)

| SCI #define                | Explanation and Default Value                                                                                                                                                                                                                                                            |
|----------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SCI1_DATA_MODE             | <p>Selects the SCI1 Data Mode</p> <p>Valid values are:<br/> SCI_8_DATA_BITS_NO_PARITY (default)<br/> SCI_7_DATA_BITS_WITH_ODD_PARITY<br/> SCI_7_DATA_BITS_WITH_EVEN_PARITY<br/> SCI_8_DATA_BITS_WITH_ODD_PARITY<br/> SCI_8_DATA_BITS_WITH_EVEN_PARITY<br/> SCI_9_DATA_BITS_NO_PARITY</p> |
| SCI1_POLARITY              | <p>Selects the polarity of the SCI1 data</p> <p>Valid values are:<br/> SCI_DONT_INVERT_ANY_BITS (default)<br/> SCI_INVERT_ALL_BITS</p>                                                                                                                                                   |
| SCI1_CONTROL_REG           | <p>Allows the user to define the SCI1 Control Register</p> <p>Default = (SCI0_DATA_MODE   SCI0_POLARITY)</p>                                                                                                                                                                             |
| SCI1_BAUD_RATE             | <p>Selects the baud rate of the SCI peripheral</p> <p>Default = 9600</p>                                                                                                                                                                                                                 |
| SCI1_SEND_BUFFER_LENGTH    | <p>Fifo buffer size for write operation for SCI1</p> <p>Default = 8</p>                                                                                                                                                                                                                  |
| SCI1_RECEIVE_BUFFER_LENGTH | <p>Fifo buffer size for read operation for SCI1</p> <p>Default = 8</p>                                                                                                                                                                                                                   |
| SCI1_TX_CALLBACK_FUNCTION  | <p>A user function for the SCI driver to call when the SCI1 TX buffer has no more samples to transmit. The user function must be of type <i>tpSciDataCallback</i>:</p> <pre>typedef void (*tpSciDataCallback)(void);</pre> <p>Default = NULL</p>                                         |
| SCI1_RX_CALLBACK_FUNCTION  | <p>A user function for the SCI driver to call when SCI1 has received at least SCI1_READ_WATERMARK samples in its internal RX buffer. The user function must be of type <i>tpSciDataCallback</i></p> <pre>typedef void (*tpSciDataCallback)(void);</pre> <p>Default = NULL</p>            |

Table 5-80. SCI Configuration Items for *appconfig.h* (Continued)

| SCI #define                  | Explanation and Default Value                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SCI1_ERROR_CALLBACK_FUNCTION | <p>A user function for the SCI driver to call when it has detected an exception condition with SCI1. The user function must be of type <i>tpSciErrorCallback</i></p> <pre>typedef void (*tpSciErrorCallback)(UWord16 Exception);</pre> <p>The parameters to the callback function are:<br/> <i>Exception</i> - The actual exception that occurred<br/> 0x0800 - SCI_EXCEPTION_OVERRUN_ERROR<br/> 0x0400 - SCI_EXCEPTION_NOISE_ERROR<br/> 0x0200 - SCI_EXCEPTION_FRAME_ERROR<br/> 0x0100 - SCI_EXCEPTION_PARITY_ERROR<br/> 0x0008 - SCI_EXCEPTION_BUFFER_OVERFLOW<br/> 0x0004 - SCI_EXCEPTION_ADDRESS_MARK<br/> 0x0002 - SCI_EXCEPTION_BREAK_SYMBOL</p> <p>Default = NULL</p> |
| SCI1_READ_WATERMARK          | <p>The number of bytes that SCI1 must receive before calling the user function specified by SCI1_RX_CALLBACK_FUNCTION. The value specified must be less than SCI1_RECEIVE_BUFFER_LENGTH.</p> <p>Increasing the SCI1_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.</p> <p>Default = 0.</p>                                                                                       |

### 5.9.3 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



## 5.9.4 SCI 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.

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.

### 5.9.4.1 *open*

**Call(s):**

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

**Arguments:**
**Table 5-81. SCI Driver Arguments - *open***

|                 |            |                                                                                                 |
|-----------------|------------|-------------------------------------------------------------------------------------------------|
| <i>pName</i>    | <i>in</i>  | The SCI device name; see <a href="#">Table 5-82</a>                                             |
| <i>OFlag</i>    | <i>in</i>  | Open Mode Flags; specify O_RDWR<br>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 5-82. SCI Device Names for DSP5685x Chips**

| Chip                                                                                        | Can Use                                        |
|---------------------------------------------------------------------------------------------|------------------------------------------------|
| <b>DSP56853</b><br><b>DSP56854</b><br><b>DSP56855</b><br><b>DSP56857</b><br><b>DSP56858</b> | BSP_DEVICE_NAME_SCI_0<br>BSP_DEVICE_NAME_SCI_1 |

**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 must be opened for *read/write* calls. The *open* function establishes the handle for the SCI driver. It initializes the internal memory buffers used by the SCI driver. It configures the driver for blocking or non-blocking modes, as defined in the *OFlag* parameter. Finally, it enables the SCI Receiver interrupts. It does not enable the SCI transmitter interrupts as this occurs when the *write* function is called. Default mode for *open*, as defined in *config.h* and *config.c*, is:

Default Baud Rate = 9600bps

Default Data Mode = 8-bit with no parity

Default Data Polarity = Not Inverted

Default Loopback = Not Enabled

Default SCI Rx Callback = NULL

Default SCI Tx Callback = NULL

Default SCI Exception Callback = NULL

Default SCI Read Watermark = 0

**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 5-31](#).



## 5.9.4.2 **write**

### Call(s):

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

### Arguments:

**Table 5-83. 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 <i>write</i> function type casts <i>pBuffer</i> to be a ( <i>char *</i> ). |
| <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.

In blocking mode, *write* waits while all required data is transferred.

In non-blocking mode, *write* starts the transfer operation and returns control to the application. It does not wait while all data is transferred via SCI.

The *write* function also enables the transmitter interrupt in order to get the transmitter started.

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

**Code Example:** See [Code Example 5-31](#).

### 5.9.4.3 *read*

**Call(s):**

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

**Arguments:**

**Table 5-84. 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 <i>read</i> function type casts <i>pBuffer</i> to be a ( <i>char *</i> ) |
| <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.

In Blocking mode, *read* waits until all data has been received.

In Non-Blocking mode, *read* returns the actual data that has already been received from the SCI device.

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

**Code Example:** None



### 5.9.4.4 *close*

**Call(s):**

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

**Arguments:**

**Table 5-85. 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 by disabling all SCI interrupts and disabling the SCI transmitter and receiver.

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

**Code Example:** See [Code Example 5-31](#).

### 5.9.4.5 *ioctl*

**Call(s):**

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

**Arguments:**

**Table 5-86. 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 5-87. SCI *ioctl* Commands**

| <b>Cmd</b>             | <b><i>pParams</i></b>                                 | <b>Return</b> | <b>Result</b>                                                                                                                                                                                                    |
|------------------------|-------------------------------------------------------|---------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SCI_DEVICE_RESET       | sci_sConfig *                                         | None          | This command disables, then enables, the SCI driver<br><br>See:<br>SCI_DEVICE_ENABLE and<br>SCI_DEVICE_DISABLE.                                                                                                  |
| SCI_DEVICE_DISABLE     | NULL                                                  | None          | This command disables all SCI interrupts and clears any pending receiver interrupt                                                                                                                               |
| SCI_DEVICE_ENABLE      | NULL                                                  | None          | This command clears the transmit and receive buffers and re-enables the SCI Receiver interrupt                                                                                                                   |
| SCI_CALLBACK_RX        | void<br>(*sciDataCallback)<br>(void)                  | None          | This command installs the Read Completed Callback function. This function is called by the SCI Driver when the receiver has received a number of bytes as defined by SCI0_READ_WATERMARK or SCI1_READ_WATERMARK. |
| SCI_CALLBACK_TX        | void<br>(*sciDataCallback)<br>(void)                  | None          | This command installs the Write Completed Callback function; this function is called when the SCI Driver has no more data to transmit                                                                            |
| SCI_CALLBACK_EXCEPTION | void<br>(*sciErrorCallback)<br>(UWord16<br>Exception) | None          | This command installs the Exception Callback function; this function is called when an exception is detected by the SCI Driver. Use SCI_GET_EXCEPTION to determine the exception.                                |
| SCI_SET_RX_WATERMARK   | UWord16 length                                        | None          | This command sets the number of bytes that must be received prior to calling the Read Completed Callback function<br>Default = SCI0_READ_WATERMARK or SCI1_READ_WATERMARK.                                       |

Table 5-87. SCI ioctl Commands (Continued)

| Cmd                              | pParams             | Return    | Result                                                                                                                                                                                                                                                                                                         |
|----------------------------------|---------------------|-----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SCI_GET_STATUS                   | NULL                | Status    | This command gets the status of the SCI driver.<br><br>Valid return values are:<br>SCI_STATUS_WRITE_INPROGRESS<br>SCI_STATUS_EXCEPTION_EXIST                                                                                                                                                                   |
| SCI_GET_EXCEPTION                | NULL                | Exception | This command gets the exception status of the SCI driver.<br><br>Valid returns values are:<br>SCI_EXCEPTION_OVERRUN_ERROR<br>SCI_EXCEPTION_NOISE_ERROR<br>SCI_EXCEPTION_FRAME_ERROR<br>SCI_EXCEPTION_PARITY_ERROR<br>SCI_EXCEPTION_BUFFER_OVERFLOW<br>SCI_EXCEPTION_ADDRESS_MARK<br>SCI_EXCEPTION_BREAK_SYMBOL |
| SCI_CMD_SEND_BREAK               | NULL                | None      | This command sends a break symbol via SCI                                                                                                                                                                                                                                                                      |
| SCI_CMD_WAIT                     | NULL                | None      | This command puts the SCI device in wait state                                                                                                                                                                                                                                                                 |
| SCI_CMD_WAKEUP                   | NULL                | None      | This command wakes up the SCI device from wait mode                                                                                                                                                                                                                                                            |
| SCI_CMD_READ_CLEAR               | NULL                | None      | This command disables the SCI Receiver interrupt, clears the Receiver's Buffer, then re-enables the SCI Receiver interrupt                                                                                                                                                                                     |
| SCI_CMD_WRITE_CLEAR              | NULL                | None      | This command disables the SCI Transmitter interrupt, then clears the Transmitter's Buffer; note that the Transmitter's interrupt is re-enabled during the next <i>write</i> call                                                                                                                               |
| SCI_CHANGE_BAUD_RATE             | UWord16<br>BaudRate | None      | This command changes the SCI Baud Rate                                                                                                                                                                                                                                                                         |
| SCI_CHANGE_DATA_MODE             | UWord16<br>DataMode | None      | This command changes the Data Mode; the following parameters are valid:<br>SCI_7_DATA_BITS_WITH_ODD_PARITY<br>SCI_7_DATA_BITS_WITH_EVEN_PARITY<br>SCI_8_DATA_BITS_NO_PARITY<br>SCI_8_DATA_BITS_WITH_ODD_PARITY<br>SCI_8_DATA_BITS_WITH_EVEN_PARITY<br>SCI_9_DATA_BITS_NO_PARITY                                |
| SCI_ENABLE_LOOPBACK              | NULL                | None      | This command enables Loopback mode                                                                                                                                                                                                                                                                             |
| SCI_DISABLE_LOOPBACK             | NULL                | None      | This command disables Loopback mode                                                                                                                                                                                                                                                                            |
| SCI_ENABLE_LOOPBACK_SINGLE_WIRE  | NULL                | None      | This command enables Single Wire Loopback mode                                                                                                                                                                                                                                                                 |
| SCI_DISABLE_LOOPBACK_SINGLE_WIRE | NULL                | None      | This command disables Single Wire Loopback mode                                                                                                                                                                                                                                                                |

**Description:** The *ioctl* call changes SCI device modes or reports 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 5-87](#) lists these returns based on the specific command.

**Code Example:** [Code Example 5-31](#) sets up the SCI to write data out the serial port.

#### Code Example 5-32. 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);
}
*****/

```

---



## 5.9.5 SCI 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.

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.

### 5.9.5.1 *sciOpen*

**Call(s):**

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

**Arguments:**
**Table 5-88. SCI Driver Arguments - *sciOpen***

|                 |            |                                                                                                    |
|-----------------|------------|----------------------------------------------------------------------------------------------------|
| <i>pName</i>    | <i>in</i>  | The SCI device name; see <a href="#">Table 5-89</a>                                                |
| <i>OFlag</i>    | <i>in</i>  | Open Mode Flags; specify O_RDWR<br>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>read/write</i> function calls |

**Table 5-89. SCI Device Names for DSP5685x Chips**

| Chip                                                                                        | Can Use                                        |
|---------------------------------------------------------------------------------------------|------------------------------------------------|
| <b>DSP56853</b><br><b>DSP56854</b><br><b>DSP56855</b><br><b>DSP56857</b><br><b>DSP56858</b> | BSP_DEVICE_NAME_SCI_0<br>BSP_DEVICE_NAME_SCI_1 |

**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 needs to be opened for *sciRead/sciWrite/sciIoctl* calls. The *sciOpen* function establishes the handle for the SCI driver. It initializes the internal memory buffers used by the SCI driver. It configures the driver for blocking or non-blocking modes, as defined in the *OFlag* parameter. Finally, it enables the SCI Receiver interrupts. It does not enable the SCI transmitter interrupts as this occurs when the *sciWrite* function is called. Default mode for *open*, as defined in *config.h* and *config.c*, is:

Default Baud Rate = 9600bps

Default Data Mode = 8-bit with no parity

Default Data Polarity = Not Inverted

Default Loopback = Not Enabled

Default SCI Rx Callback = NULL

Default SCI Tx Callback = NULL

Default SCI Exception Callback = NULL

Default SCI Read Watermark = 0

**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. When the *sciOpen* call is not successful, a "-1" is returned.

**Code Example:** See [Code Example 5-32](#).



## 5.9.5.2 *sciWrite*

### Call(s):

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

### Arguments:

**Table 5-90. 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 <i>write</i> function type casts <i>pBuffer</i> to be a ( <i>char *</i> ) |
| <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.

In blocking mode, *sciWrite* waits while all required data is transferred.

In non-blocking mode, *sciWrite* starts the transfer operation and returns control to the application. It does not wait while all data is transferred via SCI.

The *sciWrite* function also enables the transmitter interrupt in order to get the transmitter started.

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

**Code Example:** See [Code Example 5-32](#).

### 5.9.5.3 *sciRead*

**Call(s):**

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

**Arguments:**

**Table 5-91. 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 <i>read</i> function type casts <i>pBuffer</i> to be a ( <i>char *</i> ). |
| <i>Size</i>     | <i>in</i> | Number of bytes to read                                                                               |

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

In Blocking mode, *sciRead* waits until all data has been received.

In Non-Blocking mode, *sciRead* returns the actual data that has already been received from the SCI device.

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

**Code Example:** None



## 5.9.5.4 *sciClose*

### Call(s):

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

### Arguments:

**Table 5-92. 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 by disabling all SCI interrupts and disabling the SCI transmitter and receiver.

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

**Code Example:** See [Code Example 5-32](#).

### 5.9.5.5 *sciloctl*

#### Call(s):

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

#### Arguments:

**Table 5-93. SCI Driver Arguments - *sciloctl***

|                 |           |                                                                                                   |
|-----------------|-----------|---------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | SCI device descriptor returned by <i>sciOpen</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>sciloctl</i> command parameter                                                      |
| <i>pName</i>    | <i>in</i> | The SCI device name from <i>bsp.h</i> ; typically, BSP_DEVICE_NAME_SCI_0 or BSP_DEVICE_NAME_SCI_1 |

**Table 5-94. SCI *sciloctl* Commands**

| Cmd                    | <i>pParams</i>                               | Return | Result                                                                                                                                                                                                           |
|------------------------|----------------------------------------------|--------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SCI_DEVICE_RESET       | sci_sConfig *                                | None   | This command disables, then enables, the SCI driver; see: SCI_DEVICE_ENABLE and SCI_DEVICE_DISABLE.                                                                                                              |
| SCI_DEVICE_DISABLE     | NULL                                         | None   | This command disables all SCI interrupts and clears any pending receiver interrupt                                                                                                                               |
| SCI_DEVICE_ENABLE      | NULL                                         | None   | This command clears the transmit and receive buffers and re-enables the SCI Receiver interrupt                                                                                                                   |
| SCI_CALLBACK_RX        | void (*sciDataCallback) (void)               | None   | This command installs the Read Completed Callback function; this function is called by the SCI Driver when the receiver has received a number of bytes as defined by SCI0_READ_WATERMARK or SCI1_READ_WATERMARK. |
| SCI_CALLBACK_TX        | void (*sciDataCallback) (void)               | None   | This command installs the Write Completed Callback function; this function is called when the SCI Driver has no more data to transmit                                                                            |
| SCI_CALLBACK_EXCEPTION | void (*sciErrorCallback) (UWord16 Exception) | None   | This command installs the Exception Callback function; this function is called when an exception is detected by the SCI Driver. Use SCI_GET_EXCEPTION to determine the exception.                                |
| SCI_SET_RX_WATERMARK   | UWord16 length                               | None   | This command sets the number of bytes that must be received prior to calling the Read Completed Callback function. Default = SCI0_READ_WATERMARK or SCI1_READ_WATERMARK.                                         |

Table 5-94. SCI *sciloctl* Commands (Continued)

| Cmd                              | pParams             | Return    | Result                                                                                                                                                                                                                                                                                                        |
|----------------------------------|---------------------|-----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SCI_GET_STATUS                   | NULL                | Status    | This command gets the status of the SCI driver<br><br>Valid return values are:<br>SCI_STATUS_WRITE_INPROGRESS<br>SCI_STATUS_EXCEPTION_EXIST                                                                                                                                                                   |
| SCI_GET_EXCEPTION                | NULL                | Exception | This command gets the exception status of the SCI driver<br><br>Valid returns values are:<br>SCI_EXCEPTION_OVERRUN_ERROR<br>SCI_EXCEPTION_NOISE_ERROR<br>SCI_EXCEPTION_FRAME_ERROR<br>SCI_EXCEPTION_PARITY_ERROR<br>SCI_EXCEPTION_BUFFER_OVERFLOW<br>SCI_EXCEPTION_ADDRESS_MARK<br>SCI_EXCEPTION_BREAK_SYMBOL |
| SCI_CMD_SEND_BREAK               | NULL                | None      | This command sends a break symbol via SCI                                                                                                                                                                                                                                                                     |
| SCI_CMD_WAIT                     | NULL                | None      | This command puts the SCI device in wait state                                                                                                                                                                                                                                                                |
| SCI_CMD_WAKEUP                   | NULL                | None      | This command wakes up the SCI device from wait mode                                                                                                                                                                                                                                                           |
| SCI_CMD_READ_CLEAR               | NULL                | None      | This command disables the SCI Receiver interrupt, clears the Receiver's Buffer, then re-enables the SCI Receiver interrupt                                                                                                                                                                                    |
| SCI_CMD_WRITE_CLEAR              | NULL                | None      | This command disables the SCI Transmitter interrupt and the clears the Transmitter's Buffer; note that the Transmitter's interrupt is re-enabled during the next <i>sciWrite</i> call                                                                                                                         |
| SCI_CHANGE_BAUD_RATE             | UWord16<br>BaudRate | None      | This command changes the SCI Baud Rate                                                                                                                                                                                                                                                                        |
| SCI_CHANGE_DATA_MODE             | UWord16<br>DataMode | None      | This command changes the Data Mode. The following parameters are valid:<br>SCI_7_DATA_BITS_WITH_ODD_PARITY<br>SCI_7_DATA_BITS_WITH_EVEN_PARITY<br>SCI_8_DATA_BITS_NO_PARITY<br>SCI_8_DATA_BITS_WITH_ODD_PARITY<br>SCI_8_DATA_BITS_WITH_EVEN_PARITY<br>SCI_9_DATA_BITS_NO_PARITY                               |
| SCI_ENABLE_LOOPBACK              | NULL                | None      | This command enables Loopback mode                                                                                                                                                                                                                                                                            |
| SCI_DISABLE_LOOPBACK             | NULL                | None      | This command disables Loopback mode                                                                                                                                                                                                                                                                           |
| SCI_ENABLE_LOOPBACK_SINGLE_WIRE  | NULL                | None      | This command enables Single Wire Loopback mode                                                                                                                                                                                                                                                                |
| SCI_DISABLE_LOOPBACK_SINGLE_WIRE | NULL                | None      | This command disables Single Wire Loopback mode                                                                                                                                                                                                                                                               |

**Description:** The *sciIoctl* call changes SCI device modes or reports SCI driver status.

**Returns:** The *sciIoctl* call returns a type *UWord16*. This *UWord16* is dependent on the command passed in as an input parameter. [Table 5-94](#) lists these returns based on the specific command.

**Code Example:** [Code Example 5-32](#) sets up the SCI to write data out the serial port.

#### Code Example 5-33. 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, O_RDWR);

 Data = 0x0030;

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

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

```

## 5.9.6 SCI Driver Tests

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

### 5.9.6.1 SCI Loopback Test

The SCI loopback test, which runs on the DSP56858 EVM, configures the SCI port to loop data back internally and uses the SCI Driver to write and read samples from the SCI port.

#### 5.9.6.1.1 Set-up for SCI Loopback Test

##### EVM Jumper Settings:

Use default settings as shown in the **DSP56858 Evaluation Module User's Manual**.

#### 5.9.6.1.2 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

## 5.9.7 SCI Driver Demos

### 5.9.7.1 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.

#### 5.9.7.1.1 Set-up for SCI Serial Port Demo

##### EVM Jumper Settings:

Use default settings as shown in the **DSP56858 Evaluation Module User's Manual**.

#### 5.9.7.1.2 Procedure for SCI Serial Port Demo

- Using CodeWarrior, open:  
...\\nos\\applications\\bsp\\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, 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 DSP56858EVM board by selecting the data space and address range, then selecting *READ*
  - When *READ* is selected, a command is sent to the DSP56858 processor via the serial interface
  - After the application, running on the DSP56858, processes this command, a response message with the requested data is returned to the PC
  - As soon as the GUI application running on the PC receives the response, the data for the requested memory location is displayed

### 5.9.7.2 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 9600bps.

#### 5.9.7.2.1 Set-up for SCI Echo Demo

##### EVM Jumper Settings:

Use default settings as shown in the **DSP56858 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    |

#### 5.9.7.2.2 Procedure for SCI Echo Demo

- Using CodeWarrior, open:  
`...\nos\applications\bsp\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"

## 5.10 SPI Driver

### 5.10.1 Introduction

The DSP5685x 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 which provides the lowest-level software layer interfacing hardware to software.

### 5.10.2 Initialization

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



### 5.10.3 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 preprocessor statement `INCLUDE_SPI` 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 5-95](#).

**Table 5-95. SPI Configuration Items for *appconfig.h***

| SPI #define                              | Explanation and Default Value                                                                                                                                                                                                        |
|------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>SPI_NON_BLOCKED</code>             | Blocking/Non-Blocking mode is selected statically<br><br>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 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 MPIO C pin 3 to deassert the S                                                                                                                               |
| <code>SPI_SEND_BUFFER_LENGTH</code>      | FIFO transmission buffer length for Non-Blocking mode<br><br>Default = 32                                                                                                                                                            |
| <code>SPI_RECEIVE_BUFFER_LENGTH</code>   | FIFO reception buffer length for Non-Blocking mode<br><br>Default = 32                                                                                                                                                               |
| <code>SPI_BAUD_RATE_DIVISOR</code>       | Divides SPI clock by value<br><br>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.<br><br>Example: If the data size is 8 bits, then this element must be set to 7. A value of 0 is invalid.<br><br>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);
*****/

```



## 5.10.4 SPI 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.

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.

### 5.10.4.1 *open*

**Call(s):**

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

**Arguments:**
**Table 5-96. 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 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 SPI device and is 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 5-34](#).

### 5.10.4.2 **write**

**Call(s):**

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

**Arguments:**

**Table 5-97. 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 5-34](#).

### 5.10.4.3 *read*

**Call(s):**

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

**Arguments:**

**Table 5-98. 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 when using the SPI device in Master mode.

**Code Example:** None

## 5.10.4.4 *close*

### Call(s):

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

### Arguments:

**Table 5-99. SPI Driver Arguments - *close***

|                 |           |                                                                                                       |
|-----------------|-----------|-------------------------------------------------------------------------------------------------------|
| <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 |
|-----------------|-----------|-------------------------------------------------------------------------------------------------------|

**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 5-34](#) sets up the SPI as a master device and writes data out of the SPI peripheral.

#### Code Example 5-34. 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 5-35. 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);
******/
```

### 5.10.4.5 *ioctl*

Call(s):

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

Arguments:

**Table 5-100. 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 5-101. 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 5-101. SPI *ioctl* Commands (Continued)**

| Cmd                        | <i>pParams</i> | Return | Result                                                                                                                                                                                                                       |
|----------------------------|----------------|--------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SPI_CLOCK_PHASE_SET        | None           | None   | SS pin of the slave SPI module need not 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.<br>Example: 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 5-35](#) sets up the SPI as a master device and makes the LSB transmit first.

#### Code Example 5-36. SPI Driver Usage

```

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

types_tHandleSerialMaster;

/* 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);
*****/

```

## 5.10.5 SPI 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.

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.

## 5.10.5.1 *spiOpen*

### Call(s):

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

### Arguments:

**Table 5-102. 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 5-34](#).

### 5.10.5.2 *spiBlockedWrite, spiNonBlockedWrite*

Call(s):

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

Arguments:

**Table 5-103. 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 5-34](#).

### 5.10.5.3 *spiBlockedRead, spiNonBlockedRead*

Call(s):

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

Arguments:

**Table 5-104. 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

### 5.10.5.4 *spiBlockedClose, spiNonBlockedClose*

**Call(s):**

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

**Arguments:**

**Table 5-105. 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 5-34](#) sets up the SPI as a master device and writes data out of the SPI peripheral.

#### Code Example 5-37. 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);
*****/

```

### 5.10.5.5 *spioctl*

**Call(s):**

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

**Arguments:**

**Table 5-106. 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 5-107. 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                                                                         |

**Table 5-107. SPI *spioctl* Commands (Continued)**

| Cmd                        | <i>pParams</i> | Return | Result                                                                                                                                                                                                                       |
|----------------------------|----------------|--------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 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                                                                                                                                                                                     |
| SPI_CLOCK_PHASE_SET        | None           | None   | SS pin of the slave SPI module need not 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.<br>Example: 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 5-35](#) sets up the SPI as a master device and makes the LSB transmit first.

#### Code Example 5-38. SPI Driver Usage

```

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

types_tHandleSerialMaster;

/* 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);
*****/

```



## 5.10.6 SPI Driver Applications

The Serial Data Flash application uses the SPI Driver. Please see [Section 6.9.6](#) for more information.

## 5.10.7 SPI Driver Tests

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

### 5.10.7.1 SPI Test

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

#### 5.10.7.1.1 Set-up for SPI Test

##### EVM Jumper Settings:

Use default settings as shown in the **DSP56858 Evaluation Module User's Manual**.

#### 5.10.7.1.2 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

## 5.11 Timer Driver

### 5.11.1 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 Public Interface.

### 5.11.2 Timer Driver Tests

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

## 5.11.3 Timer Driver Demos

### 5.11.3.1 Timer Driver Flashing LEDs Demo

This demo exercises Timer0, Timer1, and Timer2 peripherals located in the DSP56858 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, with the net effect 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.

#### 5.11.3.1.1 Set-up for Timer Driver Flashing LEDs Demo

##### EVM Jumper Settings:

Use default settings as shown in the **DSP56858 Evaluation Module User's Manual**.

#### 5.11.3.1.2 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

## 5.12 Time of Day (TOD) Driver

### 5.12.1 Introduction

The Time of Day (TOD) device is implemented as a sequence of counters to keep track of elapsed time. The TOD is accurate to  $\pm 1$  second. Please refer to the **DSP5685x 16-Bit Digital Signal Processor User's Manual** for more information on the TOD device.

### 5.12.2 Initialization

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

```
#define INCLUDE_TIME_OF_DAY
```

Defining this preprocessor variable directs the SDK to include the TOD driver in the executable image. It enables the default configuration of the TOD device located in the *config.h* file. These defaults may be overwritten in the *appconfig.h* file.

The TOD device driver relies on the defined constant `BSP_OSCILLATOR_FREQ` to be set to the frequency of the external oscillator. The default value in the *config.h* file is 4MHz and may be overwritten in the *appconfig.h* file to match the user's target system. This defined constant is used to determine the TOD scaler value which guarantees the required 1Hz input clock to the TOD device. The scaler value is stored in the TOD global variable, *TodClockScaler*, located in the SDK configuration constants file, *const.c*.

### 5.12.3 TOD Device-Independent API Specification

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

Please refer to the Timer Services section in the **SDK Programmer's Guide** for definitions of the TOD Device-Independent Public Interface. For an example of this interface, please see [Code Example 5-41](#).

### 5.12.4 TOD Device-Dependent API Specification

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

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

#### Device-Dependent Public Interface Function(s):

```

/*****
 * types_tHandle todOpen(const char * pName, int Flags, void * pParams);
 * int todEnableCallbacks(types_tHandle FileDesc, struct sigevent *);
 * int todSetAlarm(types_tHandle FileDesc, struct itimerspec * pValue);
 * void todGetTime (struct tm * pGetTime);
 * int todClose(types_tHandle FileDesc);
 * int todIoctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams);
 *****/

```

#### Public Data Structure(s):

The following public data structures are found in *timespec.h*:

```

struct timespec
{
 time_t tv_sec; /* seconds */
 long tv_nsec; /* nanoseconds (0 - 999,999,999) */
};

```

The following public data structures are found in *time.h*:

```

struct itimerspec
{
 struct timespec it_interval; /* timer period (reload value) */
 struct timespec it_value; /* timer expiration */
};

```

The following public data structures are found in *ctime*, as part of the Metrowerks runtime library:

```
struct tm
{
 int tm_sec; /* seconds after the minute [0, 59] */
 int tm_min; /* minutes after the hour [0, 59] */
 int tm_hour; /* hour since midnight [0, 23] */
 int tm_mday; /* day of the month [1, 31] */
 int tm_mon; /* months since January [0, 11] */
 int tm_year; /* years since 1900 */
 int tm_wday; /* days since Sunday [0, 6] */
 int tm_yday; /* days since January 1 [0, 365] */
 int tm_isdst; /* flag for daylight savings time */
};
```

The following public data structure and union is found in *signal.h*

```
union sigval
{
 int sival_int;
 void * sival_ptr;
};
struct sigevent
{
 int sigev_signo;
 union sigval sigev_value;
 int sigev_notify;
 void (*sigev_notify_function)(union sigval);
};
```



## 5.12.5 TOD Device-Independent API Specification

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.

Please refer to the Timer Services section in the **SDK Programmer's Guide** for specification of the Device-Independent Public Interface.



## 5.12.6 TOD Device-Dependent API Specification

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.

### 5.12.6.1 *todOpen*

**Call(s):**

```
types_tHandle todOpen(const char * pName, int Flags, void * pParams);
```

**Arguments:**

**Table 5-108. TOD Driver Arguments - *todOpen***

|                |           |                                                                                                  |
|----------------|-----------|--------------------------------------------------------------------------------------------------|
| <i>pName</i>   | <i>in</i> | The TOD device name; use BSP_DEVICE_TIME_OF_DAY                                                  |
| <i>Flags</i>   | <i>in</i> | Open Flags, which are ignored; use NULL                                                          |
| <i>pParams</i> | <i>in</i> | A pointer to structure <i>timespec</i> ; contains the Initial Time in seconds for the TOD device |

**Description:** The *todOpen* function will initialize the TOD clock counter registers and configure the clock scaler to cause a 1Hz input clock to the TOD peripheral. The clock counter registers are configured based on the seconds value passed in the *pParams* argument. It is recommended that the seconds value be determined by the *mktime()* function, as shown in [Code Example 5-40](#). The clock scaler variable is initialized in *const.c* to:

*TodClockScaler* = BSP\_OSCILLATOR\_FREQ/128

This implementation will automatically ensure the correct scaler value, since the oscillator frequency changes as long as the user sets the BSP\_OSCILLATOR\_FREQ to the value used by the target system.

**Returns:** The *todOpen* call returns a type *types\_tHandle*. When the *todOpen* call is successful, this *types\_tHandle* is a file descriptor for the TOD device and is used by the TOD driver's *todSetAlarm*, *todIoctl*, *todEnableCallbacks*, and *todClose* functions. A "-1" is returned when the *todOpen* call is not successful.

**Code Example:** See [Code Example 5-40](#).

### 5.12.6.2 *todEnableCallbacks*

**Call(s):**

```
int todEnableCallbacks(types_tHandle FileDesc, struct sigevent * pCallbacks);
```

**Arguments:**

**Table 5-109. TOD Driver Arguments - *todEnableCallbacks***

|                   |           |                                                                                                                              |
|-------------------|-----------|------------------------------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i>   | <i>in</i> | TOD device descriptor returned by <i>todOpen</i> call                                                                        |
| <i>pCallbacks</i> | <i>in</i> | A pointer to structure <i>sigevent</i> ; contains the interrupt the user wants to enable and the callback for that interrupt |

**Description:** The *todEnableCallbacks* function enables a specific TOD interrupt and the corresponding callback function for that interrupt. The *sigev\_signo* element of the *sigevent* structure can be set to TOD\_ALARM\_INTERRUPT or TOD\_ONE\_SECOND\_INTERRUPT. The *sigev\_notify\_function* element of the *sigevent* structure is a pointer to the user's callback function. The *union sigval sigev\_value* of the *sigevent* structure is an argument of the user's callback function.

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

**Code Example:** See [Code Example 5-40](#).



### 5.12.6.3 *todSetAlarm*

Call(s):

```
int todSetAlarm(types_tHandle FileDesc, struct itimerspec * pValue);
```

Arguments:

**Table 5-110. TOD Driver Arguments - *todSetAlarm***

|                 |           |                                                                                                                                                                                                                         |
|-----------------|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | TOD device descriptor returned by <i>todopen</i> call                                                                                                                                                                   |
| <i>pValue</i>   | <i>in</i> | <p>Pointer to structure <i>itimerspec</i>; contains the first TOD alarm value and TOD repeating alarm value, both in seconds</p> <p><b>Note:</b> The First Alarm Value must be set for a time greater than 1 second</p> |

**Description:** The *todSetAlarm* call configures the TOD's seconds, minutes, hours, and days alarm registers with an alarm time. This function allows the setting of two types of alarms, which are specified in the structure *itimerspec*. These alarm values should be set relative to the initial time set in the *todOpen* call.

The first element, *it\_value.tv\_sec*, specifies an alarm that will only occur once. This alarm will occur relative to the next unit of time higher than the alarm value. If the current time of the TOD has passed this alarm value, the first alarm will not occur until the next unit of time higher than the alarm value has incremented. For example, if the current time is 11 seconds when the first alarm time of 10 seconds is set by this function, then the first alarm will not occur until the current time is equal to 1 minute and 10 seconds. Specifying a value of zero will disable the alarm. Typically, this alarm is used to cause an interrupt on a specific date and time. For portability reasons, it is recommended that the *mktime()* function be used to determine this seconds value based on the specified date and time.

The second element, *it\_interval.tv\_sec*, specifies an alarm that will occur repeatedly after the first alarm has occurred. This alarm will occur repeatedly each next unit of time higher than the alarm value and may be disabled by specifying a value of zero. Since this alarm is used to cause a recurring interrupt, the time interval should be limited to a 24 hour period (e.g., *it\_interval.tv\_sec* < 86400). Specifying a time within this range will cause a recurring interrupt at least once a day.

In [Code Example 5-39](#), the single alarm is set to 10 seconds and the repeating alarm is set to 5 seconds. Assuming that these values are set when the current time is equal to zero, the first alarm will occur when the current time is 10 seconds. Alarms will then occur repeatedly 5 seconds after each minute; see [Figure 5-2](#).

**Code Example 5-39. *todSetAlarm***

```
struct itimerspec SetAlarm;

SetAlarm.it_value.tv_sec = 10; /* Single alarm value 10 seconds*/
SetAlarm.it_interval.tv_sec = 5; /* Repeating alarm value 5 seconds*/
```

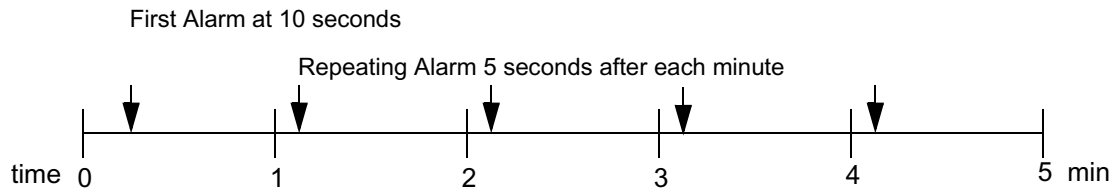


**Figure 5-2. TOD Alarm**

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

**Code Example:** See [Code Example 5-40](#).



### 5.12.6.4 *todGetTime*

Call(s):

```
void todGetTime (struct tm * pGetTime);
```

Arguments:

Table 5-111. TOD Driver Arguments - *todGetTime*

|                 |           |                                                                                                            |
|-----------------|-----------|------------------------------------------------------------------------------------------------------------|
| <i>pGetTime</i> | <i>in</i> | Pointer to structure <i>tm</i> ; the <i>todGetTime</i> function fills this structure with the current time |
|-----------------|-----------|------------------------------------------------------------------------------------------------------------|

**Description:** The *todGetTime* call obtains the current time of the TOD device and puts this current time into the *tm* structure passed into the function.

**Returns:** The *todGetTime* call returns a pointer to the *tm* structure passed in, now filled with the current time.

**Code Example:** See [Code Example 5-40](#).



5.12.6.5 *todClose*

Call(s):

```
int todClose(types_tHandle FileDesc);
```

Arguments:

Table 5-112. TOD Driver Arguments - *todClose*

|                 |           |                                                       |
|-----------------|-----------|-------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | TOD device descriptor returned by <i>todOpen</i> call |
|-----------------|-----------|-------------------------------------------------------|

**Description:** The *todClose* call closes the TOD peripheral, disables all TOD alarms, and releases the file descriptor handle.

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

**Code Example:** See [Code Example 5-40](#).

### 5.12.6.6 *todloctl*

Call(s):

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

Arguments:

**Table 5-113. TOD Driver Arguments - *todloctl***

|                |           |                                                                           |
|----------------|-----------|---------------------------------------------------------------------------|
| <i>TodFd</i>   | <i>in</i> | TOD file device descriptor returned by <i>todOpen</i> call                |
| <i>Cmd</i>     | <i>in</i> | Commands found in <i>tod.h</i> which are used to modify the TOD registers |
| <i>pParams</i> | <i>in</i> | See <a href="#">Table 5-114</a> , <i>todloctl</i> Commands                |

**Table 5-114. *todloctl* Commands**

| Cmd                          | <i>pParams</i>                      | Return | Result                         |
|------------------------------|-------------------------------------|--------|--------------------------------|
| TOD_ENABLE                   | None                                | None   | Enables TOD operation          |
| TOD_ALLOW_WRITE_TO_REGISTERS | None                                | None   | Allows writes to TOD registers |
| TOD_ENABLE_ALARM_IRQ         | None                                | None   | TOD Alarm IRQ enable           |
| TOD_DISABLE_ALARM_IRQ        | None                                | None   | TOD Alarm IRQ disable          |
| TOD_ENABLE_ONE_SEC_IRQ       | None                                | None   | TOD One Second IRQ enable      |
| TOD_DISABLE_ONE_SEC_IRQ      | None                                | None   | TOD One Second IRQ disable     |
| TOD_ENABLE_SEC_ALARM         | None                                | None   | TOD Seconds Alarm enable       |
| TOD_DISABLE_SEC_ALARM        | None                                | None   | TOD Seconds Alarm disable      |
| TOD_ENABLE_MIN_ALARM         | None                                | None   | TOD Minutes Alarm enable       |
| TOD_DISABLE_MIN_ALARM        | None                                | None   | TOD Minutes Alarm disable      |
| TOD_ENABLE_HR_ALARM          | None                                | None   | TOD Hours Alarm enable         |
| TOD_DISABLE_HR_ALARM         | None                                | None   | TOD Hours Alarm disable        |
| TOD_ENABLE_DAY_ALARM         | None                                | None   | TOD Days Alarm enable          |
| TOD_DISABLE_DAY_ALARM        | None                                | None   | TOD Days Alarm disable         |
| TOD_CLEAR_ALARM_IRQ          | None                                | None   | TOD Alarm IRQ clear            |
| TOD_CLEAR_ONE_SEC_IRQ        | None                                | None   | TOD One Second IRQ clear       |
| TOD_LOAD_CLOCK_SCALER        | UWord16 which includes scaler value | None   | Scaler to produce 1Hz clock    |
| TOD_LOAD_SECS                | None                                | None   | Loads Seconds register         |

**Table 5-114. *todloctl* Commands (Continued)**

| Cmd                            | pParams                                                           | Return | Result                          |
|--------------------------------|-------------------------------------------------------------------|--------|---------------------------------|
| TOD_LOAD_SECS_ALARM            | UWord16 which includes seconds value                              | none   | Loads Seconds Alarm register    |
| TOD_LOAD_MINS                  | UWord16 which includes minutes value                              | none   | Loads Minutes register          |
| TOD_LOAD_MINS_ALARM            | UWord16 which includes minutes alarm value                        | none   | Loads Minutes Alarm register    |
| TOD_LOAD_HRS                   | UWord16 which includes hours value                                | none   | Loads Hours register            |
| TOD_LOAD_HRS_ALARM             | UWord16 which includes hours alarm value                          | none   | Loads Hours Alarm register      |
| TOD_LOAD_DAYS                  | UWord16 which includes days value                                 | none   | Loads Days register             |
| TOD_LOAD_DAYS_ALARM            | UWord16 which includes days alarm value                           | none   | Loads Days Alarm register       |
| TOD_READ_SECS                  | None                                                              | int    | Reads Seconds register          |
| TOD_READ_MINS                  | None                                                              | int    | Reads Minutes register          |
| TOD_READ_HRS                   | None                                                              | int    | Reads Hours register            |
| TOD_READ_DAYS                  | None                                                              | int    | Reads Days register             |
| TOD_CONFIGURE_CONTROL_REGISTER | UWord16 which includes configuration of TOD control register bits | none   | Configures TOD Control register |
| TOD_READ_CONTROL_REGISTER      | None                                                              | int    | Reads Control register          |

**Description:** The *todloctl* call changes TOD device modes.

**Returns:** See [Table 5-114](#), *todloctl* Commands.

**Code Example:** [Code Example 5-40](#) sets up the TOD to alarm after 10 seconds with no interval alarm set.

## Code Example 5-40. TOD Device-Dependent API Driver Usage

```

/*****
#include "port.h"
#include "bsp.h"
#include "tod.h"
#include "stdio.h"

#define TOD_ALARM 10
#define TOD_CALLBACK_ARG_VALUE 6

static bool bTodDoneFlag = false;
static struct tm GetTime;

static void TodInterrupt(union sigval sigev_value)
{
 union sigval TestValue = sigev_value; /* This demonstrates callback arg
*/
 todGetTime(&GetTime);
 bTodDoneFlag = true;
}

void main (void)
{
 types_tHandle TodFD;
 time_t InitialTime;
 struct sigevent TodCallBack;
 struct itimerspec SetAlarm;
 struct tm Time;

 /* Set TOD initial time to Monday Jan. 1, 2002 00:00:00 */
 Time.tm_sec = 0; /* seconds after the minute [0,59] */
 Time.tm_min = 0; /* minutes after the hour [0,59] */
 Time.tm_hour = 0; /* hour since midnight [0,23] */
 Time.tm_mday = 1; /* day of the month [1,31] */
 Time.tm_mon = 0; /* months since January [0,11] */
 Time.tm_year = 102; /* years since 1900 [71,136] */
 Time.tm_wday = 1; /* days since Sunday [0,6] */
 Time.tm_yday = 0; /* days since January 1 [0,365] */
 Time.tm_isdst = 0; /* flag for daylight savings time [0,-1] */

 InitialTime = mktime(&Time);

 TodFD = todOpen(BSP_DEVICE_TIME_OF_DAY, 0, &InitialTime);

 /* Set TOD callback for alarm interrupt */
 TodCallBack.sigev_signo = TOD_ALARM_INTERRUPT;
 TodCallBack.sigev_notify_function = TodInterrupt;
 TodCallBack.sigev_value.sival_int = TOD_CALLBACK_ARG_VALUE;

 todEnableCallBacks(TodFD, &TodCallBack);

 /* Enable TOD device */
 todIoctl(TodFD, TOD_ENABLE, 0);

 /* Set Alarm */
 SetAlarm.it_value.tv_sec = 10; /* Alarm will go off after 10 seconds */
 SetAlarm.it_interval.tv_sec = 0; /* The reload value being set to zero
 indicates that only one alarm interrupt
 will be handled */

```

```

todSetAlarm(TodFD, &SetAlarm);

/* Enable Alarm */
todIoctl(TodFD, TOD_ENABLE_ALARM_IRQ, 0);

while(1)
{
 if(bTodDoneFlag == true)
 {
 archDisableInt();
 printf("Alarm occurred: %d/%d/%d %d:%d:%d\n", (GetTime.tm_mon+1), \
 (GetTime.tm_mday), \
 (GetTime.tm_year+1900), \
 (GetTime.tm_hour), \
 (GetTime.tm_min), \
 (GetTime.tm_sec));

 todClose(TodFD);
 break;
 }
}
/*****

```

**Code Example 5-41** utilizes the high-level interface to set up the TOD to alarm after 10 seconds with no interval alarm set.

#### Code Example 5-41. TOD Device-Independent API Driver Usage

```

/*****
#include "port.h"
#include "bsp.h"
#include "tod.h"
#include "stdio.h"

#define FIRST_ALARM 10

static bool bTodDoneFlag = false;
static struct timespec GetClockTime;

static void AlarmInterrupt(union sigval);

static void AlarmInterrupt(union sigval)
{
 clock_gettime(CLOCK_TOD, &GetClockTime);
 bTodDoneFlag = true;
}

void main (void)
{
 int TodFD;
 struct timespec InitialTime;
 struct sigevent TodCallBack;
 struct sigevent TimerEvent;

```



```

struct itimerspec SetAlarm;
timer_t TimerID;
struct tm Time;

/* Set TOD initial time to Monday Jan. 1, 2002 00:00:00 */
Time.tm_sec = 0; /* seconds after the minute [0,59] */
Time.tm_min = 0; /* minutes after the hour [0,59] */
Time.tm_hour = 0; /* hour since midnight [0,23] */
Time.tm_mday = 1; /* day of the month [1,31] */
Time.tm_mon = 0; /* months since January [0,11] */
Time.tm_year = 102; /* years since 1900 [71,136] */
Time.tm_wday = 1; /* days since Sunday [0,6] */
Time.tm_yday = 0; /* days since January 1 [0,365] */
Time.tm_isdst = 0; /* flag for daylight savings time [0,-1] */

InitialTime.tv_sec = mktime(&Time);

/* Set TOD initial time */
clock_settime(CLOCK_TOD, &InitialTime);

/* Install TOD interrupt and corresponding callback */
TimerEvent.sigev_signo = TOD_ALARM_INTERRUPT;
TimerEvent.sigev_notify_function = AlarmInterrupt;
timer_create (CLOCK_TOD, &TimerEvent, &TimerID);

/* Set up TOD alarm */
SetAlarm.it_value.tv_sec = FIRST_ALARM;
SetAlarm.it_interval.tv_sec = 0;
timer_settime(TimerID, 0, &SetAlarm, NULL);

while(1)
{
 if(bTodDoneFlag == true)
 {
 /* Convert seconds to tm structure */
 Time = *(localtime(&GetClockTime.tv_sec));

 /* Display alarm data */
 archDisableInt();
 printf("Alarm occurred: %d/%d/%d %d:%d:%d\n", (Time.tm_mon+1), \
 (Time.tm_mday), \
 (Time.tm_year+1900), \
 (Time.tm_hour), \
 (Time.tm_min), \
 (Time.tm_sec));

 archEnableInt();
 timer_delete(TimerID);
 break;
 }
}
}
/*****/

```

## 5.12.7 TOD Driver Application

An application source file demonstrates the Time Of Day (TOD) high-level interface. This demo application will configure the TOD clock, then set a first alarm and interval alarm. Progress information is written to the console window. Normal operation will cause a message to be



written to the console when the first alarm interrupt occurs at 1:0:05, followed by each interval alarm at 10 seconds past each minute for three consecutive alarms.

The Console Display Window Shows:

|                     |          |        |
|---------------------|----------|--------|
| Set TOD Clock:      | 1/1/2002 | 1:0:0  |
| Set First Alarm:    | 1/1/2002 | 1:0:5  |
| Set Interval Alarm: |          | 1:0:10 |
| Alarm occurred:     | 1/1/2002 | 1:0:5  |
| Alarm occurred:     | 1/1/2002 | 1:0:10 |
| Alarm occurred:     | 1/1/2002 | 1:1:10 |
| Done                |          |        |



## 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\dsp5685xevm\nos\bsp` of the SDK where **dsp5685xevm** specifies the target being used (i.e., **dsp56858evm** selects the DSP56858EVM). This chapter is composed of two major components: Interfaces and Drivers. This document focuses on specific DSP56858EVM 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 DSP56858EVM 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 either of two levels: a device-independent interface and a device-dependent Button driver interface. The device-independent interface invokes the device-dependent Button driver interface.

The device-independent API to the Button interface is standard and more portable than the device-dependent Button interface, but less efficient. The device-dependent Button driver provides a non-standard interface that is more efficient than the device-independent interface, but less portable.

Your application may use either the device-independent interface or the device-dependent 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 device-independent *open* function with any device-dependent Button Driver function call, nor may you use the File Descriptor returned by the device-dependent driver *buttonOpen* call with any device-independent 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 device-independent interface defines functions as follows:
 * types_tHandle open(const char *pName, int OFlags,
 * button_sCallback * pCallbackSpec);
 * int close(types_tHandle FileDesc);
 *
 * The device-dependent 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 <a href="#">Table 6-2</a>                                         |
| <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**

| DSP56858EVM              |
|--------------------------|
| 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 6-1](#).

### 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 **DSP56858 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 DSP56858EVM 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 DSP56858EVM configures the codec to be in Mode 4, which has separate data and control channels. The DSP56858EVM 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 ESSIO for network mode to handle this case (one frame sync for multiple 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 ESSIO to synchronize the receive and transmit channels which means only one interrupt service routine (ISR) is needed. The Codec driver uses the ESSIO 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 defined constants     | Explanation and Default Value                                                                                                                       |
|-----------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| CODEC_MODE                  | Sets Codec operation to either stereo or mono mode<br><br>Valid values are:<br>CODEC_STEREO (default)<br>CODEC_MONO                                 |
| CODEC_LEFT_D2A_ATTENUATION  | Left Channel Output Attenuation in 1.5dB Increments:<br><br>00000 = 0dB attenuation (default)<br>11111 = 46.5dB attenuation                         |
| CODEC_RIGHT_D2A_ATTENUATION | Right Channel Output Attenuation in 1.5dB Increments:<br><br>00000 = 0dB attenuation (default)<br>11111 = 46.5dB attenuation                        |
| CODEC_MUTE                  | Initial state of the Codec Mute<br><br>Valid values are:<br>CODEC_MUTE_DISABLED (default)<br>CODEC_MUTE_ENABLED                                     |
| CODEC_ISL                   | Selects the Codec input pin for the left channel<br><br>Valid values are:<br>CODEC_LEFT_INPUT_LINE_1 (default)<br>CODEC_LEFT_INPUT_LINE_2           |
| CODEC_ISR                   | Selects the Codec input pin for the right channel<br><br>Valid values are:<br>CODEC_RIGHT_INPUT_LINE_1 (default)<br>CODEC_RIGHT_INPUT_LINE_2        |
| CODEC_LEFT_A2D_GAIN         | Left Channel Input Gain in 1.5dB Increments:<br><br>0000 = 0dB gain<br>0001 = 1.5dB gain<br>0010 = 3.0dB gain (default)<br>1111 = 22.5dB gain       |
| CODEC_RIGHT_A2D_GAIN        | Right Channel Input Gain in 1.5dB Increments:<br><br>0000 = no gain (0dB)<br>0001 = 1.5dB gain<br>0010 = 3.0dB gain (default)<br>1111 = 22.5dB gain |

**Table 6-6. Codec Configuration Items for *appconfig.h* (Continued)**

| Codec defined constants | Explanation and Default Value                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CODEC_FIFO_SIZE         | <p>Default size (number of 16-bit samples) for both the RX FIFO and TX FIFO buffers used internally by the driver.</p> <p>For more information, refer to CODEC_RX_FIFO_SIZE and CODEC_TX_FIFO_SIZE</p> <p>Default = 32</p>                                                                                                                                                                                                                                                                                                                                                            |
| CODEC_RX_FIFO_SIZE      | <p>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.</p> <p>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.</p> <p>Default = CODEC_FIFO_SIZE</p>                                                                                    |
| CODEC_TX_FIFO_SIZE      | <p>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.</p> <p>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.</p> <p>Default = CODEC_FIFO_SIZE</p>                                                                                 |
| CODEC_RX_CALLBACK_LEVEL | <p>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.</p> <p>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.</p> <p>Default = 8</p> |

**Table 6-6. Codec Configuration Items for *appconfig.h* (Continued)**

| Codec defined constants | Explanation and Default Value                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|-------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CODEC_RX_CALLBACK       | <p>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>:</p> <pre>typedef void (*codec_tCallback) (void *pCallbackArg, int MaxNBytes);</pre> <p>The parameters to the callback function are:</p> <p><i>pCallbackArg</i> - The original value specified by the user for CODEC_RX_CALLBACK_ARG</p> <p><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></p> <p>Default = NULL</p> |
| CODEC_RX_CALLBACK_ARG   | <p>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.</p> <p>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.</p> <p>Default = NULL</p>                                                                                                                                                    |
| CODEC_TX_CALLBACK_LEVEL | <p>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.</p> <p>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.</p> <p>Default = 8</p>             |



**Table 6-6. Codec Configuration Items for *appconfig.h* (Continued)**

| Codec defined constants | Explanation and Default Value                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|-------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CODEC_TX_CALLBACK       | <p>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>:</p> <pre>typedef void (*codec_tCallback) (void *pCallbackArg, int MaxNBytes);</pre> <p>The parameters to the callback function are:</p> <p><i>pCallbackArg</i> - The original value specified by the user for CODEC_TX_CALLBACK_ARG</p> <p><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 <i>MaxNBytes/2</i></p> <p>Default = NULL</p> |
| CODEC_TX_CALLBACK_ARG   | <p>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.</p> <p>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.</p> <p>Default = NULL</p>                                                                                                          |
| CODEC_MASK              | <p>Bit 11 of CODEC_TX_CONTROL_WORD</p> <p>Valid values are:<br/>CODEC_INTERRUPT_MASKED (default)<br/>CODEC_INTERRUPT_UNMASKED</p> <p>Please see the Crystal CS4218 Stereo Codec hardware reference for more details</p>                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| CODEC_DO1               | <p>Bit 10 of CODEC_TX_CONTROL_WORD</p> <p>Valid values are<br/>CODEC_DIGITAL_OUTPUT_1_LOW (default)<br/>CODEC_DIGITAL_OUTPUT_1_HIGH</p> <p>Please see the Crystal CS4218 Stereo Codec hardware reference for more details.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                |

Code Example: [Code Example 6-3](#) configures and initializes the Codec driver.

**Code Example 6-3. Sample *appconfig.h* code**

---

```
#define INCLUDE_BSP /* include board support package (required) */
#define INCLUDE_IO /* include device-independent 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 either of two levels: a device-independent interface and a device-dependent Codec driver interface. The device-independent interface invokes the device-dependent Codec driver interface.

The device-independent API to the Codec interface is standard and more portable than the device-dependent codec interface, but less efficient. The device-dependent Codec driver provides a non-standard interface that is more efficient than the device-independent interface, but less portable.

Your application may use either the device-independent interface or the device-dependent 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 device-independent *open* function with any device-dependent Codec driver function call, nor may you use the File Descriptor returned by the device-dependent driver *codecOpen* call with any device-independent 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 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 device-dependent 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 Codec Device-Independent API Specification

The following pages specify the Application Programming Interface (API) for the device-independent interface to the Codec driver.

To use the device-independent 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 device-independent 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.

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.

### 6.2.4.1 *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:<br><br>BSP_DEVICE_NAME_CODEC_0     |
| <i>OFlag</i>    | <i>in</i>  | Open Mode Flags<br><br>Valid values are:<br>0 (default is Blocking)<br>O_NONBLOCKING                                     |
| <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/2</i> |

**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 1.5dB Increments:<br><br>0000 = 0dB gain<br>0001 = 1.5dB gain<br>0010 = 3.0dB gain (default)<br>...<br>1111 = 22.5dB gain<br><br>To calculate the RX gain value using a portable function, use the macro CODEC_RX_GAIN_FROM_PERCENT<br>See <b>Macros</b> | <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 | <p>Right Channel Input Gain in 1.5dB Increments:</p> <p>0000 = 0dB gain<br/> 0001 = 1.5dB gain<br/> 0010 = 3.0dB gain<br/> (default)<br/> ...<br/> 1111 = 22.5dB gain</p> <p>To calculate the RX gain value using a portable function, use the macro CODEC_RX_GAIN_FROM_PERCENT</p> <p>A description follows in <b>Macros</b></p>                | void   | Sets the right channel input gain               |
| CODEC_DEVICE_SET_TX_LEFT_GAIN  | <p>Left Channel Output Gain (integer)</p> <p>However, the Crystal CS4218 Stereo Codec supports only attenuation in 1.5dB Increments:</p> <p>00000 = 0dB<br/> ...<br/> 11111 = 46.5dB</p> <p>To calculate the TX gain value using a portable function, use the macro CODEC_TX_GAIN_FROM_PERCENT</p> <p>A description follows in <b>Macros</b></p> | void   | Sets the left channel output gain (attenuation) |

Table 6-12. Codec *ioctl* Commands (Continued)

| Cmd                                | pParams                                                                                                                                                                                                                                                                                                                                           | Return      | Result                                                                                                                 |
|------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------|------------------------------------------------------------------------------------------------------------------------|
| CODEC_DEVICE_SET_TX_RIGHT_GAIN     | <p>Right Channel Output Gain (integer)</p> <p>However, the Crystal CS4218 Stereo Codec supports only attenuation in 1.5dB Increments:</p> <p>00000 = 0dB<br/> ...<br/> 11111 = 46.5dB</p> <p>To calculate the TX gain value using a portable function, use the macro CODEC_TX_GAIN_FROM_PERCENT</p> <p>A description follows in <b>Macros</b></p> | <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_MODE                  | <i>int</i> (CODEC_MONO or CODEC_STEREO)                                                                                                                                                                                                                                                                                                           | <i>void</i> | Sets the mode of the codec to either stereo or mono                                                                    |
| 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 the return for each command.

**Code Example:** [Code Example 6-4](#) reads and writes eight codec samples (16 bytes) at a time, using the device-independent API to the Codec Driver in a Non-Blocking mode:

#### Code Example 6-4. Codec Driver Device-Independent API Example

```

/*****
#include "port.h"
#include "bsp.h"
#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:<br><br>BSP_DEVICE_NAME_CODEC_0                              |
| <i>OFlag</i>    | <i>in</i>  | Open Mode Flags<br><br>Valid values are:<br>0 (default is Blocking)<br>O_NONBLOCKING                                                              |
| <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. This file descriptor 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/2</i> |

**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:<br>BSP_DEVICE_NAME_CODEC_0 |

**Table 6-18. Codec *codecloctl* 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 1.5dB Increments:<br>0000 = 0dB gain<br>0001 = 1.5dB gain<br>0010 = 3.0dB gain (default)<br>...<br>1111 = 22.5dB gain<br><br>To calculate the RX gain value using a portable function, use the macro CODEC_RX_GAIN_FROM_PERCENT<br>See <b>Macros</b> | <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 | <p>Right Channel Input Gain in 1.5dB Increments:</p> <p>0000 = 0dB gain<br/> 0001 = 1.5dB gain<br/> 0010 = 3.0dB gain (default)<br/> ...<br/> 1111 = 22.5dB gain</p> <p>To calculate the RX gain value using a portable function, use the macro CODEC_RX_GAIN_FROM_PERCENT<br/> See <b>Macros</b></p>                     | <i>void</i> | Sets the right channel input gain               |
| CODEC_DEVICE_SET_TX_LEFT_GAIN  | <p>Left Channel Output Gain (integer)</p> <p>However, the Crystal CS4218 Stereo Codec supports only attenuation in 1.5dB Increments:</p> <p>00000 = 0dB<br/> ...<br/> 11111 = 46.5dB</p> <p>To calculate the TX gain value using a portable function, use the macro CODEC_TX_GAIN_FROM_PERCENT<br/> See <b>Macros</b></p> | <i>void</i> | Sets the left channel output gain (attenuation) |

**Table 6-18. Codec *codecloctl* Commands (Continued)**

| Cmd                                | pParams                                                                                                                                                                                                                                                                                                       | Return      | Result                                                                                                                 |
|------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------|------------------------------------------------------------------------------------------------------------------------|
| CODEC_DEVICE_SET_TX_RIGHT_GAIN     | Right Channel Output Gain (integer)<br><br>However, the Crystal CS4218 Stereo Codec supports only attenuation in 1.5dB Increments:<br><br>00000 = 0dB<br>...<br>11111 = 46.5dB<br><br>To calculate the TX gain value using a portable function, use the macro CODEC_TX_GAIN_FROM_PERCENT<br>See <b>Macros</b> | <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_MODE                  | <i>int</i> (CODEC_MONO or CODEC_STEREO)                                                                                                                                                                                                                                                                       | <i>void</i> | Sets the mode of the codec to either stereo or mono                                                                    |
| 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 *codecloctl* call provides various control functions for the codec. See the list of commands above for specific control functions.

**Returns:** The *codecIoctl* call returns a type *UWord16*. This *UWord16* is dependent on the command passed in as an input parameter. [Table 6-18](#) 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 "port.h"
#include "bsp.h"
#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 DSP56858 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 DSP56858EVM board:
  - The three switches on the dipswitch (S5) 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 **DSP56858 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 (S5); note the difference in sound quality between an 8KHz sampling rate and a 48KHz sampling rate

## 6.3 Codec DMA Driver

### 6.3.1 Introduction

This driver is designed for the Crystal CS4218 Stereo Codec hardware located on the DSP56858EVM 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 DSP56858EVM configures the codec to be in Mode 4, which has separate data and control channels. The DSP56858EVM 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 DMA Driver configures the ESSIO for network mode to handle this case (one frame sync for multiple samples).

The Codec DMA Driver can be configured for either stereo mode or mono mode. In stereo mode, all receive 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. All samples to be transmitted in stereo mode must also be interlaced prior to being sent to the Codec DMA driver. In mono mode, the Codec DMA 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 only the left channel.

The Codec DMA Driver also configures the ESSIO to synchronize the receive and transmit channels.

The Codec DMA Driver provides the following features:

- Uses DMA
- Non-Blocking mode only
- Function callbacks to notify the user when data is received, data is transmitted, or an exception has occurred
- Static configuration parameters to establish initial codec settings for gain; attenuation; mode (stereo/mono); and user callback functions
- Dynamic control functions to change channel gain and attenuation; enable/disable the codec; and mute all sound

The following sections describe how to statically configure the Codec DMA Driver via the SDK *appconfig.h* file, along with APIs for I/O and control functions.

### 6.3.2 Initialization

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

To include the device-dependent Codec DMA Driver in the application, insert the following line in the *appconfig.h* file:

```
#define INCLUDE_CODECDMA
```

Defining the preprocessor variable, INCLUDE\_CODECDMA, directs the SDK to include the Codec DMA Driver in the executable image. If the preprocessor variable INCLUDE\_CODECDMA is not defined, then the Codec DMA Driver will not be included in the executable.

To use the device-independent Application Programming Interface for the Codec DMA Driver, insert the following line in the *appconfig.h* file:

```
#define INCLUDE_IO
#define INCLUDE_CODECDMA
```

Defining the preprocessor variable, INCLUDE\_IO, directs the SDK to include the device-independent Application Programming Interface in the executable image. If the preprocessor variable INCLUDE\_IO is not defined, then the user application must use the low-level Application Programming Interface of the Codec DMA Driver.

You may also define additional preprocessor variables in the *appconfig.h* file to statically configure and initialize the Codec DMA Driver. A list of these variables is provided in [Table 6-19](#).

**Table 6-19. Codec DMA Configuration Items for *appconfig.h***

| Codec DMA Defined Constants | Explanation and Default Value                                                                                                                                   |
|-----------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CODECDMA_RX_DMA_CHANNEL     | <p>DMA Channel to use for the Codec Receiver</p> <p>Valid values are 0 to 5. Cannot be the same as CODECDMA_TX_DMA_CHANNEL</p> <p>Default = 0</p>               |
| CODECDMA_TX_DMA_CHANNEL     | <p>DMA Channel to use for the Codec Transmitter</p> <p>Valid values are:<br/>0 to 5</p> <p>Cannot be the same as CODECDMA_RX_DMA_CHANNEL</p> <p>Default = 1</p> |

Table 6-19. Codec DMA Configuration Items for *appconfig.h* (Continued)

| Codec DMA Defined Constants | Explanation and Default Value                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|-----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CODECDMA_MODE               | <p>Sets codec operation to either stereo or mono mode</p> <p>Valid values are:<br/>CODEC_STEREO<br/>CODEC_MONO</p> <p>Default = CODEC_STEREO</p>                                                                                                                                                                                                                                                                                                                                                                                                       |
| CODECDMA_RX_CALLBACK        | <p>A user function for the Codec DMA Driver to call when the DMA for the Codec Receiver has completed its transfer. The user function must be of type <i>types_tCallback</i>:</p> <pre>typedef void (*types_tCallback) (void *pCallbackArg);</pre> <p>The parameters to the callback function are:</p> <p><i>pCallbackArg</i> - The original value specified by the user for CODECDMA_RX_CALLBACK_ARG</p> <p>Default = NULL</p>                                                                                                                        |
| CODECDMA_RX_CALLBACK_ARG    | <p>A value that the user wants to pass to the CODECDMA_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.</p> <p>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 CODECDMA_RX_CALLBACK function.</p> <p>Default = NULL</p> |
| CODECDMA_TX_CALLBACK        | <p>A user function for the Codec Driver to call when the DMA for the Codec Transmitter has completed its transfer. The user function must be of type <i>types_tCallback</i>:</p> <pre>typedef void (*types_tCallback) (void *pCallbackArg);</pre> <p>The parameters to the callback function are:</p> <p><i>pCallbackArg</i> - The original value specified by the user for CODECDMA_TX_CALLBACK_ARG</p> <p>Default = NULL</p>                                                                                                                         |

**Table 6-19. Codec DMA Configuration Items for *appconfig.h* (Continued)**

| Codec DMA Defined Constants    | Explanation and Default Value                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|--------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CODECDMA_TX_CALLBACK_ARG       | <p>A value that the user wants to pass to the CODECDMA_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.</p> <p>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 CODECDMA_TX_CALLBACK function.</p> <p>Default = NULL</p> |
| CODECDMA_CALLBACK_EXCEPTION    | <p>A user function for the Codec DMA Driver to call when an exception has been detected. The user function must be of type <i>codecdma_tErrorCallback</i>:</p> <pre>typedef void (*codecdma_tErrorCallback) (UWord16 Error);</pre> <p>The parameters to the callback function are:</p> <p><i>Error</i> - Contains the exception</p> <p>Default = NULL</p>                                                                                                                                                                                              |
| CODECDMA_LEFT_D2A_ATTENUATION  | <p>Left Channel Output Attenuation in 1.5dB Increments; see <a href="#">Figure 6-1</a></p> <p>Valid values are 0 to 31<br/>                     0 = no attenuation (0dB)<br/>                     1 = 1.5dB attenuation<br/>                     2 = 3.0dB attenuation<br/>                     ...<br/>                     31 = 46.5dB attenuation</p> <p>Default = 0</p>                                                                                                                                                                            |
| CODECDMA_RIGHT_D2A_ATTENUATION | <p>Right Channel Output Attenuation in 1.5dB Increments; see <a href="#">Figure 6-1</a></p> <p>Valid values are 0 to 31<br/>                     0 = no attenuation (0dB)<br/>                     1 = 1.5dB attenuation<br/>                     2 = 3.0dB attenuation<br/>                     ...<br/>                     31 = 46.5dB attenuation</p> <p>Default = 0</p>                                                                                                                                                                           |

**Table 6-19. Codec DMA Configuration Items for *appconfig.h* (Continued)**

| Codec DMA Defined Constants | Explanation and Default Value                                                                                                                                                                                       |
|-----------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CODECDMA_MUTE               | Initial state of the Codec Mute<br><br>Valid values are:<br>CODEC_MUTE_DISABLED<br>CODEC_MUTE_ENABLED<br><br>Default = CODEC_MUTE_DISABLED                                                                          |
| CODECDMA_ISL                | Selects the Codec input pin for the left channel; ISL stands for Input Source Left.<br><br>Valid values are:<br>CODEC_LEFT_INPUT_LINE_1<br>CODEC_LEFT_INPUT_LINE_2<br><br>Default = CODEC_LEFT_INPUT_LINE_1         |
| CODECDMA_ISR                | Selects the Codec input pin for the right channel; ISR stands for Input Source Right<br><br>Valid values are:<br>CODEC_RIGHT_INPUT_LINE_1<br>CODEC_RIGHT_INPUT_LINE_2<br><br>Default = CODEC_RIGHT_INPUT_LINE_1     |
| CODECDMA_LEFT_A2D_GAIN      | Left Channel Input Gain in 1.5dB Increments; see <a href="#">Figure 6-1</a><br><br>Valid values are 0 to 15<br>0 = no gain (0dB)<br>1 = 1.5dB gain<br>2 = 3.0dB gain<br>...<br>15 = 22.5dB gain<br><br>Default = 2  |
| CODECDMA_RIGHT_A2D_GAIN     | Right Channel Input Gain in 1.5dB Increments; see <a href="#">Figure 6-1</a><br><br>Valid values are 0 to 15<br>0 = no gain (0dB)<br>1 = 1.5dB gain<br>2 = 3.0dB gain<br>...<br>15 = 22.5dB gain<br><br>Default = 2 |

Table 6-19. Codec DMA Configuration Items for *appconfig.h* (Continued)

| Codec DMA Defined Constants | Explanation and Default Value                                                                                                                                                                                                                                                 |
|-----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CODECDMA_MASK               | <p>Masks or unmask an interrupt on the actual codec</p> <p>Valid values are:<br/>CODEC_INTERRUPT_MASKED<br/>CODEC_INTERRUPT_UNMASKED</p> <p>Please see the Crystal CS4218 Stereo Codec hardware reference for more details</p> <p>Default = CODEC_INTERRUPT_MASKED</p>        |
| CODECDMA_DO1                | <p>Sets the digital output gain to high or low</p> <p>Valid values are:<br/>CODEC_DIGITAL_OUTPUT_1_LOW<br/>CODEC_DIGITAL_OUTPUT_1_HIGH</p> <p>Please see the Crystal CS4218 Stereo Codec hardware reference for more details.</p> <p>Default = CODEC_DIGITAL_OUTPUT_1_LOW</p> |

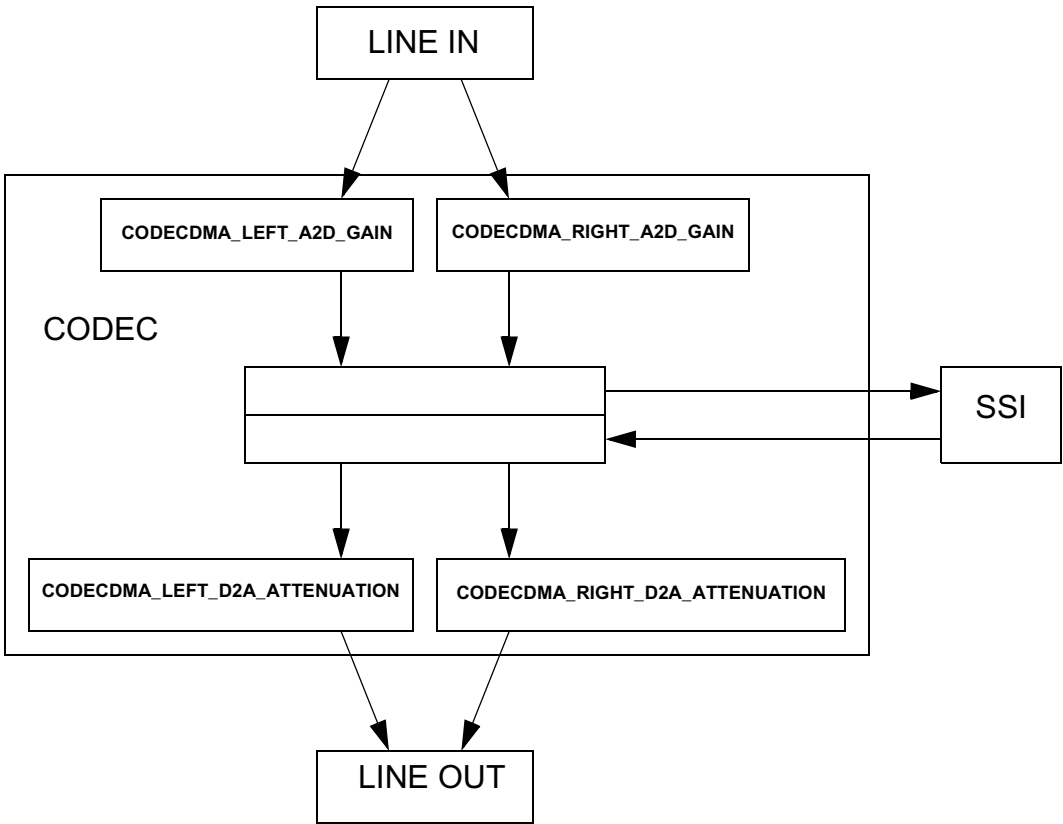


Figure 6-1. Codec Configuration



**Code Example:** [Code Example 6-6](#) configures and initializes the Codec DMA Driver.

**Code Example 6-6. Sample *appconfig.h* code**

---

```
#define INCLUDE_BSP /* include board support package (required) */
#define INCLUDE_IO /* include device-independent API (optional) */
#define INCLUDE_CODECDMA /* include Codec DMA driver (required) */

#define CODECDMA_MODE CODEC_MONO

extern void CodecRXISR (void *pCallbackArg);
extern void CodecTXISR (void *pCallbackArg);

#define CODECDMA_RX_CALLBACK CodecRXISR
#define CODECDMA_TX_CALLBACK CodecTXISR
```



### 6.3.3 API Definition

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

The codec DMA interface can be used at either of two levels: a device-independent interface or a device-dependent Codec DMA driver interface. The device-independent interface invokes the device-dependent Codec DMA driver interface.

The device-independent API to the Codec DMA interface is standard and more portable than the device-dependent Codec DMA interface, but less efficient. The device-dependent Codec DMA 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 Codec DMA driver interface to the Codec DMA driver, depending on your specific goals for efficiency and portability. However, you should not mix the Codec DMA 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 Codec DMA driver function call, nor may you use the File Descriptor returned by the device-dependent *codecdmaOpen* call with any device-independent interface function call.

The following information may be found in the public header file *codecdma.h*. You must include *codecdma.h* in your application to utilize the Codec DMA driver 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 device-dependent interface defines functions as follows:
* types_tHandle codecdmaOpen (const char *pName, int OFlags);
* ssize_t codecdmaRead (types_tHandle FileDesc, void * pBuffer, size_t NBytes);
* ssize_t codecdmaWrite (types_tHandle FileDesc, const void * pBuffer,
* size_t NBytes);
* UWord16 codecdmaIoctl (types_tHandle FileDesc, UWord16 Cmd,
* codec_sParams * pParams, const char *pName);
* int codecdmaClose (types_tHandle FileDesc);
*
*****/

```

#### Public Data Structure(s):

None



## 6.3.4 Codec DMA Device-Independent API Specification

The following pages specify the Application Programming Interface (API) for the device-independent interface to the Codec DMA Driver.

To use the device-independent interface to the Codec DMA Driver, you must define both the `INCLUDE_CODECDMA` 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_CODECDMA
```

The user's application code then uses the Codec DMA 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 from the User Buffer to the codec (transmit)
- *read* calls are made to read data from the codec to the User Buffer (receive)
- After all codec operations are completed, the Codec DMA driver is closed via a *close* call

The following sections describe each function of the device-independent API for the Codec DMA 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.

*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. Codec DMA 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:<br>BSP_DEVICE_NAME_CODECDMA_0                                          |
| <i>OFlag</i>    | <i>in</i>  | Open Mode Flags. Note that this driver only supports Non-Blocking mode;<br>- O_RDWR - Read and Write (default)<br>- O_NONBLOCK - Non-Blocking Mode (default) |
| <i>FileDesc</i> | <i>out</i> | Codec DMA device descriptor returned by <i>open</i> call and used in subsequent <i>read/write/close/ioctl</i> function calls                                 |

**Description:** This function opens the Codec DMA peripheral for operations for use with the common I/O layer interface.

This function calls *codecdmaOpen()*, so please see the description of *codecdmaOpen()* for a description of this function.

**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 DMA device and is used by the Codec DMA 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. Codec DMA Driver Arguments - write**

|                 |           |                                                                                                                                       |
|-----------------|-----------|---------------------------------------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | Codec DMA device descriptor returned by the <i>open</i> call                                                                          |
| <i>pBuffer</i>  | <i>in</i> | Array containing the samples to be transmitted                                                                                        |
| <i>NBytes</i>   | <i>in</i> | Size (in 8-bit bytes) of the data to be written out of the Codec DMA device; note that the number of 16-bit words is <i>NBytes</i> /2 |

**Description:** This function is mapped to *codecdmaWrite()* via the *codecdmadrvtIOInterfaceVT* structure. Please see the description of *codecdmaWrite()* for a description of this function.

**Returns:** The size returned is zero. The user's callback function will be called when the desired amount of data has been written.

**Code Example:** See [Code Example 6-7](#).

### 6.3.4.3 *read*

**Call(s):**

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

**Arguments:**

**Table 6-22. Codec DMA Driver Arguments - *read***

|                 |              |                                                                                                                                  |
|-----------------|--------------|----------------------------------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i>    | Codec DMA device descriptor returned by the <i>open</i> call                                                                     |
| <i>pBuffer</i>  | <i>inout</i> | Array to store the received samples in                                                                                           |
| <i>NBytes</i>   | <i>in</i>    | Size (in 8-bit bytes) of the data to be read from the Codec DMA device; note that the number of 16-bit words is <i>NBytes</i> /2 |

**Description:** This function is mapped to *codecdmaRead()* via the *codecdmadrvtIOInterfaceVT* structure. Please see the description of *codecdmaRead()* for a description of this function.

**Returns:** The size returned is zero. The user's callback function will be called when the desired amount of data has been read.

**Code Example:** See [Code Example 6-7](#).



### 6.3.4.4 *close*

**Call(s):**

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

**Arguments:**

**Table 6-23. Codec DMA Driver Arguments - *close***

|                 |           |                                                          |
|-----------------|-----------|----------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | Codec DMA device descriptor returned by <i>open</i> call |
|-----------------|-----------|----------------------------------------------------------|

**Description:** This function is mapped to *codecdmaClose()* via the *codecdmadrvtIOInterfaceVT* structure. Please see the description of *codecdmaClose()* 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 6-7](#).

### 6.3.4.5 *ioctl*

**Call(s):**

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

**Arguments:**

**Table 6-24. Codec DMA Driver Arguments - *ioctl***

|                 |           |                                                                                   |
|-----------------|-----------|-----------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | Codec DMA device descriptor returned by <i>open</i> call                          |
| <i>Cmd</i>      | <i>in</i> | Commands found in <i>codecdma.h</i> which are used to manage the Codec DMA driver |
| <i>pParams</i>  | <i>in</i> | <i>ioctl</i> command parameter                                                    |

**Table 6-25. Codec DMA *ioctl* Commands**

| <b>Cmd</b>                      | <b><i>pParams</i></b>                                  | <b>Return</b> | <b>Result</b>                                                                                                                                                                                                                                                                                                                               |
|---------------------------------|--------------------------------------------------------|---------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CODECDMA_DEVICE_ENABLE          | <i>NULL</i>                                            | None          | Enables the Codec device and starts data transfer. To configure the DMA device to read from or write to a valid buffer, <i>read()</i> or <i>write()</i> should be called prior to using the command. This command allows the user to synchronize the transfer of data by setting up all DMA buffers prior to enabling the Codec peripheral. |
| CODECDMA_DEVICE_DISABLE         | <i>NULL</i>                                            | None          | Disables the Codec device and stops data transfer                                                                                                                                                                                                                                                                                           |
| CODECDMA_SET_TX_CALLBACK        | void (*types_tCallback)<br>(void *)                    | None          | This command installs a Transmit Callback function to be called by the Codec DMA driver when a <i>Write</i> DMA transfer has completed                                                                                                                                                                                                      |
| CODECDMA_SET_TX_CALLBACK_ARG    | void * <i>pCallbackArg</i>                             | None          | This command defines the transmit callback argument to be used when the transmit callback function is called                                                                                                                                                                                                                                |
| CODECDMA_SET_RX_CALLBACK        | void (*types_tCallback)<br>(void *)                    | None          | This command installs a Receive Callback function to be called by the Codec DMA driver when a <i>Read</i> DMA transfer has completed                                                                                                                                                                                                        |
| CODECDMA_SET_RX_CALLBACK_ARG    | void * <i>pCallbackArg</i>                             | None          | This command defines the receive callback argument to be used when the transmit callback function is called                                                                                                                                                                                                                                 |
| CODECDMA_SET_CALLBACK_EXCEPTION | void<br>(*types_tErrorCallback)<br>(UWord16 Exception) | None          | This command installs the Exception Callback function                                                                                                                                                                                                                                                                                       |

**Table 6-25. Codec DMA *ioctl* Commands (Continued)**

| Cmd                               | pParams                                                                                                                                                                                                                                                                                                                                            | Return | Result                                                       |
|-----------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------|--------------------------------------------------------------|
| CODECDMA_DEVICE_SET_RX_LEFT_GAIN  | <p>Left Channel Input Gain in 1.5dB Increments:</p> <p>0000 = 0dB gain<br/> 0001 = 1.5dB gain<br/> 0010 = 3.0dB gain (default)<br/> ...<br/> 1111 = 22.5dB gain</p> <p>To calculate the RX gain value using a portable function, use the macro CODEC_RX_GAIN_FROM_PERCENT</p> <p><b>A description follows in "Macros"</b></p>                      | None   | This command sets the left channel input gain                |
| CODECDMA_DEVICE_SET_RX_RIGHT_GAIN | <p>Right Channel Input Gain in 1.5dB Increments:</p> <p>0000 = 0dB gain<br/> 0001 = 1.5dB gain<br/> 0010 = 3.0dB gain (default)<br/> ...<br/> 1111 = 22.5dB gain</p> <p>To calculate the RX gain value using a portable function, use the macro CODEC_RX_GAIN_FROM_PERCENT</p> <p><b>A description follows in "Macros"</b></p>                     | None   | This command sets the right channel input gain               |
| CODECDMA_DEVICE_SET_TX_LEFT_GAIN  | <p>Left Channel Output Gain (integer)</p> <p>However, the Crystal CS4218 Stereo Codec supports only attenuation in 1.5dB Increments:</p> <p>00000 = 0dB<br/> ...<br/> 11111 = 46.5dB</p> <p>To calculate the TX gain value using a portable function, use the macro CODEC_TX_GAIN_FROM_PERCENT</p> <p><b>A description follows in "Macros"</b></p> | None   | This command sets the left channel output gain (attenuation) |

**Table 6-25. Codec DMA *ioctl* Commands (Continued)**

| Cmd                               | pParams                                                                                                                                                                                                                                                                                                                                  | Return | Result                                                        |
|-----------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------|---------------------------------------------------------------|
| CODECDMA_DEVICE_SET_TX_RIGHT_GAIN | Right Channel Output Gain (integer)<br><br>However, the Crystal CS4218 Stereo Codec supports only attenuation in 1.5dB Increments:<br><br>00000 = 0dB<br>...<br>11111 = 46.5dB<br><br>To calculate the TX gain value using a portable function, use the macro CODEC_TX_GAIN_FROM_PERCENT<br><br><b>A description follows in "Macros"</b> | None   | This command sets the right channel output gain (attenuation) |
| CODECDMA_DEVICE_MUTE              | Mute (true or false)                                                                                                                                                                                                                                                                                                                     | None   | This command mutes or un-mutes the codec                      |

**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:** This function is used to dynamically control the Codec DMA device. See the list of commands in [Table 6-25](#) 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 6-25](#) lists these returns based on the specific command.

**Code Example:** [Code Example 6-7](#) shows how to write Codec DMA Driver IO Layer API.

---

**Code Example 6-7. Codec DMA Driver Device-Independent API Example**


---

- file *appconfig.h*

```
#define INCLUDE_BSP
#define INCLUDE_CODECDMA
#define INCLUDE_IO
```



- user application:

```
#include "port.h"
#include "bsp.h"
#include "codecdma.h"

void rx_callback(void * pArg);
void tx_callback(void * pArg);

void rx_callback(void * pArg)
{
}

void tx_callback(void * pArg)
{
}

void main(void)
{
 types_tHandle CodecDma;
 int RxBuffer[10];
 int TxBuffer[10];

 // Open Codec DMA device

 CodecDma = open(BSP_DEVICE_NAME_CODECDMA_0, O_RDWR | O_NONBLOCK);

 // Dynamically change the Gain

 ioctl(CodecDma, CODEC_DEVICE_SET_RX_LEFT_GAIN,
 CODEC_RX_GAIN_FROM_PERCENT(50));
 ioctl(CodecDma, CODEC_DEVICE_SET_RX_RIGHT_GAIN,
 CODEC_RX_GAIN_FROM_PERCENT(50));

 ioctl(CodecDma, CODEC_DEVICE_SET_TX_LEFT_GAIN,
 CODEC_TX_GAIN_FROM_PERCENT(100));
 ioctl(CodecDma, CODEC_DEVICE_SET_TX_RIGHT_GAIN,
 CODEC_TX_GAIN_FROM_PERCENT(100));

 // Dynamically define callback functions

 ioctl(CodecDma, CODECDMA_SET_RX_CALLBACK, rx_callback);
 ioctl(CodecDma, CODECDMA_SET_TX_CALLBACK, tx_callback);

 // Read data from Codec

 read(CodecDma, (void *)&RxBuffer[0], sizeof(RxBuffer));

 // Write data to Codec

 write(CodecDma, (void *)&TxBuffer[0], sizeof(TxBuffer));

 // Enable the Codec DMA Device
```



```
ioctl(CodecDma, CODECDMA_DEVICE_ENABLE, NULL);

while(1)
{
 // Wait for callbacks to be called
 // Read more data
 // Transmit more data
 //...
}

// close Codec DMA device

close(CodecDma);
}
```

## 6.3.5 Codec DMA Device-Dependent API Specification

The following pages specify the Application Programming Interface (API) for the device-dependent interface to the Codec DMA Driver.

To use the device-dependent interface to the Codec DMA Driver, you must define the `INCLUDE_CODECDMA` variable in the *appconfig.h* file of the SDK Embedded Project you created in CodeWarrior:

```
#define INCLUDE_CODECDMA
```

The user's application code then uses the Codec DMA Driver API as follows:

- A *codecdmaOpen* call is made to initialize the codec device
- *codecdmaIoctl* calls are made to control the codec device
- *codecdmaWrite* calls are made to write data from the User Buffer to the codec (transmit)
- *codecdmaRead* calls are made to read data from the codec to the User Buffer (receive)
- After all codec operations are completed, the Codec DMA Driver is closed via a *codecdmaClose* call

The following sections describe each function of the device-dependent API for the Codec DMA 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.

*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.

### 6.3.5.1 *codecdmaOpen*

**Call(s):**

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

**Arguments:**

**Table 6-26. Codec DMA Driver Arguments - *codecdmaOpen***

|                 |            |                                                                                                                                                                                                 |
|-----------------|------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>pName</i>    | <i>in</i>  | The codec device name; see the <i>bsp.h</i> file for target specific values, such as:<br>BSP_DEVICE_NAME_CODECDMA_0                                                                             |
| <i>OFlag</i>    | <i>in</i>  | Open Mode Flags:<br>- O_RDWR - Read and Write (default)<br>- O_NONBLOCK - Non-Blocking Mode (default)<br><br>Note that this driver supports only Non-Blocking mode                              |
| <i>FileDesc</i> | <i>out</i> | Codec DMA device descriptor returned by <i>codecdmaOpen</i> call and used in subsequent <i>codecdmaRead</i> / <i>codecdmaWrite</i> / <i>codecdmaClose</i> / <i>codecdmaIoctl</i> function calls |

**Description:** This function opens the Codec DMA peripheral for operations for use with the device-dependent interface.

This function opens the desired Codec DMA device and initializes it based on the parameters passed in. This function opens an ESSI DMA Receiver device and a ESSI DMA Transmitter device and uses these to transfer data to/from the codec. This function also sets up the GPIO to communicate with the codec and configures the codec with the defaults, as defined in *config.c* and *const.c*.

This function does not enable the codec. The user must use *codecdmaIoctl(CODECDMA\_DEVICE\_ENABLE)* to enable the codec. By doing this, the Codec Receiver and Transmitter can be synchronized. Prior to enabling the codec, the user should call *codecdmaRead()* and/or *codecdmaWrite()* to set up a buffer for DMA transfer. If this is not done, data from the codec will be missed.

**Return:** The *codecdmaOpen* call returns a type *types\_tHandle*. When the *codecdmaOpen* call is successful, this *types\_tHandle* is a file descriptor (*FileDesc*) for the codec device and is used by the Codec DMA Driver's *codecdmaRead*, *codecdmaWrite*, *codecdmaClose*, and *codecdmaIoctl* functions. A "-1" is returned when the *codecdmaOpen* call is unsuccessful.

**Code Example:** See [Code Example 6-8](#).

### 6.3.5.2 *codecdmaWrite*

#### Call(s):

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

#### Arguments:

**Table 6-27. Codec DMA Driver Arguments - *codecdmaWrite***

|                 |           |                                                                                                                                                                      |
|-----------------|-----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | Codec DMA device descriptor returned by the <i>codecdmaOpen</i> call                                                                                                 |
| <i>pBuffer</i>  | <i>in</i> | Array containing the samples to write. If configured for Stereo mode, the data in the buffer must be interleaved (left sample first, right sample second, and so on) |
| <i>NBytes</i>   | <i>in</i> | <i>NBytes</i> (in 8-bit bytes) of the data to send; note that the number of 16-bit samples to send is <i>NBytes</i> /2; <i>NBytes</i> is assumed to be even          |

**Description:** This function calls the *essidmaWrite()* function using the ESSI DMA TX0 handle as the file descriptor.

No data is actually written with this function call. The user's application must define a callback function that the DMA Driver will call via the ESSI DMA Driver when the desired amount of data has been written to the codec

This function does not enable the codec. The user must use *codecdmaIoctl(CODECDMA\_DEVICE\_ENABLE)* to enable the codec. This allows the user to synchronize all of the DMA buffers for the codec peripheral by enabling the codec after configuring the DMA for the Codec Transmitter and Receiver.

**Returns:** The size returned is zero. The user's callback function will be called when the desired amount of data has been written.

**Code Example:** See [Code Example 6-8](#).

### 6.3.5.3 *codecdmaRead*

**Call(s):**

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

**Arguments:**

**Table 6-28. Codec DMA Driver Arguments - *codecdmaRead***

|                 |              |                                                                                                                                                                        |
|-----------------|--------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i>    | Codec DMA device descriptor returned by the <i>codecdmaOpen</i> call                                                                                                   |
| <i>pBuffer</i>  | <i>inout</i> | Array to store the received samples in. If configured for Stereo mode, the data in this buffer will be interleaved (left sample first, right sample second, and so on) |
| <i>NBytes</i>   | <i>in</i>    | <i>NBytes</i> (in 8-bit bytes) of the data to read; note that the number of 16-bit samples to read is <i>NBytes</i> /2; <i>NBytes</i> is assumed to be even            |

**Description:** This function calls the *essidmaRead()* function using the ESSI DMA Receiver handle as the file descriptor.

No data is actually read with this function call. The user's application must define a callback function that the DMA Driver will call via the ESSI DMA Driver when the desired mount of data has been read from the codec.

This function does not enable the codec. The user must use *codecdmaIoctl(CODECDMA\_DEVICE\_ENABLE)* to enable the codec. This allows the user to synchronize all of the DMA buffers for the codec peripheral by enabling the codec after configuring the DMA for the Codec Transmitter and Receiver.

**Returns:** The size returned is zero. The user's callback function will be called when the desired amount of data has been read.

**Code Example:** See [Code Example 6-8](#).



### 6.3.5.4 *codecdmaClose*

Call(s):

```
int codecdmaClose (types_tHandle FileDesc);
```

Arguments:

Table 6-29. Codec DMA Driver Arguments - *codecdmaClose*

|                 |           |                                                                  |
|-----------------|-----------|------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | Codec DMA device descriptor returned by <i>codecdmaOpen</i> call |
|-----------------|-----------|------------------------------------------------------------------|

**Description:** This function disables the codec and closes the ESSI DMA Receiver and Transmitter devices that were opened in *codecdmaOpen()*.

**Returns:** The *codecdmaClose* call returns a type *int*; it returns “0” if successful and “-1” if unsuccessful.

**Code Example:** See [Code Example 6-8](#).

### 6.3.5.5 *codecdmaIoctl*

#### Call(s):

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

#### Arguments:

**Table 6-30. Codec DMA Driver Arguments - *codecdmaIoctl***

|                 |           |                                                                                                                         |
|-----------------|-----------|-------------------------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | Codec DMA device descriptor returned by the <i>codecdmaOpen</i> call                                                    |
| <i>Cmd</i>      | <i>in</i> | Commands found in <i>codecdma.h</i> which are used to control the Codec DMA Driver                                      |
| <i>pParams</i>  | <i>in</i> | <i>codecdmaIoctl</i> command parameter                                                                                  |
| <i>pName</i>    | <i>in</i> | The codec DMA device name; see the <i>bsp.h</i> file for target specific values, such as:<br>BSP_DEVICE_NAME_CODECDMA_0 |

For a list of commands, see [Table 6-25](#). This table contains a list of commands for the Codec DMA *ioctl* function, which is identical to the *codecdmaIoctl* function

**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 Receiver 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 Transmitter 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:** This function is used to dynamically control the Codec DMA device.

**Returns:** The *codecdmaIoctl* call returns a type *UWord16*. This *UWord16* is dependent on the command passed in as an input parameter. [Table 6-25](#) lists these returns based on the specific command.

**Code Example:** [Code Example 6-8](#) shows how to create code for a Codec DMA Driver device-dependent API.



## Code Example 6-8. Codec DMA Driver Device-Dependent API Example

- file *appconfig.h*

```
#define BSP
#define INCLUDE_CODECDMA
```

- user application:

```
#include "port.h"
#include "bsp.h"
#include "codecdma.h"

void rx_callback(void * pArg);
void tx_callback(void * pArg);

void rx_callback(void * pArg)
{
}

void tx_callback(void * pArg)
{
}

void main(void)
{
 types_tHandle CodecDma;
 int RxBuffer[10];
 int TxBuffer[10];

 // Open Codec DMA device

 CodecDma = codecdmaOpen(BSP_DEVICE_NAME_CODECDMA_0,
 O_RDWR | O_NONBLOCK);

 // Dynamically change the Gain

 codecdmaIoctl(CodecDma, CODEC_DEVICE_SET_RX_LEFT_GAIN,
 CODEC_RX_GAIN_FROM_PERCENT(50),
 BSP_DEVICE_NAME_CODECDMA_0);
 codecdmaIoctl(CodecDma, CODEC_DEVICE_SET_RX_RIGHT_GAIN,
 CODEC_RX_GAIN_FROM_PERCENT(50),
 BSP_DEVICE_NAME_CODECDMA_0));

 codecdmaIoctl(CodecDma, CODEC_DEVICE_SET_TX_LEFT_GAIN,
 CODEC_TX_GAIN_FROM_PERCENT(100),
 BSP_DEVICE_NAME_CODECDMA_0));
 codecdmaIoctl(CodecDma, CODEC_DEVICE_SET_TX_RIGHT_GAIN,
 CODEC_TX_GAIN_FROM_PERCENT(100),
 BSP_DEVICE_NAME_CODECDMA_0));

 // Dynamically define callback functions
```



ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

```
codecdmaIoctl(CodecDma, CODECDMA_SET_RX_CALLBACK, rx_callback,
 BSP_DEVICE_NAME_CODECDMA_0));
codecdmaIoctl(CodecDma, CODECDMA_SET_TX_CALLBACK, tx_callback,
 BSP_DEVICE_NAME_CODECDMA_0));

// Read data from Codec

codecdmaRead(CodecDma, (void *)&RxBuffer[0], sizeof(RxBuffer));

// Write data to Codec

codecdmaWrite(CodecDma, (void *)&TxBuffer[0], sizeof(TxBuffer));

// Enable the Codec DMA Device

codecdmaIoctl(CodecDma, CODECDMA_DEVICE_ENABLE, NULL,
 BSP_DEVICE_NAME_CODECDMA_0));

while(1)
{
 // Wait for callbacks to be called
 // Read more data
 // Transmit more data
 ...
}

// close Codec DMA device

codecdmaClose(CodecDma);
}
```

## 6.3.6 Codec DMA Driver Tests

Currently, there are no specific tests provided for the Codec DMA Driver. However, many applications and demos use the Codec DMA Driver. In particular, the Codec DMA Driver applications in the next section exercise most of Codec DMA Driver's capabilities.

## 6.3.7 Codec DMA Driver Applications

### 6.3.7.1 Codec DMA Loopback Application

This application performs a simple audio loopback using codec hardware and a DSP56858 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 DMA driver interface functions.

#### 6.3.7.1.1 Set-up for Codec DMA Loopback Application

##### EVM Jumper Settings:

- Make the following changes to the default jumper settings for the DSP56858EVM board:
  - The three switches on the dipswitch (S5) 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 is shown in the **DSP56858 Evaluation Module User's Manual**
- Connect the audio source to Line In on the EVM
- Connect the Audio Output to Line Out or headphone jack 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 Line Out

#### 6.3.7.1.2 Procedure for Codec DMA Loopback Application

- Using CodeWarrior, open:
  - ...\\nos\\applications\\bsp\\codecdma\\codecdma.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 6.3.7.1.1](#)
- If the application is successful, you will hear audio output from the speaker, which should be the same as the input audio
  - You will 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 (S5); note the difference in sound quality between an 8KHz sampling rate and a 48KHz sampling rate
- If you press Button A, the volume will increase. If the volume is at its maximum level, then it switches to the lowest volume.
- If you press Button B, the sound will be muted (if sound is not muted) or unmuted (if sound is muted)

## 6.4 FILE I/O Driver

### 6.4.1 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 6-2** contains a basic diagram showing the set-up required to utilize the File I/O driver in an application.

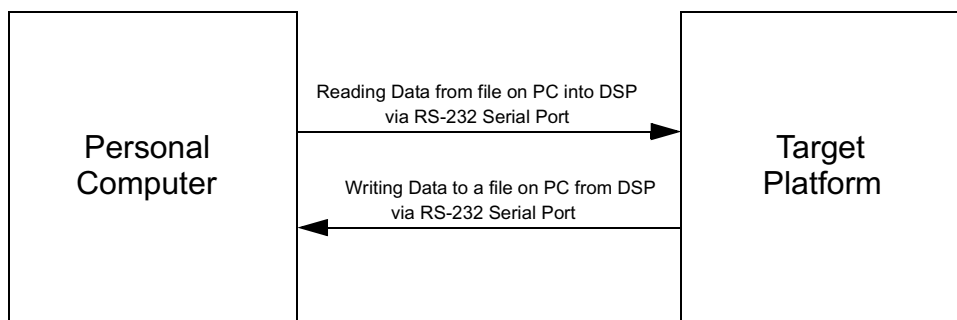


**Figure 6-2. Basic Set-up for File I/O Utilization**

**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, minimize the number of resources the PC is using when running a demo or user application, including the docking station. You 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 and then try running it again.**

## 6.4.2 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. A list of these variables is provided in [Table 6-31](#).

**Table 6-31. File I/O Configuration Items for *appconfig.h***

| File IO Defined Constants | Explanation and Default Value                              |
|---------------------------|------------------------------------------------------------|
| FILEIO_BAUD_RATE          | Sets the Baud Rate for the SCI port<br><br>Default = 56000 |

**Code Example:** [Code Example 6-9](#) configures and initializes the File I/O Driver.

**Code Example 6-9. Sample *appconfig.h* code**

```
#define INCLUDE_FILEIO /* include File I/O driver (Required) */
#define FILEIO_BAUD_RATE 9600 /* Changes SCI rate to 9600 bps */
```

## 6.4.3 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 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

## 6.4.4 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.



### 6.4.4.1 *open*

**Call(s):**

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

**Arguments:**
**Table 6-32. File I/O Driver Arguments - *open***

|                 |            |                                                                                                                   |
|-----------------|------------|-------------------------------------------------------------------------------------------------------------------|
| <i>pName</i>    | <i>in</i>  | The File I/O driver name; use:<br>\\\\PC\\specificdrive\\yourfile.txt<br>or<br>\\\\PC\\Embedded SDK\\yourfile.txt |
| <i>OFlag</i>    | <i>in</i>  | Use O_RDONLY for a read operation<br>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>writel/read</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 6-10](#).

### 6.4.4.2 *write*

**Call(s):**

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

**Arguments:**

**Table 6-33. File I/O Driver Arguments - *write***

|                 |           |                                                                                                      |
|-----------------|-----------|------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | File I/O device descriptor returned by <i>open</i> call and used in <i>write/read</i> function calls |
| <i>pBuffer</i>  | <i>in</i> | Pointer to user buffer. The write function type casts <i>pBuffer</i> to be a (char *)                |
| <i>Size</i>     | <i>in</i> | 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 6-10](#).



### 6.4.4.3 *read*

**Call(s):**

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

**Arguments:**

**Table 6-34. File I/O Driver Arguments - *read***

|                 |           |                                                                                                      |
|-----------------|-----------|------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | File I/O device descriptor returned by <i>open</i> call and used in <i>write/read</i> function calls |
| <i>pBuffer</i>  | <i>in</i> | Pointer to user buffer; the read function type casts <i>pBuffer</i> to be a ( <i>char *</i> )        |
| <i>Size</i>     | <i>in</i> | 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

#### 6.4.4.4 *close*

**Call(s):**

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

**Arguments:**

**Table 6-35. File I/O Driver Arguments - *close***

|                 |           |                                                                                                      |
|-----------------|-----------|------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | File I/O device descriptor returned by <i>open</i> call and used in <i>write/read</i> 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 6-10](#).

**6.4.4.5 ioctl**

Currently, there are no File I/O *ioctl* Commands.

**Code Example 6-10. 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);
}

*****/

```

**6.4.5 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.

## 6.5 Keypad Driver

### 6.5.1 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.

### 6.5.2 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.

### 6.5.3 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);
```



## 6.5.4 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.

### 6.5.4.1 *open*

#### Call(s):

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

#### Arguments:

**Table 6-36. KEYPAD Driver Arguments - *open***

|                 |            |                                                                                                                                |
|-----------------|------------|--------------------------------------------------------------------------------------------------------------------------------|
| <i>pName</i>    | <i>in</i>  | The TDC1 keypad device name:<br>BSP_DEVICE_NAME_KEYPAD                                                                         |
| <i>OFlag</i>    | <i>in</i>  | Open Mode Flags :<br>O_RDONLY- read allowed (default)<br>O_BLOCK - Blocking mode (default)<br>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 6.5.5.1](#).

**Returns:** See *keypadkeypadOpen*, [Section 6.5.5.1](#).

**Code Example:** [Code Example 6-11](#) opens the keypad for communication using default configuration settings.

#### Code Example 6-11. 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);
}
```

### 6.5.4.2 *ioctl*

#### Call(s):

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

#### Arguments:

**Table 6-37. 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 <a href="#">Table 6-41</a>                  |
| <i>pParams</i>  | <i>in</i> | Parameters specific to each command; see <a href="#">Table 6-41</a> |

**Description:** The Keypad driver supports the same commands described for *keypadIoctl*; see *keypadIoctl*, [Section 6.5.5.2](#).

**Return:** See *keypadIoctl*, [Section 6.5.5.2](#).

**Code Example:** [Code Example 6-12](#) opens the driver in Blocking mode (default) and continually scans for the KEYPAD\_S26 key to be pressed.

**Code Example 6-12. 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);
 }
}
```

### 6.5.4.3 *close*

#### Call(s):

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

#### Arguments:

**Table 6-38. 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 6.5.5.3](#).

**Returns:** This function returns “0” if successful and “-1” if unsuccessful.

**Code Example:** [Code Example 6-13](#) 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 6-13. 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);
}
```



## 6.5.5 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.

### 6.5.5.1 keypadOpen

#### Call(s):

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

#### Arguments:

**Table 6-39. Keypad Driver Arguments - keypadOpen**

|                 |            |                                                                                                                                                  |
|-----------------|------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>pName</i>    | <i>in</i>  | The TDC1 keypad device name:<br>BSP_DEVICE_NAME_KEYPAD                                                                                           |
| <i>OFlag</i>    | <i>in</i>  | Open Mode Flags:<br>O_RDONLY - read allowed (default)<br>O_BLOCK - Blocking mode (default)<br>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 6-14](#) opens the TDC1 keypad for Non-Blocking communication.

#### Code Example 6-14. 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);
}
```

### 6.5.5.2 keypadIoctl

Call(s):

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

Arguments:

**Table 6-40. 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 <a href="#">Table 6-41</a>                  |
| <i>pParams</i>  | <i>in</i> | Parameters specific to each command; see <a href="#">Table 6-41</a> |
| <i>pName</i>    | <i>in</i> | BSP device name from <i>bsp.h</i> :<br>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 6-41](#).

**Returns:** Command specific; see [Table 6-41](#).

**Table 6-41. keypadIoctl Commands**

| <i>Cmd</i>                        | <i>pParams</i>                                       | <i>Return</i> | <i>Result</i>                                                                                                                                                                                             |
|-----------------------------------|------------------------------------------------------|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| KEYPAD_SET_KEY_PRESS_CALLBACK     | void<br>(*keypad_tCallback)<br>(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 6-41. keypadIoctl Commands (Continued)**

| <i>Cmd</i>                          | <i>pParams</i>                                       | <i>Return</i>      | <i>Result</i>                                                                                                                                                                                                                                                                                                                                                                                         |
|-------------------------------------|------------------------------------------------------|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| KEYPAD_SET_KEY_RELEASE_CALLBACK     | void<br>(*keypad_tCallback)<br>(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 6-15](#) 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 6-15. 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);
}
```

### 6.5.5.3 keypadClose

Call(s):

```
int keypadClose (types_tHandle FileDesc);
```

Arguments:

**Table 6-42. 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 6-16](#) 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 6-16. 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);
```



```
while(keyStatus.current != KEYPAD_S26)
{
 if(keyStatus.prev != keyStatus.current)
 {
 keyStatus.prev = keyStatus.current;
 }
}
keypadClose(keypadFD);
}
```

## 6.5.6 Demo Applications

The TDC1 Keypad Driver is used in the Feature Phone Demo Application, located in the following directory:

.../sdk/src/dsp56858evm/nos/applications/telephony/featurephone

## 6.5.7 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/dsp56858evm/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 DSP56858 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
```

### 6.5.7.1 Hardware Set-up

This test requires a TDC1 attached to the DSP56858EVM 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 **DSP56858 Evaluation Module User's Manual**.

### 6.5.7.2 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

## 6.6 LCD Driver

### 6.6.1 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.

## 6.6.2 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.



### 6.6.3 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);
*****/

```





## 6.6.4 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.

## 6.6.4.1 open

### Call(s):

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

### Arguments:

**Table 6-43. LCD Driver Arguments - open**

|                 |            |                                                                                                                                      |
|-----------------|------------|--------------------------------------------------------------------------------------------------------------------------------------|
| <i>pName</i>    | <i>in</i>  | The LCD device name:<br>BSP_DEVICE_NAME_LCD                                                                                          |
| <i>OFlag</i>    | <i>in</i>  | Open Mode Flags:<br>O_RDWR - read and write allowed (default)<br>O_BLOCK - Blocking mode (default)<br>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 6-11](#) opens the LCD for communication using default configuration settings.

### Code Example 6-17. 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);
}
```

## 6.6.4.2 write

Call(s):

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

Arguments:

**Table 6-44. 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 6.6.5.2](#).

**Returns:** See *lcdWrite*, [Section 6.6.5.2](#).

**Code Example:** [Code Example 6-18](#) 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 6-18. 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);
}
```

## 6.6.4.3 *ioctl*

### Call(s):

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

### Arguments:

**Table 6-45. 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 6.5.5.2](#).

**Return:** See *lcdIoctl*, [Section 6.5.5.2](#).

**Code Example:** [Code Example 6-12](#) opens the LCD driver and sets the cursor position to the last location on line 1.

### Code Example 6-19. 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);
}
```

### 6.6.4.4 *close*

#### Call(s):

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

#### Arguments:

**Table 6-46. 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 6.5.5.3](#).

**Returns:** This function returns “0” if successful, “-1” if unsuccessful.

**Code Example:** [Code Example 6-13](#) 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 6-20. 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);
}
```

## 6.6.5 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.

### 6.6.5.1 *lcdOpen*

#### Call(s):

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

#### Arguments:

**Table 6-47. LCD Driver Arguments - *lcdOpen***

|                 |            |                                                                                                                                                                 |
|-----------------|------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>pName</i>    | <i>in</i>  | The LCD device name:<br>BSP_DEVICE_NAME_LCD                                                                                                                     |
| <i>OFlag</i>    | <i>in</i>  | Open Mode Flags:<br>O_RDWR - read and write allowed (default)<br>O_BLOCK - Blocking mode (default)<br>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 6-14](#) opens the LCD for communication using default configuration settings.

#### Code Example 6-21. 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);
}
```

### 6.6.5.2 *lcdWrite*

**Call(s):**

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

**Arguments:**

**Table 6-48. 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 6-22](#) 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 6-22. 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);
```

```
}
```

### 6.6.5.3 *lcdloctl*

Call(s):

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

Arguments:

**Table 6-49. 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 <a href="#">Table 6-41</a>                  |
| <i>pParams</i>  | <i>in</i> | Parameters specific to each command; see <a href="#">Table 6-41</a> |
| <i>pName</i>    | <i>in</i> | BSP device name from <i>bsp.h</i> :<br>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 6-41](#).

**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 6-50. *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 <a href="#">Section 6.5.7.2</a> . |

Table 6-50. *lcdIoctl* Commands (Continued)

| <i>Cmd</i>              | <i>pParams</i>            | <b>Result</b>                                                                                                                                                                                                                                                                                                                                      |
|-------------------------|---------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 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 <a href="#">Section 6.5.7.2</a> . |
| LCD_SET_CURSOR_POSITION | LCD_LINE <i>n</i> +offset | <p>Sets cursor position as specified by <i>pParams</i>.</p> <p>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.</p> <p>Example: The last character position on line 1 is represented by (<i>LCD_LINE1</i>+19). Maximum offset is 19.</p>  |

**Code Example:** [Code Example 6-23](#) opens the driver and sets the cursor position to the last location on line 1.

#### Code Example 6-23. Use of *lcdIoctl*

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);
}
```

### 6.6.5.4 *lcdClose*

#### Call(s):

```
int lcdClose (types_tHandle FileDesc);
```

#### Arguments:

**Table 6-51. 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 6-24** 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 6-24. 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);
}
```

## 6.6.6 Demo Applications

The LCD Driver is used in the Feature Phone Demo Application located in the following directory:

`.../sdk/src/dsp56858evm/nos/applications/telephony/featurephone`

## 6.6.7 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/dsp56858evm/nos/bsp/test/tdc1-LCD/lcdtest.mcp`

### 6.6.7.1 Hardware Set-up

This test requires a TDC1 attached to the DSP56858EVM 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 DSP56858 Evaluation Module User's Manual.

### 6.6.7.2 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

## 6.7 LED Driver

### 6.7.1 Introduction

The LED interface manipulates the light emitting diodes (LEDs).

There are six LEDs on the DSP56858EVM, all controlled by GPIO. Since there are no dedicated GPIO for the DSP56858, 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. For the DSP5685x, the LEDs are multiplexed with ESS11, so the user will not be able to use both the LEDs and the ESS11 at the same time. [Table 6-52](#) shows which LEDs and peripherals are multiplexed.

**Table 6-52. LEDs and Peripheral Multiplexing**

| LED         | Peripheral |
|-------------|------------|
| LED_RED     | STD1       |
| LED_RED2    | SRD1       |
| LED_YELLOW  | SCK1       |
| LED_YELLOW2 | SC10       |
| LED_GREEN   | SC11       |
| LED_GREEN2  | SC12       |

### 6.7.2 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.

### 6.7.3 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 either of two levels: a device-independent interface and a device-dependent LED driver interface. The device-independent interface invokes the device-dependent LED driver interface.

The device-independent API to the LED interface is standard and more portable than the device-dependent LED interface, but less efficient. The device-dependent LED driver provides a

non-standard interface that is more efficient than the device-independent interface, but less portable.

Your application may use either the device-independent interface or the device-dependent 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 device-independent *open* function with any device-dependent LED Driver function call, nor may you use the File Descriptor returned by the device-dependent driver *ledOpen* call with any device-independent interface function call.

You must include *led.h* in your application to utilize the LED Driver API. The following information is found in the public header file *led.h*:

#### Public Interface Function(s):

```

/*****
* The device-independent 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 device-dependent 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

## 6.7.4 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.





### 6.7.4.1 open

Call(s):

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

Arguments:

Table 6-53. 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 6-25](#).

6.7.4.2 close

Call(s):

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

Arguments:

Table 6-54. LED Driver Arguments - close

|          |    |                                                                              |
|----------|----|------------------------------------------------------------------------------|
| FileDesc | in | LED device descriptor returned by open call and used in ioctl 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 6-25.

### 6.7.4.3 *ioctl*

#### Call(s):

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

#### Arguments:

**Table 6-55. 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 <a href="#">Table 6-56</a>                                               |

**Table 6-56. LEDs for Specific EVM Boards (*pParams* - input parameter)**

| DSP56858EVM                                                                 |
|-----------------------------------------------------------------------------|
| LED_RED<br>LED_RED2<br>LED_YELLOW<br>LED_YELLOW2<br>LED_GREEN<br>LED_GREEN2 |

**Table 6-57. LED *ioctl* Commands**

| Cmd        | <i>pParams</i>                               | Return | Result                 |
|------------|----------------------------------------------|--------|------------------------|
| LED_OFF    | Specific LED; see <a href="#">Table 6-56</a> | None   | Turns specific LED off |
| LED_ON     | Specific LED; see <a href="#">Table 6-56</a> | None   | Turns specific LED on  |
| LED_TOGGLE | Specific LED; see <a href="#">Table 6-56</a> | 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 6-25](#) toggles a green LED.

#### Code Example 6-25. 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(decFD);

/*****/
```



## 6.7.5 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.

### 6.7.5.1 *ledOpen*

**Call(s):**

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

**Table 6-58. Low-Level LED Driver Arguments - *ledOpen***

|              |           |                                                |
|--------------|-----------|------------------------------------------------|
| <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              |

**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 6-26](#).



## 6.7.5.2 *ledClose*

**Call(s):**

```
int ledClose (types_tHandle FileDesc);
```

**Table 6-59. 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 6-26](#).

## 6.7.5.3 *ledIoctl*

### Call(s):

```
UWord16 ledIoctl(types_tHandle FileDesc, UWord16 Cmd, UWord16 led, const char
 *ledDevice);
```

**Table 6-60. 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_RED, 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 6-26](#).

### Code Example 6-26. 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);
}
*****/

```



## 6.7.6 LED Driver Tests

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

### 6.7.6.1 LED Driver Test

This test flashes all six LEDs at the same time.

#### 6.7.6.1.1 Set-up for LED Driver Test

##### EVM Jumper Settings:

Use default settings as shown in the **DSP56858 Evaluation Module User's Manual**.

#### 6.7.6.1.2 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 six LEDs should begin flashing at the same time

## 6.7.7 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.

## 6.8 PC Master Software SCI Communication Driver

### 6.8.1 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 writes the application command status for the user application. The *pcmasterdrvRecorder* samples data into an internal data buffer. This driver is automatically installed and initialized simply 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, please refer to the **PC Master User's Manual**.

### 6.8.2 Initialization

The PC master software driver has default configurations, defined in *config.h*, which can be redefined in the *appconfig.h* for that project. These settings are explained in [Table 6-61](#).

**Table 6-61. PC Master *appconfig.h* Settings**

| PC Master 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          |

### 6.8.3 API Definitions

This section defines the Application Programming Interface (API) and public data structures. For a more detailed description of this interface, see [Section 6.8.4](#).

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 6-62](#)

```

/*****
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 6-62. 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 <a href="#">Table 6-63</a> |
| <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 6-63](#).

**Table 6-63. 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<br>0- 1 ( $10^0$ )<br>1 - m ( $10^{-3}$ )<br>2- u ( $10^{-6}$ )<br>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



## 6.8.4 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.

## 6.8.4.1 *pcmasterdrvInit* - Initialize PC Master Software Communication

**Call(s):**

```
Word16 pcmasterdrvInit(sPCMasterComm *p_sPCMasterComm);
```

**Arguments:**

**Table 6-64. *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

### 6.8.4.2 *pcmasterdrvGetAppCmdSts* - Get Application Command Status

**Call(s):**

```
UWord16 pcmasterdrvGetAppCmdSts(void);
```

**Arguments:** None

**Description:** This function confirms whether or not 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 6-62](#) 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 6.8.4.3](#).

**Code Example:** None



### 6.8.4.3 *pcmasterWriteAppCmdSts* - Write Application Command Status

Call(s):

```
Word16 pcmasterdrvWriteAppCmdSts(UWord16 state);
```

**Table 6-65. *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 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 *pcmastercrvWriteAppCmdSts* function

**Returns:** The *pcmastercrvWriteAppCmdSts* 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 <= *state* <= 0x00FD

**Special Issues:** None

**Code Example:** None

#### 6.8.4.4 *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

### 6.8.4.5 *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 6-62](#) and [Table 6-63](#) 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

## 6.8.5 PC Master Software Tests

Currently, there are no stand-alone PC master software test applications.

## 6.8.6 PC Master Software Applications

### 6.8.6.1 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 the DSP56858EVM increments three internal variables (*test16*, *fast16*, *test32*) to obtain sawtooth signals, which it can display.

#### 6.8.6.1.1 Set-up for PC Master Software Demo

##### EVM Jumper Settings:

Use default settings shown in the **DSP56858 Evaluation Module User's Manual**.

#### 6.8.6.1.2 Procedure for PC Master Software Demo

- Using CodeWarrior, open:  
...\\sdk\\src\\dsp56858evm\\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\\dsp56858evm\\nos\\applications\\bsp\\pcmaster\\pcmaster\_test.pmp
- At this time, all prepared variables should be read periodically. The scope also is 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 6-3](#).

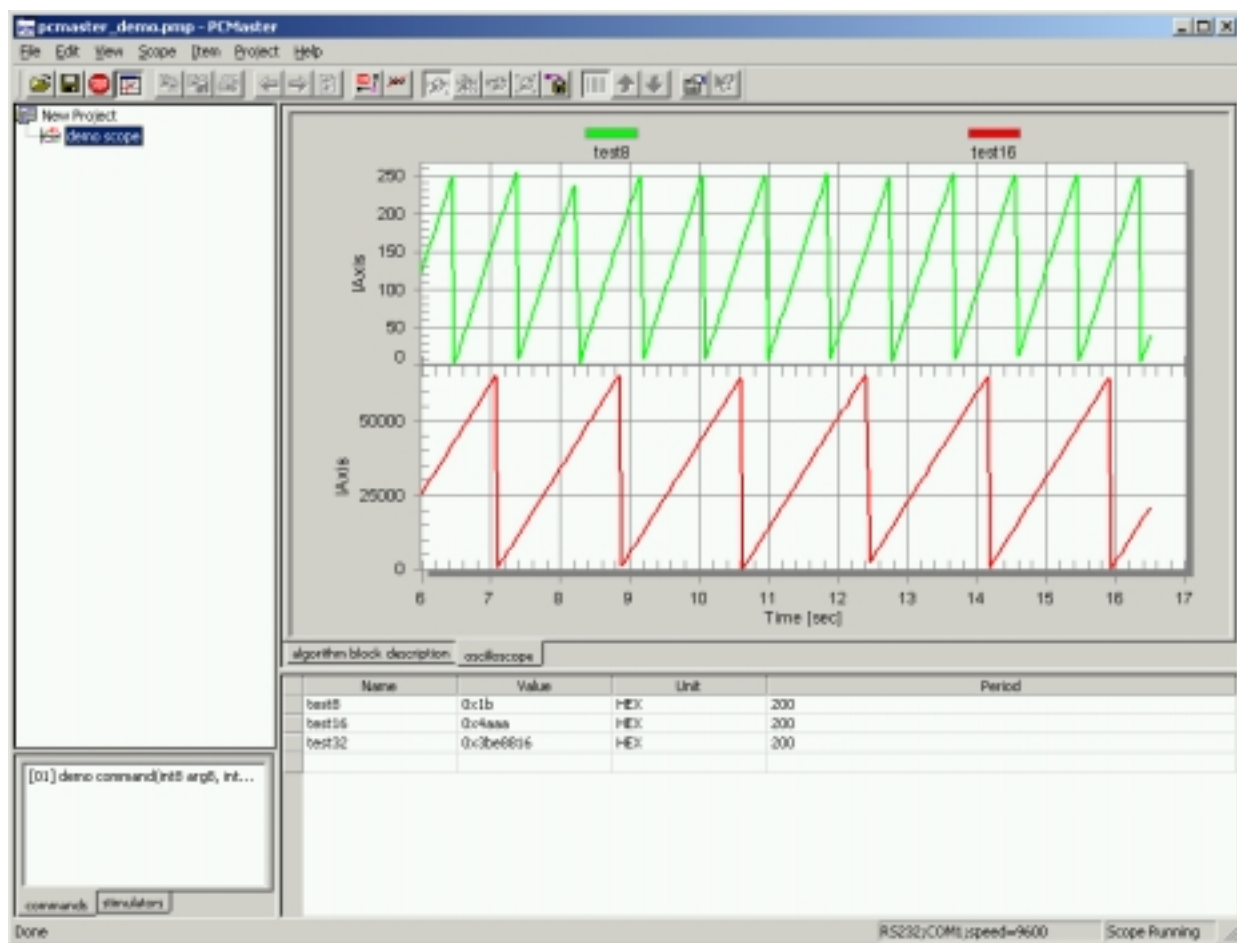


Figure 6-3. PC Master Software Application Window

## 6.9 Serial Data Flash Driver

The Atmel AT45DB011 device is a 1Mbit Serial Data Flash. It is connected to DSP56858 SPI0 interface on the DSP56858EVM 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.

### 6.9.1 Introduction

The Serial Data Flash driver is designed for the Atmel AT45DB011 SPI Data Flash hardware located on the DSP56858EVM 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 read 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.

## 6.9.2 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.

## 6.9.3 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 Codec 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



## 6.9.4 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.

### 6.9.4.1 *open*

#### Call(s):

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

#### Arguments:

**Table 6-66. 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 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 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 6-27](#).



## 6.9.4.2 **write**

**Call(s):**

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

**Arguments:**

**Table 6-67. 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 6-27](#).

### 6.9.4.3 *read*

Call(s):

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

Arguments:

**Table 6-68. 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*. This *ssize\_t* is the number of received bytes.

**Code Example:** See [Code Example 6-27](#).



## 6.9.4.4 *close*

### Call(s):

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

### Arguments:

**Table 6-69. Serial Data Flash Driver Arguments - *close***

|                 |    |                                                                                                                   |
|-----------------|----|-------------------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | in | 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*. This *int* is always zero.

**Code Example:** See [Code Example 6-27](#).

### 6.9.4.5 *ioctl*

**Call(s):**

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

**Arguments:**

**Table 6-70. 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<br>SERIAL_DATAFLASH_MODE_VERIFY: A pointer to a boolean value<br>If "True", verification mode is on; if "False", verification is off |

**Table 6-71. 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*. This *UWord16* is 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** writes 0xA5C3 to the entire Data Flash, reads back, and verifies that the entire Data Flash contains 0xA5C3.

## Code Example 6-27. 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);

*****/

```

## 6.9.5 Low-Level Device Drivers I/O API Specifications

Currently, there are no low-level device driver I/O API specifications.

## 6.9.6 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.

### 6.9.6.1 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 DSP56858EVM.

#### 6.9.6.1.1 Set-up for Serial Data Flash Driver Test

##### EVM Jumper Settings:

Use the default settings shown in the **DSP56858 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.

#### 6.9.6.1.2 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 verification fails for any reason, a message indicating the failure is displayed

## 6.9.7 Serial Data Flash Driver Demo

Currently, there are no demos that use the Serial Data Flash driver.

## 6.10 TDC1 Driver

### 6.10.1 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 DSP56858EVM 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.

### 6.10.2 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 6-72](#).

**Table 6-72. 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<br><br>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.<br><br>Default = 8                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| TDC1_DAA_RX_CALLBACK       | A user function for the TDC1 driver to call when it has received at least <code>TDC1_DAA_RX_CALLBACK_LEVEL</code> samples in its internal TDC1 DAA RX buffer. The user function must be of type <i>void</i> :<br><br><pre>typedef void (*tdc1_tCallback) (void *pCallbackArg, int MaxNBytes);</pre> <p>The parameters to the callback function are:</p> <p><i>pCallbackArg</i> - This is a pointer to the user's callback argument</p> <p><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</p> <p>Default = NULL</p> |
| TDC1_DAA_RX_CALLBACK_ARG   | A value that the user would like passed to the <code>TDC1_DAA_RX_CALLBACK</code> 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.<br><br>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 <code>TDC1_DAA_RX_CALLBACK</code> function.<br><br>Default = NULL                                                                                                                                |

Table 6-72. TDC1 Configuration Items for *appconfig.h* (Continued)

| TDC1 #define                 | Explanation and Default Value                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| TDC1_DAA_TX_CALLBACK_LEVEL   | <p>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.</p> <p>Default = 2</p>                                                                                                                                                                                                                                                                                                                                                                                                               |
| TDC1_DAA_TX_CALLBACK         | <p>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>:</p> <pre>typedef void (*tdc1_tCallback) (void *pCallbackArg, int MaxNBytes);</pre> <p>The parameters to the callback function are:</p> <p><i>pCallbackArg</i> - This is a pointer to the user's callback argument</p> <p><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</p> <p>Default = NULL</p> |
| TDC1_DAA_TX_CALLBACK_ARG     | <p>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.</p> <p>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.</p> <p>Default = NULL</p>                                                                                                                                      |
| TDC1_CODEC_FIFO_SIZE         | <p>Size of the TDC1 codec FIFO in 16-bit words</p> <p>Default = 32</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| TDC1_CODEC_RX_CALLBACK_LEVEL | <p>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.</p> <p>Default = 8</p>                                                                                                                                                                                                                                                                                                                                                                                                                       |

Table 6-72. TDC1 Configuration Items for *appconfig.h* (Continued)

| TDC1 #define                 | Explanation and Default Value                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| TDC1_CODEC_RX_CALLBACK       | <p>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>:</p> <pre>typedef void (*tdc1_tCallback) (void *pCallbackArg, int MaxNBytes);</pre> <p>The parameters to the callback function are:</p> <p><i>pCallbackArg</i> - This is a pointer to the user's callback argument</p> <p><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></p> <p>Default = NULL</p> |
| TDC1_CODEC_RX_CALLBACK_ARG   | <p>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.</p> <p>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.</p> <p>Default = NULL</p>                                                                                                                                  |
| TDC1_CODEC_TX_CALLBACK_LEVEL | <p>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.</p> <p>Default = 2</p>                                                                                                                                                                                                                                                                                                                                                                                                           |

Table 6-72. TDC1 Configuration Items for *appconfig.h* (Continued)

| TDC1 #define                 | Explanation and Default Value                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| TDC1_CODEC_TX_CALLBACK       | <p>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>:</p> <pre>typedef void (*tdc1_tCallback) (void *pCallbackArg, int MaxNBytes);</pre> <p>The parameters to the callback function are:</p> <p><i>pCallbackArg</i> - This is a pointer to the user's callback argument</p> <p><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></p> <p>Default = NULL</p> |
| TDC1_CODEC_TX_CALLBACK_ARG   | <p>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.</p> <p>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.</p> <p>Default = NULL</p>                                                                                                                             |
| TDC1_3000_LINE_IN_GAIN       | <p>TDC1 Audio Codec Line Input Gain Value</p> <p>Possible values are 0db, 10db and 20db:<br/> TDC1_3000_LINE_IN_20DB<br/> TDC1_3000_LINE_IN_10DB<br/> TDC1_3000_LINE_IN_0DB</p> <p>Default = TDC1_3000_LINE_IN_0DB</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| TDC1_3000_LINE_IN_MUTE_STATE | <p>TDC1 Audio Codec Line In Mute State</p> <p>Possible values are:<br/> TDC1_3000_LINE_IN_MUTE and<br/> TDC1_3000_LINE_IN_NOT_MUTE</p> <p>Default = TDC1_LINE_IN_NOT_MUTE</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |

Table 6-72. TDC1 Configuration Items for *appconfig.h* (Continued)

| TDC1 #define                   | Explanation and Default Value                                                                                                                                                                                                                                                                |
|--------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| TDC1_3000_MIC_GAIN             | <p>TDC1 Audio Codec Microphone Gain value</p> <p>Possible values are 0db, 10db, 20db and 30db:<br/> TDC1_3000_MIC_30DB<br/> TDC1_3000_MIC_20DB<br/> TDC1_3000_MIC_10DB<br/> TDC1_3000_MIC_0DB</p> <p>Default = TDC1_3000_MIC_0DB</p>                                                         |
| TDC1_3000_MIC_MUTE_STATE       | <p>TDC1 Audio Codec Microphone Mute</p> <p>Possible values are:<br/> TDC1_3000_MIC_MUTE<br/> TDC1_3000_MIC_NOT_MUTE</p> <p>Default = TDC1_3000_MIC_MUTE</p>                                                                                                                                  |
| TDC1_3000_HNDST_MUTE_STATE     | <p>TDC1 Audio Codec Handset Mute</p> <p>Possible values are:<br/> TDC1_3000_HNDST_MUTE<br/> TDC1_3000_HNDST_NOT_MUTE</p> <p>Default = TDC1_3000_HNDST_MUTE</p>                                                                                                                               |
| TDC1_3000_ADC_GAIN_VALUE       | <p>TDC1 Audio ADC Gain</p> <p>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_CODEC_GAIN_FROM_PERCENT(x)", where "x" is a value from 0-100.</p> <p>Default = 0db</p> |
| TDC1_3000_LINE_OUT_MUTE_STATE  | <p>TDC1 Audio Codec Line Out Mute State</p> <p>Possible values are:<br/> TDC1_3000_LINE_OUT_MUTE_ENABLE<br/> TDC1_3000_LINE_OUT_MUTE_DISABLE</p> <p>Default = TDC1_3000_LINE_OUT_MUTE_ENABLE</p>                                                                                             |
| TDC1_3000_HNDST_OUT_MUTE_STATE | <p>TDC1 Audio Codec Handset Out Mute State</p> <p>Possible values are:<br/> TDC1_HNDST_OUT_MUTE_ENABLE<br/> TDC1_HNDST_OUT_MUTE_DISABLE</p> <p>Default = TDC1_HNDST_OUT_MUTE_DISABLE</p>                                                                                                     |

**Table 6-72. TDC1 Configuration Items for *appconfig.h* (Continued)**

| TDC1 #define                    | Explanation and Default Value                                                                                                                                                                                                                                                             |
|---------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| TDC1_3000_DAC_GAIN_VALUE        | <p>TDC1 Audio DAC Gain</p> <p>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.</p> <p>Default = 0db</p> |
| TDC1_3000_SPKR_LEFT_MUTE_STATE  | <p>TDC1 Audio Codec Speaker Left Mute State</p> <p>Possible values are:<br/> TDC1_3000_SPKR_LEFT_MUTE_ENABLE<br/> TDC1_3000_SPKR_LEFT_MUTE_DISABLE</p> <p>Default = TDC1_3000_SPKR_LEFT_MUTE_ENABLE</p>                                                                                   |
| TDC1_3000_SPKR_RIGHT_MUTE_STATE | <p>TDC1 Audio Codec Speaker Left Mute State</p> <p>Possible values are:<br/> TDC1_3000_SPKR_RIGHT_MUTE_ENABLE<br/> TDC1_3000_SPKR_RIGHT_MUTE_DISABLE</p> <p>Default = TDC1_3000_SPKR_LEFT_MUTE_ENABLE</p>                                                                                 |
| TDC1_3000_LINE_OUT_ATT          | <p>TDC1 Audio Codec Line Out Attenuation</p> <p>Possible values are:<br/> TDC1_3000_LINE_OUT_ATT_0DB<br/> TDC1_3000_LINE_OUT_ATT_6DB<br/> TDC1_3000_LINE_OUT_ATT_12DB<br/> TDC1_3000_LINE_OUT_ATT_18DB</p> <p>Default = TDC1_3000_LINE_OUT_ATT_0DB</p>                                    |
| TDC1_3000_SPK_OUT_ATT           | <p>TDC1 Audio Codec Speaker Out Attenuation</p> <p>Possible values are:<br/> TDC1_3000_SPKR_OUT_ATT_0DB<br/> TDC1_3000_SPKR_OUT_ATT_6DB<br/> TDC1_3000_SPKR_OUT_ATT_12DB<br/> TDC1_3000_SPKR_OUT_ATT_18DB</p> <p>Default = TDC1_3000_SPKR_OUT_ATT_0DB</p>                                 |
| TDC1_3021_AOUT_TX_ATT           | <p>TDC1 DAA Codec TX Attenuation</p> <p>Possible values are:<br/> TDC1_3021_AOUT_TX_20DB_ATT<br/> TDC1_3021_AOUT_TX_26DB_ATT<br/> TDC1_3021_AOUT_TX_32DB_ATT<br/> TDC1_3021_AOUT_TX_MUTE</p> <p>Default = TDC1_3021_AOUT_TX_20DB_ATT</p>                                                  |

Table 6-72. TDC1 Configuration Items for *appconfig.h* (Continued)

| TDC1 #define              | Explanation and Default Value                                                                                                                                                                                                                                                                                         |
|---------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| TDC1_3021_AOUT_RX_ATT     | <p>TDC1 DAA Codec RX Attenuation</p> <p>Possible values are:<br/> TDC1_3021_AOUT_RX_20DB_ATT<br/> TDC1_3021_AOUT_RX_26DB_ATT<br/> TDC1_3021_AOUT_RX_32DB_ATT<br/> TDC1_3021_AOUT_RX_MUTE</p> <p>Default = TDC1_3021_AOUT_RX_20DB_ATT</p>                                                                              |
| TDC1_DAA_N2_M2_DIVIDE_REG | <p>TDC1 DAA N2 M2 divider</p> <p>Possible values are:<br/> TDC1_3021_SAMPLE_AT_7200<br/> TDC1_3021_SAMPLE_AT_8000<br/> TDC1_3021_SAMPLE_AT_8229<br/> TDC1_3021_SAMPLE_AT_8400<br/> TDC1_3021_SAMPLE_AT_9000<br/> TDC1_3021_SAMPLE_AT_9600<br/> TDC1_3021_SAMPLE_AT_10286</p> <p>Default =TDC1_3021_SAMPLE_AT_7200</p> |
| TDC1_DAA_CGM_DIVIDE_REG   | <p>TDC1 DAA CGM Divide Register</p> <p>Possible values are:<br/> 0<br/> 1</p> <p>Default = 1</p>                                                                                                                                                                                                                      |
| TDC1_3021_TX_MUTE_STATE   | <p>TDC1 DAA TX Mute State</p> <p>Possible values are:<br/> TDC1_TX_MUTE<br/> TDC1_TX_NOT_MUTE</p> <p>Default = TDC1_TX_NOT_MUTE</p>                                                                                                                                                                                   |
| TDC1_3021_RX_MUTE_STATE   | <p>TDC1 DAA RX Mute State</p> <p>Possible values are:<br/> TDC1_RX_MUTE<br/> TDC1_RX_NOT_MUTE</p> <p>Default = TDC1_RX_NOT_MUTE</p>                                                                                                                                                                                   |
| TDC1_3021_TX_ATT_VALUE    | <p>TDC1 DAA TX Attenuation</p> <p>Possible values are:<br/> TDC1_3021_TX_ATT_0DB<br/> TDC1_3021_TX_ATT_3DB<br/> TDC1_3021_TX_ATT_6DB<br/> TDC1_3021_TX_ATT_9DB<br/> TDC1_3021_TX_ATT_12DB</p> <p>Default = TDC1_3021_TX_ATT_0DB</p>                                                                                   |



ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005  
**Table 6-72. TDC1 Configuration Items for *appconfig.h* (Continued)**

| TDC1 #define            | Explanation and Default Value                                                                                                                                                                                                            |
|-------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| TDC1_3021_RX_GAIN_VALUE | <p>TDC1 DAA RX Gain Value</p> <p>Possible values are:<br/> TDC1_3021_RX_GAIN_0DB<br/> TDC1_3021_RX_GAIN_3DB<br/> TDC1_3021_RX_GAIN_6DB<br/> TDC1_3021_RX_GAIN_9DB<br/> TDC1_3021_RX_GAIN_12DB</p> <p>Default = TDC1_3021_RX_GAIN_0DB</p> |

**Code Example:** [Code Example 6-28](#) configures and initializes the TDC1 driver.

#### Code Example 6-28. 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

```



### 6.10.3 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:
* int 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

## 6.10.4 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.

### 6.10.4.1 *open*

Call(s):

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

Arguments:

**Table 6-73. 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:<br>BSP_DEVICE_NAME_TDC1_DAA_0<br>BSP_DEVICE_NAME_TDC1_CODEEC_0 |
| <i>OFlag</i>    | <i>in</i>  | Open Mode Flags<br><br>Valid values are:<br>O_RDWR<br>O_NONBLOCKING<br>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 6-29](#).

### 6.10.4.2 *write*

**Call(s):**

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

**Arguments:**

**Table 6-74. 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 6-29](#).

### 6.10.4.3 *read*

Call(s):

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

Arguments:

**Table 6-75. 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 6-29](#).

#### 6.10.4.4 *close*

**Call(s):**

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

**Arguments:**

**Table 6-76. 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 6-29](#).

### 6.10.4.5 *ioctl*

#### Call(s):

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

#### Arguments:

**Table 6-77. 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 6-83](#) 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 6-83](#) for specific control functions.

**Returns:** The *ioctl* call returns a value dependent on the command passed in as an input parameter. [Table 6-83](#) lists these returns based on the specific command.

**Code Example:** [Code Example 6-30](#) 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 6-29. 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);
}

```

## 6.10.5 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.

### 6.10.5.1 *tdc1Open*

**Call(s):**

```
types_tHandle tdc1open(const char *pName, int OFlags);
```

**Arguments:**
**Table 6-78. 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:<br>BSP_DEVICE_NAME_TDC1_DAA_0<br>BSP_DEVICE_NAME_TDC1_CODEEC_0 |
| <i>OFlag</i>    | <i>in</i>  | Open Mode Flags<br>Valid values are:<br>O_RDWR<br>O_NONBLOCKING<br>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 6-30](#).

## 6.10.5.2 *tdc1Write*

### Call(s):

```
ssize_t tdc1Write(types_tHandle FileDesc, const void * pBuffer, size_t NBytes);
```

### Arguments:

**Table 6-79. 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 6-30](#).



### 6.10.5.3 *tdc1Read*

#### Call(s):

```
ssize_t tdc1Read(types_tHandle FileDesc, void * pBuffer, size_t NBytes);
```

#### Arguments:

**Table 6-80. 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 6-30](#).





6.10.5.4 *tdc1Close*

Call(s):

```
int tdc1Close (types_tHandle FileDesc);
```

Arguments:

Table 6-81. 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 6-30](#).

### 6.10.5.5 *tdc1ioctl*

#### Call(s):

```
UWord16 tdc1Ioctl(types_tHandle FileDesc, UWord16 Cmd, void * pParams, const char *pName);
```

#### Arguments:

**Table 6-82. TDC1 Driver Arguments - *tdc1ioctl***

|                 |           |                                                                                                                                                    |
|-----------------|-----------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| <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>tdc1ioctl</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:<br>BSP_DEVICE_NAME_TDC1_DAA_0<br>BSP_DEVICE_NAME_TDC1_CODEC_0 |

**Table 6-83. TDC1 *tdc1ioctl* Commands**

| Cmd                               | <i>pParams</i>      | Return      | 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_CODEC_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_CODEC_FIFO_SIZE depending on the device addressed by FileDesc              |

**Table 6-83. TDC1 *tdc1Ioctl* Commands (Continued)**

| Cmd                         | pParams                                                                                                                                                                                                                                                                                            | Return      | Result                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| TDC1_DEVICE_READ_REG        | <i>tdc1_sRegister</i> *                                                                                                                                                                                                                                                                            | <i>bool</i> | <p>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.</p> <p>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>.</p> |
| TDC1_DEVICE_WRITE_REG       | <i>tdc1_sRegister</i> *                                                                                                                                                                                                                                                                            | <i>bool</i> | <p>Writes a value to the TDC1 device's register. Returns "true" if successful or false if unsuccessful.</p> <p>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.</p>                                                                                                                                              |
| TDC1_DEVICE_SET_SAMPLE_RATE | <i>int</i><br><br>Sample rate values allowed are 7200, 8000, 8229, 8400, 9000, 9600 and 10286<br>TDC1_3021_SAMPLE_AT_7200<br>TDC1_3021_SAMPLE_AT_8000<br>TDC1_3021_SAMPLE_AT_8229<br>TDC1_3021_SAMPLE_AT_8400<br>TDC1_3021_SAMPLE_AT_9000<br>TDC1_3021_SAMPLE_AT_9600<br>TDC1_3021_SAMPLE_AT_10286 | <i>bool</i> | <p>Sets the TDC1 DAA and Codec sample rates. Returns "true" if successful or "false" if unsuccessful.</p>                                                                                                                                                                                                                                                                                                                                      |
| TDC1_DEVICE_MUTE_HDST_IN    | <i>bool</i><br><br>TRUE = Mute<br>FALSE = Unmute                                                                                                                                                                                                                                                   | <i>bool</i> | <p>Sets the state of the TDC1 codec's handset input mute. Returns "true" if successful or "false" if unsuccessful.</p>                                                                                                                                                                                                                                                                                                                         |
| TDC1_DEVICE_MUTE_HDST_OUT   | <i>bool</i><br><br>TRUE = Mute<br>FALSE = Unmute                                                                                                                                                                                                                                                   | <i>bool</i> | <p>Sets the state of the TDC1 codec's handset output mute. Returns "true" if successful or "false" if unsuccessful.</p>                                                                                                                                                                                                                                                                                                                        |

**Table 6-83. TDC1 *tdc1loctl* Commands (Continued)**

| Cmd                       | pParams                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     | Return      | Result                                                                                                                                                          |
|---------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| TDC1_DEVICE_MUTE_MIC      | <i>bool</i><br><br>TRUE = Mute<br>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><br><br>TRUE = Mute<br>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><br><br>TRUE = Mute<br>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><br><br>TRUE = Mute<br>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><br><br>For the <b>Codec</b> 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.<br><br>For the <b>DAA</b> 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 6-83. TDC1 *tdc1loctl* Commands (Continued)**

| Cmd                      | pParams                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        | Return      | Result                                                                                                                                                           |
|--------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| TDC1_DEVICE_SET_TX_GAIN  | <i>unsigned int</i><br><br>For the <b>Codec</b> 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.<br><br>For the <b>DAA</b> 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><br>TRUE = OFF HOOK<br>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><br><br>TRUE = Mute<br>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><br><br>TRUE = Mute<br>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 6-83](#) for specific control functions.

**Returns:** The *ioctl* call returns a value dependent on the command passed in as an input parameter. [Table 6-83](#) lists these returns based on the specific command.

**Code Example:** [Code Example 6-30](#) 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 6-30. 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++)
 {
 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;
 }
}

```

ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

```

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;

 /* 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;
```

```
}
```

```
}
```

## 6.10.6 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.

## 6.10.7 TDC1 Driver Applications

### 6.10.7.1 TDC1 Loopback Application

This application performs a simple audio loopback using TDC1 hardware and a DSP56858 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.

#### 6.10.7.1.1 Set-up for TDC1 Loopback Application

##### EVM Jumper Settings:

- Make the following changes to the default jumper settings for the DSP56858EVM 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 DSP56858.
- The complete list of jumper settings are shown in the **DSP56858 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 DSP56858 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.)

#### 6.10.7.1.2 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 6.10.7.1.1](#)

- 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.

## 6.11 IDC Driver

### 6.11.1 Introduction

The Internet Daughter Card (IDC) driver allows a DSP application to write and read frames of data to and from the 10BASET part (CIRRUS LOGIC CS8900A) on the IDC board. Please refer to the CIRRUS LOGIC CS8900A Product Data Sheet for more information on the part.

### 6.11.2 Initialization

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

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

```
#define INCLUDE_IDC
```

Defining the preprocessor variable, INCLUDE\_IDC, directs the SDK to include the IDC driver in the executable image. If the preprocessor variable INCLUDE\_IDC is not defined, then the IDC 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 IDC Driver. A list of these variables is provided in [Table 6-84](#).

**Table 6-84. IDC Configuration Items for *appconfig.h***

| IDC Defined Constants | Type          | Explanation and Default Value                                                                                |
|-----------------------|---------------|--------------------------------------------------------------------------------------------------------------|
| IDC_RX_CALLBACK       | idc_tCallback | This callback is called when the IDC driver receives a frame of data via RX interrupt<br><br>default = NULL  |
| IDC_TX_CALLBACK       | idc_tCallback | This callback is called when the IDC driver transmits a frame of data via TX interrupt<br><br>default = NULL |

Table 6-84. IDC Configuration Items for *appconfig.h* (Continued)

| IDC Defined Constants | Type          | Explanation and Default Value                                                      |
|-----------------------|---------------|------------------------------------------------------------------------------------|
| IDC_RX_CALLBACK_ARG   | idc_tCallback | This allows a user to pass in an argument to the RX callback<br><br>default = NULL |
| IDC_TX_CALLBACK_ARG   | idc_tCallback | This allows a user to pass in an argument to the TX callback<br><br>default = NULL |
| IDC_RX_CALLBACK_LEVEL | idc_tCallback | Currently not implemented                                                          |
| IDC_TX_CALLBACK_LEVEL | idc_tCallback | Currently not implemented                                                          |
| IDC_MAC_ADDRESS_1     | UInt8         | IDC chip MAC address 1<br><br>default 0x12                                         |
| IDC_MAC_ADDRESS_2     | UInt8         | IDC chip MAC address 2<br><br>default 0x34                                         |
| IDC_MAC_ADDRESS_3     | UInt8         | IDC chip MAC address 3<br><br>default 0x56                                         |
| IDC_MAC_ADDRESS_4     | UInt8         | IDC chip MAC address 4<br><br>default 0x78                                         |
| IDC_MAC_ADDRESS_5     | UInt8         | IDC chip MAC address 5<br><br>default 0x9A                                         |
| IDC_MAC_ADDRESS_6     | UInt8         | IDC chip MAC address 6<br><br>default 0xBC                                         |

**Code Example:** [Code Example 6-31](#) configures and initializes the IDC driver.

#### Code Example 6-31. Sample *appconfig.h* code

```

#ifndef __APPCONFIG_H
#define __APPCONFIG_H

/*****
 *
 * Include needed SDK components below by changing #undef to #define.
 * Refer to ../config/config.h for complete list of all components and
 * component default initialization.
 *****/
#define INCLUDE_BSP /* BSP support - includes SIM, COP, PLL, and INTC
*/

```



```

#define INCLUDE_IO /* I/O support */
#define INCLUDE_IDC /* IDC support */

/*****
 *
 * Overwrite default component initializations from config/config.h
 * using #defines here
 *
 *****/
extern void idcRxCallback(void *pCallbackArg, int MaxNBytes);
extern void idcTxCallback(void *pCallbackArg, int MaxNBytes);

#define IDC_RX_CALLBACK idcRxCallback
#define IDC_RX_CALLBACK_ARG NULL

#define IDC_TX_CALLBACK idcTxCallback
#define IDC_TX_CALLBACK_ARG NULL

#define IDC_MAC_ADDRESS_1 0xAA
#define IDC_MAC_ADDRESS_2 0xBB
#define IDC_MAC_ADDRESS_3 0xCC
#define IDC_MAC_ADDRESS_4 0xDD
#define IDC_MAC_ADDRESS_5 0xEE
#define IDC_MAC_ADDRESS_6 0xFF

#endif

```



### 6.11.3 Device-Independent 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 IDC Driver may automatically be included in an SDK Embedded Project by inserting the following line in the *appconfig.h* file:

```
#define INCLUDE_IDC
```

The following information may be found in the public header file *idc.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);
*****/

```

#### Public Data Structure(s):

/\* Structure to be passed into ioctl and idcIoctl via pParams.

**NOTE : Values and ranges for Data and Register values depend on a particular register. Please refer to the CS8900A Product Data Sheet for more information on particular registers and configurations. \*/**

```

typedef struct
{
 Word16 Data; /* Data for CS8900A chip on IDC board */
 UWord16 Register; /* CS8900A chip register on IDC board */
}idc_sParams;

/* This is the destination address for the frame a user wants to transmit.
This structure needs to be filled up and passed into a write as a member
of an idc_sTxPacket. */
typedef struct
{
 UInt8 Address[6]; /* Destination Address for TX */
}idc_sAddress;

/* Transmit packet to be used in write function calls */
typedef struct
{
 UInt8 * pData; /* Packet data to be sent through the ethernet device */
 idc_sAddress DstAddr; /* Destination Address for TX */
 UWord16 Length; /* Length of the packet data */
 UWord16 Type; /* Packet Type, for example 0x0008 corresponds to a TCP/IP
 packet*/
}idc_sTxPacket;

/* Receive packet to be used in read function calls */
typedef struct
{
 UInt8 PacketData[1518]; /* Packet data received from the ethernet device.
 Maximum frame length device allows is 1518, minimum is 64 bytes */

```





```
}idc_sRxPacket;

/* Callback structure */
typedef struct
{
 idc_tCallback pCallback;
 void * pCallbackArg;
 int CallbackLevel; /* Currently not implemented */
}idc_sCallback;
```

## 6.11.4 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.

### 6.11.4.1 **open**

**Call(s):**

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

**Arguments:**
**Table 6-85. IDC Driver Arguments - open**

|                 |            |                                                                                                             |
|-----------------|------------|-------------------------------------------------------------------------------------------------------------|
| <i>pName</i>    | <i>in</i>  | <i>BSP_DEVICE_NAME_IDC</i>                                                                                  |
| <i>OFlag</i>    | <i>in</i>  | Use O_RDWR - default<br>Use O_BLOCK - default                                                               |
| <i>FileDesc</i> | <i>out</i> | IDC device descriptor returned by <i>open</i> call and used in <i>write/read/close/ioctl</i> function calls |

**Description:** The *open* call opens the IDC driver for operations by putting the IDC board into a memory-mapped mode and appropriately initializing the chip's registers. The function returns a file descriptor used by subsequent *read/write/close/ioctl* function calls. The IDC device must be opened before any calls are made to other IDC 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 IDC device and is used by the IDC driver's *read*, *write*, *close*, and *ioctl* functions. A "-1" is returned when the *open* call is not successful.

**Code Example:** See [Code Example 6-32](#).

### 6.11.4.2 *write*

**Call(s):**

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

**Arguments:**

**Table 6-86. IDC Driver Arguments - *write***

|                 |           |                                                               |
|-----------------|-----------|---------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | IDC Device descriptor returned by "open" call                 |
| <i>pBuffer</i>  | <i>in</i> | Pointer to a structure of type <i>sldcTxPacket</i>            |
| <i>NBytes</i>   | <i>in</i> | Not used; packet length is in packet header of <i>pBuffer</i> |

**Description:** The *write* call writes the user buffer pointed to by *pBuffer* out of the IDC device. This function blocks until the entire frame has been sent out of the ethernet device.

**Returns:** The *write* call returns a type *ssize\_t*. This *ssize\_t* is the number of transferred bytes.

**Code Example:** See [Code Example 6-32](#).

### 6.11.4.3 *read*

**Call(s):**

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

**Arguments:**
**Table 6-87. IDC Driver Arguments - *read***

|                 |           |                                                                                                      |
|-----------------|-----------|------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | IDC Device descriptor returned by <i>open</i> call                                                   |
| <i>pBuffer</i>  | <i>in</i> | Pointer to a structure of type <i>sldcRxPacket</i> ; allocates buffer for maximum frame size of 1518 |
| <i>Size</i>     | <i>in</i> | Not used; frame receive size can be 64-1518                                                          |

**Description:** The *read* call reads data from IDC device the user buffer pointed by *pBuffer*.

**NOTE:** The *read* function will block until an entire frame of data has been read in. The chip will hold off another *read* interrupt until this is complete.

**Returns:** The *read* call returns a type *ssize\_t*, which is the number of received bytes. A frame receive size can be 64-1518 bytes. The frame contains source address, destination address, packet type, and data.

**Code Example:** See [Code Example 6-32](#).

6.11.4.4 *close*

Call(s):

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

Arguments:

Table 6-88. IDC Driver Arguments - *close*

|                 |           |                                                    |
|-----------------|-----------|----------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | IDC Device descriptor returned by <i>open</i> call |
|-----------------|-----------|----------------------------------------------------|

**Description:** The *close* call closes the IDC driver by resetting the chip and disabling interrupts; it also releases the file descriptor handle.

**Returns:** The *close* call returns a type *int*. This *int* is always zero, indicating the close was successful.

**Code Example:** See [Code Example 6-32](#).

### 6.11.4.5 *ioctl*

#### Call(s):

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

#### Arguments:

**Table 6-89. IDC Driver Arguments - *ioctl***

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

**Table 6-90. IDC *ioctl* Commands**

| Cmd           | <i>pParams</i> | Return      | Result                   |
|---------------|----------------|-------------|--------------------------|
| IDC_REG_WRITE | idc_sParams    | <i>void</i> | Write to an IDC register |
| IDC_REG_READ  | idc_sParams    | <i>int</i>  | Read an IDC register     |

**Description:** The *ioctl* call provides various control functions for the IDC. See the list of commands in [Table 6-90](#) for specific control functions.

**Returns:** The *ioctl* call returns a value dependent on the command passed in as an input parameter. [Table 6-90](#) lists these returns based on the specific command.

**Code Example:** See [Code Example 6-32](#).

**Code Example 6-32. IDC Driver Usage**

```

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

/* Dummy frame length */
#define DATA_TX_LENGTH 1500
#define MAX_FRAME_SIZE 1518

/* Function Prototypes */
void idcRxCallback(void *pCallbackArg, int CallBackLevel);
void idcTxCallback(void *pCallbackArg, int CallBackLevel);

/* Global Data */
idc_sRxPacket IdcRxPacket;
bool bIdcRxFrameAvailable;
bool bIdcTxFrameInProgress;
bool bIdcDeviceEnabled;
types_tHandle IdcFD;

```

```

void main (void)
{
 UWord16 I;
 UWord16 ReturnValue;
 idc_sTxPacket IdcTxPacket;
 idc_sAddress DestinationAddress;
 UInt8 IdcTxBuffer[MAX_FRAME_SIZE];
 idc_sParams IdcReg;
 /* Initialize global variables */
 bIdcRxFrameAvailable = false;
 bIdcTxFrameInProgress = false;
 bIdcDeviceEnabled = false;

 /* The open command opens the IDC Device to receive "Good Frames"
 addressed to its Broadcast or its own Source address */
 IdcFD = open(BSP_DEVICE_NAME_IDC,0); /* Open IDC device */

 IdcReg.Data = 0x0F00;
 IdcReg.Register = (UWord16)IDC_RX_CFG;

 ioctl(IdcFD, IDC_REG_WRITE, (void *)&IdcReg);

 bIdcDeviceEnabled = true;

 DestinationAddress.Address[0] = 0xFF; /* will broadcast */
 DestinationAddress.Address[1] = 0xFF;
 DestinationAddress.Address[2] = 0xFF;
 DestinationAddress.Address[3] = 0xFF;
 DestinationAddress.Address[4] = 0xFF;
 DestinationAddress.Address[5] = 0xFF;

 /* Create TX Packet */
 IdcTxPacket.Length = DATA_TX_LENGTH;
 IdcTxPacket.DstAddr = DestinationAddress;
 IdcTxPacket.Type = 0x0008; /* IP packet type */

 IdcTxPacket.pData = IdcTxBuffer;

 /* Fill up frame data */
 for(I = 0; I < DATA_TX_LENGTH; I++)
 {
 IdcTxPacket.pData[I] = 0x50 + I;
 }

 /* Write the frame to the device */
 ReturnValue = write(IdcFD, &IdcTxPacket, 0);

 /* Wait for a received frame, when frame received send the tx frame again
 */
 while(1)
 {
 if(bIdcRxFrameAvailable == true)
 {
 /* New frame received, clear the bIdcRxFrameAvailable flag
 */

```





```

 bIdcRxFrameAvailable = false;

 if(bIdcTxFrameInProgress == false)
 {
 ReturnValue = write(IdcFD, &IdcTxPacket, 0);
 bIdcTxFrameInProgress = true;
 }
 }

 /* Close the device */
 close(IdcFD);
}

/*****
/* idcRxCallback is called when a good frame has been received by the driver */
/* User can read the frame by calling the read function */
*****/
void idcRxCallback(void *pCallbackArg, int CallBackLevel)
{
 UWord16 ReturnValue;

 if (bIdcDeviceEnabled)
 {
 ReturnValue = read(IdcFD, &IdcRxPacket, 0);

 if (ReturnValue != 0)
 {
 bIdcRxFrameAvailable = true;
 }
 }
}

/*****
/* idcTxCallback is called when a complete frame has been transmitted by the */
/* IDC Device. We get here as a result of a write called elsewhere */
*****/
void idcTxCallback(void *pCallbackArg, int CallBackLevel)
{
 bIdcTxFrameInProgress = false;
}

/*****/

```

## 6.11.5 Device-Dependent IDC Driver API

The following pages specify the Application Programming Interface (API) for the device-dependent interface to the IDC driver.

To use the device-dependent interface to the IDC driver, you must define the `INCLUDE_IDC` variable in the *appconfig.h* file of the SDK Embedded Project you created in CodeWarrior:

```
#define INCLUDE_IDC
```

The user's application code then uses the IDC driver API as follows:

- *idcOpen* call is made to initialize the IDC device
- *idcIoctl* calls are made to control the IDC device
- *idcWrite* calls are made to write data to the IDC (transmit)
- *idcRead* calls are made to read data from the IDC (receive)
- After all IDC operations are completed, the IDC driver is closed via an *idcClose* call

The following sections describe each function of the device-dependent API for the IDC 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.

### 6.11.5.1 *idcOpen*

#### Call(s):

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

#### Arguments:

**Table 6-91. IDC Driver Arguments - *idcOpen***

|                 |            |                                                                                                                            |
|-----------------|------------|----------------------------------------------------------------------------------------------------------------------------|
| <i>pName</i>    | <i>in</i>  | <i>BSP_DEVICE_NAME_IDC</i>                                                                                                 |
| <i>OFlag</i>    | <i>in</i>  | Use O_RDWR - default<br>Use O_BLOCK - default                                                                              |
| <i>FileDesc</i> | <i>out</i> | IDC device descriptor returned by <i>idcOpen</i> call and used in <i>idcWrite/idcRead/idcClose/idcIoctl</i> function calls |

**Description:** See *open* description in [Section 6.11.4.1](#).

**Return:** See *open* description in [Section 6.11.4.1](#).

**Code Example:** See [Code Example 6-33](#).

### 6.11.5.2 *idcWrite*

**Call(s):**

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

**Arguments:**

**Table 6-92. IDC Driver Arguments - *idcWrite***

|                 |           |                                                               |
|-----------------|-----------|---------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | IDC Device descriptor returned by the <i>idcOpen</i> call     |
| <i>pBuffer</i>  | <i>in</i> | Pointer to a structure of type <i>sIdcTxPacket</i>            |
| <i>NBytes</i>   | <i>in</i> | Not used; packet length is in packet header of <i>pBuffer</i> |

**Description:** See *write* description in [Section 6.11.4.2](#).

**Returns:** See *write* description in [Section 6.11.4.2](#).

**Code Example:** See [Code Example 6-33](#).

### 6.11.5.3 *idcRead*

Call(s):

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

Arguments:

**Table 6-93. IDC Driver Arguments - *idcRead***

|                 |           |                                                                                                      |
|-----------------|-----------|------------------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | IDC Device descriptor returned by the <i>idcOpen</i> call                                            |
| <i>pBuffer</i>  | <i>in</i> | Pointer to a structure of type <i>sIdcRxPacket</i> ; allocates buffer for maximum frame size of 1518 |
| <i>Size</i>     | <i>in</i> | Not used; frame receive size can be 64-1518                                                          |

**Description:** See *read* description in [Section 6.11.4.3](#).

**Returns:** See *read* description in [Section 6.11.4.3](#).

**Code Example:** See [Code Example 6-33](#).

### 6.11.5.4 *idcClose*

**Call(s):**

```
int idcClose (types_tHandle FileDesc);
```

**Arguments:**

**Table 6-94. IDC Driver Arguments - *idcClose***

|                 |           |                                                       |
|-----------------|-----------|-------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | IDC Device descriptor returned by <i>idcOpen</i> call |
|-----------------|-----------|-------------------------------------------------------|

**Description:** See *close* description in [Section 6.11.4.4](#).

**Returns:** See *close* description in [Section 6.11.4.4](#).

**Code Example:** See [Code Example 6-33](#).

### 6.11.5.5 *idcloctl*

#### Call(s):

```
UWord16 idcIoctl(int FileDesc, UWord16 Cmd, void * pParams);
```

#### Arguments:

**Table 6-95. IDC Driver Arguments - *idcloctl***

|                 |           |                                                                                          |
|-----------------|-----------|------------------------------------------------------------------------------------------|
| <i>FileDesc</i> | <i>in</i> | IDC device descriptor returned by the <i>idcOpen</i> call                                |
| <i>Cmd</i>      | <i>in</i> | Commands found in <i>idc.h</i> , which are used to control the IDC driver and IDC device |
| <i>pParams</i>  | <i>in</i> | The <i>pParam</i> is of structure type <i>idc_sParams</i> and is found in <i>idc.h</i>   |

**Description:** See *ioctl* description in [Section 6.11.4.5](#).

**Returns:** See *ioctl* description in [Section 6.11.4.5](#).

**Code Example:** See [Code Example 6-33](#).

#### Code Example 6-33. TDC1 Driver Low-Level API Example

```

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

/* Dummy frame length */
#define DATA_TX_LENGTH 1500
#define MAX_FRAME_SIZE 1518

/* Function Prototypes */
void idcRxCallback(void *pCallbackArg, int MaxNBytes);
void idcTxCallback(void *pCallbackArg, int MaxNBytes);

/* Global Data */
idc_sRxPacket IdcRxPacket;
bool bIdcRxFrameAvailable;
bool bIdcTxFrameInProgress;
bool bIdcDeviceEnabled;
types_tHandle IdcFD;

void main (void)
{
 UWord16 I;
 UWord16 ReturnValue;
 idc_sTxPacket IdcTxPacket;
 idc_sAddress DestinationAddress;
 UInt8 IdcTxBuffer[MAX_FRAME_SIZE];
 idc_sParams IdcReg;

```

ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

```

/* Initialize global variables */
bIdcRxFrameAvailable = false;
bIdcTxFrameInProgress = false;
bIdcDeviceEnabled = false;

/* The idcOpen command opens the IDC Device to receive "Good Frames"
 addressed to its Broadcast or its own Source address */
IdcFD = idcOpen(BSP_DEVICE_NAME_IDC,0); /** Open IDC device */

IdcReg.Data = 0x0F00;
IdcReg.Register = (UWord16)IDC_RX_CFG;

idcIoctl(IdcFD, IDC_REG_WRITE, (void *)&IdcReg);

bIdcDeviceEnabled = true;

DestinationAddress.Address[0] = 0xFF; /* will broadcast */
DestinationAddress.Address[1] = 0xFF;
DestinationAddress.Address[2] = 0xFF;
DestinationAddress.Address[3] = 0xFF;
DestinationAddress.Address[4] = 0xFF;
DestinationAddress.Address[5] = 0xFF;

/* Create TX Packet */
IdcTxPacket.Length = DATA_TX_LENGTH;
IdcTxPacket.DstAddr = DestinationAddress;
IdcTxPacket.Type = 0x0008; /* IP packet type */

IdcTxPacket.pData = IdcTxBuffer;

/* Fill up frame data */
for(I = 0; I < DATA_TX_LENGTH; I++)
{
 IdcTxPacket.pData[I] = 0x50 + I;
}

/* Write the frame to the device */
ReturnValue = idcWrite(IdcFD, &IdcTxPacket, 0);

/* Wait for a received frame, when frame received send the tx frame again
*/
while(1)
{
 if(bIdcRxFrameAvailable == true)
 {
 /* New frame received, clear the bIdcRxFrameAvailable flag
*/
 bIdcRxFrameAvailable = false;

 if(bIdcTxFrameInProgress == false)
 {
 ReturnValue = idcWrite(IdcFD, &IdcTxPacket, 0);
 bIdcTxFrameInProgress = true;
 }
 }
}

```





```

 /* Close the device */
 idcClose(IdcFD);
}

/*****
/* idcRxCallback is called when a good frame has been received by the driver */
/* User can read the frame by calling the idcRead function */
*****/
void idcRxCallback(void *pCallbackArg, int MaxNBytes)
{
 UWord16 ReturnValue;

 if (bIdcDeviceEnabled)
 {
 ReturnValue = idcRead(IdcFD, &IdcRxPacket, 0);

 if (ReturnValue != 0)
 {
 bIdcRxFrameAvailable = true;
 }
 }
}

/*****
/* idcTxCallback is called when a complete frame has been transmitted by the */
/* IDC Device. We get here as a result of an idcWrite called elsewhere */
*****/
void idcTxCallback(void *pCallbackArg, int MaxNBytes)
{
 bIdcTxFrameInProgress = false;
}

```



## **6.11.6 IDC Driver Tests**

Currently, there are no specific tests provided for the IDC driver.



## 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 DSP5685x 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.723.1A Speech Codec Library (*g723.lib*)
- G.726 Library (*g726.lib*)
- G.729AB Vocoder Library (*g729ab.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 DSP5685x.

#### 7.1.2.1 Trigonometric Math Function Performance

This section documents the performance of the trigonometric functions on the DSP5685x. [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                   |
| 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 DSP5685x.

### 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 the table 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 the table do not include instruction cycles needed by the *cbitrev* 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,  $n$  is the number of input samples to be processed by the filter and  $f$  is 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 | $50 + n(16 + 11nbiq)$   | 121                    | 0                   | 0                | 4                   |
| Case 2 | $50 + n(20 + 11nbiq)$   |                        |                     |                  |                     |
| Case 3 | $55 + n(22 + 11nbiq)$   |                        |                     |                  |                     |

$n$  = the number of input samples to be processed by the filter

$nbiq$  = the number of filter biquads; each biquad uses five 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             |                     |

**AS** stands for Auto Scale

**BFP** stands for Block Floating Point

**NS** stands for No Scale

### 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 *cbitrev* 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             |                     |

**AS** stands for Auto Scale

**BFP** stands for Block Floating Point

**NS** stands for No Scale

## 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 DSP5685x.

**Table 7-10. V.8bis Memory and MIPS Requirements for Internal Memory on the DSP5685x**

|                                                                                                                       | Program<br>Memory<br>(words) | Data ROM<br>(words) | Software<br>Stack Per<br>Channel<br>(words) | Data RAM<br>Per Channel<br>(words) | MIPS |
|-----------------------------------------------------------------------------------------------------------------------|------------------------------|---------------------|---------------------------------------------|------------------------------------|------|
| <b>Library allocates<br/>memory</b> [includes<br><i>v8bisCreate()</i> and<br><i>v8bisDestroy()</i> ]                  | 9573                         | 557                 | 80                                          | 1899                               | N/A  |
| <b>User application<br/>allocates memory</b> [does<br>not include <i>v8bisCreate()</i><br>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 DS {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 DSP5685x are shown in **Table 7-11**.

**Table 7-11. MIPS on the DSP5685x**

| 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 DSP5685x.

**Table 7-13. V.21 Memory and MIPS Requirements for Internal Memory on the DSP5685x**

|                                                                                                         | Program Memory (words) | Data ROM (words) | Software Stack Per Channel (words) | Data RAM Per Channel (words) | MIPS at 7200Hz Sampling Rate |          |
|---------------------------------------------------------------------------------------------------------|------------------------|------------------|------------------------------------|------------------------------|------------------------------|----------|
|                                                                                                         |                        |                  |                                    |                              | Transmitter                  | Receiver |
| <b>Library allocates memory</b> [includes <i>v21Create()</i> and <i>v21Destroy()</i> ]                  | 2325                   | 535              | 24                                 | 248                          | 0.14                         | 1.0      |
| <b>User application allocates memory</b> [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 Requirements on the DSP5685x**

|                                                                                                               | Program Memory (words) | Data ROM (words) | Software Stack Per Channel (words) | Data RAM Per Channel (words)                              | MIPS        |          |
|---------------------------------------------------------------------------------------------------------------|------------------------|------------------|------------------------------------|-----------------------------------------------------------|-------------|----------|
|                                                                                                               |                        |                  |                                    |                                                           | Transmitter | Receiver |
| <b>Library allocates memory</b> [includes <i>v22bisCreate()</i> and <i>v22bisDestroy()</i> ]                  | 5778                   | 1065             | 23                                 | Only one channel exists<br><br>Data RAM per channel = 310 | 0.94        | 6.21     |
| <b>User application allocates memory</b> [does not include <i>v22bisCreate()</i> and <i>v22bisDestroy()</i> ] | 5744                   | 1065             | 23                                 | Only one channel exists<br><br>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 uncompressed (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 DSP5685x**

|                                                                                                               | Program Memory (words) | Data ROM (words) | Software Stack Per Channel (words) | Data RAM Per Channel (words) | MIPS |
|---------------------------------------------------------------------------------------------------------------|------------------------|------------------|------------------------------------|------------------------------|------|
| <b>Library allocates memory</b> [includes <i>v42bisCreate()</i> and <i>v42bisDestroy()</i> ]                  | 2381                   | 0                | 35                                 | DictionarySize * 4 + 72      | NA   |
| <b>User application allocates memory</b> [does not include <i>v42bisCreate()</i> and <i>v42bisDestroy()</i> ] | 2171                   | 0                | 35                                 | DictionarySize * 4 + 72      | NA   |

**Table 7-16. V.42bis Cycle Count for Internal Memory on the DSP5685x**

| Dictionary Parameters | Encoder Cycle Count                 | Decoder Cycle Count                 |
|-----------------------|-------------------------------------|-------------------------------------|
| P1 = 512, P2 = 6      | MAX count = 7000<br>MIN count = 176 | MAX count = 7000<br>MIN count = 144 |
| P1 = 1024, P2 = 10    | MAX count = 7000<br>MIN count = 176 | MAX count = 7000<br>MIN count = 144 |

P1 is the dictionary size and P2 is the string length.

#### Notes:

- 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 reconverting 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 DSP5685x.

**Table 7-17. DES Memory and MIPS Requirements for Internal Memory on the DSP5685x**

|                                                                                                         | Program Memory (words) | Data ROM (words) | Software Stack Per Channel (words)                                                                           | Data RAM Per Channel (words) | MIPS |
|---------------------------------------------------------------------------------------------------------|------------------------|------------------|--------------------------------------------------------------------------------------------------------------|------------------------------|------|
| <b>Library allocates memory</b> [includes <i>desCreate()</i> and <i>desDestroy()</i> ]                  | 2752                   | 742              | Encrypt (CBC Mode)<br>42<br>Decrypt (CBC Mode)<br>42<br>Encrypt (CFB Mode)<br>42<br>Decrypt (CFB Mode)<br>42 | 901                          | N/A  |
| <b>User application allocates memory</b> [does not include <i>desCreate()</i> and <i>desDestroy()</i> ] | 2292                   | 742              | Encrypt (CBC Mode)<br>42<br>Decrypt (CBC Mode)<br>42<br>Encrypt (CFB Mode)<br>42<br>Decrypt (CFB Mode)<br>42 | 901                          | N/A  |

**Note:** Data RAM figures in **Table 7-17** 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 DSP5685x.

**Table 7-18. DES Cycle Counts for Internal Memory on the DSP5685x**

| Module         | Instruction Cycle Count<br>for CFB Mode | Instruction Cycle Count<br>for CBC Mode |
|----------------|-----------------------------------------|-----------------------------------------|
| Initialization | 20795                                   | 20795                                   |
| Encryption     | 238217                                  | 116596                                  |
| Decryption     | 239262                                  | 117460                                  |

Instruction Cycle Count = Oscillator Clock Cycles

**Notes:**

- The cycle counts in **Table 7-18** are obtained for an input block (data to be encrypted or decrypted) of size 128 characters

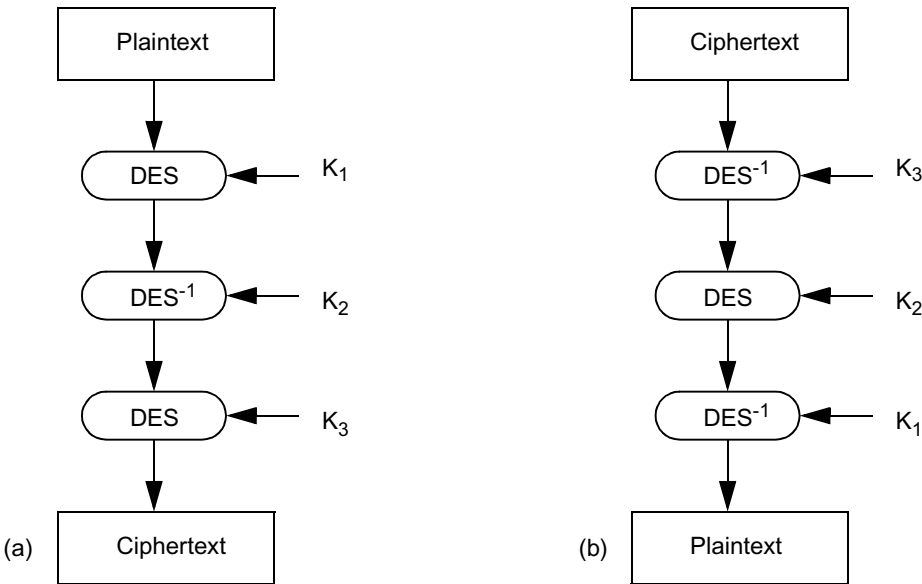
This library is described in more detail in the document **Data Encryption Standard (DES) Library**.

## 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 reconvertng 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:

$K^1$ ,  $K^2$  and  $K^3$  = 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 DSP5685x.

Table 7-19. 3DES Memory and MIPS Requirements for Internal Memory on the DSP5685x

|                                                                                                               | Program Memory (words) | Data ROM (words) | Software Stack Per Channel (words) | Data RAM Per Channel (words) | MIPS |
|---------------------------------------------------------------------------------------------------------------|------------------------|------------------|------------------------------------|------------------------------|------|
| <b>Library allocates memory</b><br>[includes <i>triDesCreate()</i> and <i>triDesDestroy()</i> ]               | 2033                   | 742              | Encrypt<br>41<br>Decrypt<br>41     | 1012                         | N/A  |
| <b>User application allocates memory</b> (does not include <i>triDesCreate()</i> and <i>triDesDestroy()</i> ) | 1572                   | 742              | Encrypt<br>41<br>Decrypt<br>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 DSP5685x.

**Table 7-20. 3DES Cycle Count for Internal Memory on the DSP5685x**

| 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,  $x.\exp(\Phi(n)) \pmod n = 1$ , if  $x$  and  $n$  are relatively prime. Also,  $x.\exp(k\Phi(n)+1) \pmod n = x$ , 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 signaturing 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 DSP5685x.

**Table 7-21. RSA Memory and MIPS Requirements for Internal Memory on the DSP5685x**

|                                                                                                                                                                | Program Memory (words) | Data ROM (words) | Software Stack Per Channel (words) | Data RAM Per Instance (words)                     | MIPS |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------|------------------|------------------------------------|---------------------------------------------------|------|
| <b>Library allocates memory</b> [includes <i>rsaEncCreate()</i> , <i>rsaEncDestroy()</i> , <i>rsaDecCreate()</i> and <i>rsaDecDestroy()</i> ]                  | 2652                   | 0                | Encrypt 20<br>Decrypt 18           | 153 + $n*26$<br><br>$n$ = length of modulo buffer | N/A  |
| <b>User application allocates memory</b> [does not include <i>rsaEncCreate()</i> , <i>rsaEncDestroy()</i> , <i>rsaDecCreate()</i> and <i>rsaDecDestroy()</i> ] | 2350                   | 0                | Encrypt 20<br>Decrypt 18           | 153 + $n*26$<br><br>$n$ = length of modulo buffer | N/A  |

**Table 7-22** shows the cycle count for RSA on the DSP5685x, when data memory is internal.

**Table 7-22. RSA Cycle Count for Internal Memory on the DSP5685x**

| 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 DSP5685x.



**Table 7-23. AEC Memory and MIPS Requirements for Internal Memory on the DSP5685x**

|                                                                                                                          | Program<br>Memory<br>(words) | Data ROM<br>(words) | Software<br>Stack Per<br>Channel<br>(words) | Data RAM<br>Per Channel<br>(words)                                                                         | MIPS                                                                                                                                     |
|--------------------------------------------------------------------------------------------------------------------------|------------------------------|---------------------|---------------------------------------------|------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Library allocates<br/>memory</b> [includes<br><i>aecCreate()</i> and<br><i>aecDestroy()</i> ]                         | 1103                         | 0                   | 27                                          | 57 + 3N<br><br>N = Filter length<br>(or echo tail length<br>in taps)<br><br>Multiple channels<br>can exist | $(673 + 2N)F_s \times 10^{-6}$<br><br>N = Filter length (or echo tail<br>length in taps)<br>F <sub>s</sub> = Sampling Frequency in<br>Hz |
| <b>User<br/>application<br/>allocates memory</b><br>[does not include<br><i>aecCreate()</i> and<br><i>aecDestroy()</i> ] | 880                          | 0                   | 27                                          | 57 + 3N<br><br>N = Filter length<br>(or echo tail length<br>in taps)<br><br>Multiple channels<br>can exist | $(673 + 2N)F_s \times 10^{-6}$<br><br>N = Filter length (or echo tail<br>length in taps)<br>F <sub>s</sub> = Sampling Frequency in<br>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 DSP56858 for Small Memory Model.



**Table 7-24. Automatic Gain Control Memory and MIPS Requirements for Internal Memory on the DSP56858 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 |
|---------------------------------------------------------------------------------------------------------|------------------------|------------------|------------------------------------|------------------------------|------------------------------|
| <b>Library allocates memory</b> [includes <i>agcCreate()</i> and <i>agcDestroy()</i> ]                  | 117                    | 0                | 13                                 | 4                            | 0.34                         |
| <b>User application allocates memory</b> [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 DSP56858 for Large Memory Model.

**Table 7-25. Automatic Gain Control Memory and MIPS Requirements for Internal Memory on the DSP56858 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 |
|---------------------------------------------------------------------------------------------------------|------------------------|------------------|------------------------------------|------------------------------|------------------------------|
| <b>Library allocates memory</b> [includes <i>agcCreate()</i> and <i>agcDestroy()</i> ]                  | 117                    | 0                | 13                                 | 4                            | 0.34                         |
| <b>User application allocates memory</b> [does not include <i>agcCreate()</i> and <i>agcDestroy()</i> ] | 87                     | 0                | 13                                 | 4                            | 0.34                         |

**Note:** MIPS 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 are calculated with the following formula. Sampling rate is assumed to be **8000Hz**.

$$\frac{8000[43 + N \times 42]}{N \times 10^6}$$

### 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 (DTMF) -D 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%<br>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 information for internal memory on the DSP5685x.

**Table 7-27. CAS Detect Memory and MIPS Requirements for Internal Memory on the DSP5685x**

|                                                                                                                     | Program Memory (words) | Data ROM (words)                            | Software Stack Per Channel (words) | Data RAM Per Channel (words) | MIPS at 8000Hz Sampling Rate |
|---------------------------------------------------------------------------------------------------------------------|------------------------|---------------------------------------------|------------------------------------|------------------------------|------------------------------|
| <b>Library allocates memory</b><br>[includes <i>casDetectCreate()</i> and <i>casDetectDestroy()</i> ]               | 1413                   | Internal/External = 54<br><br>Internal = 28 | 27                                 | 400                          | 1.328                        |
| <b>User application allocates memory</b> [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  $N \leq 80$ ), MIPS are 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<br>(+/- 0.7%) | Steady On                                                          | -29 to -10dBm |
| Message Waiting Indicator Tone     | 350+440<br>(+/- 0.7%) | 10 Bursts<br>(0.1Seconds On;<br>0.1 Seconds Off)<br>then steady On | -29 to -10dBm |
| Recall Dial Tone/Confirmation Tone | 350+440<br>(+/- 0.7%) | 3 Bursts<br>(0.1 Seconds On;<br>0.1 Seconds Off)<br>then steady On | -29 to -10dBm |
| Line Busy                          | 480+620<br>(+/- 1.5%) | 0.5 Seconds On,<br>0.5 Seconds Off;<br>repeating                   | -52 to -21dBm |
| Reorder                            | 480+620<br>(+/- 1.5%) | 0.25 Seconds On,<br>0.25 Seconds Off;<br>repeating                 | -52 to -21dBm |
| Audible Ringing                    | 440+480<br>(+/- 1.6%) | 2 Seconds On,<br>4 Seconds Off;<br>repeating                       | -47 to -16dBm |

### 7.4.4.1 Performance

**Table 7-29** details CPT memory and MIPS requirements for internal memory on the DSP5685x.

**Table 7-29. CPT Memory and MIPS Requirements for Internal Memory on the DSP5685x**

|                                                                                                                         | Program<br>Memory<br>(words) | Data ROM<br>(words) | Software<br>Stack Per<br>Channel<br>(words) | Data RAM<br>Per Channel<br>(words) | MIPS at<br>8000Hz<br>Sampling Rate |
|-------------------------------------------------------------------------------------------------------------------------|------------------------------|---------------------|---------------------------------------------|------------------------------------|------------------------------------|
| <b>Library allocates memory</b><br>[includes <i>CPTDetCreate()</i><br>and <i>CPTDetDestroy()</i> ]                      | 1214                         | 6                   | 23                                          | 142                                | 0.55                               |
| <b>User application allocates<br/>memory</b> [does not include<br><i>CPTDetCreate()</i> and<br><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 ≤ 160), MIPS are calculated using 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 DSP5685x.

**Table 7-30. Caller ID Memory and MIPS Requirements for Internal Memory on the DSP5685x**

|                                                                                                                             | Program<br>Memory<br>(words) | Data ROM<br>(words) | Software<br>Stack Per<br>Channel<br>(words) | Data RAM<br>Per Channel<br>(words) | MIPS at<br>8000Hz<br>Sampling Rate |
|-----------------------------------------------------------------------------------------------------------------------------|------------------------------|---------------------|---------------------------------------------|------------------------------------|------------------------------------|
| <b>Library allocates memory</b><br>[includes <i>CallerIDCreate()</i> and<br><i>callerIDDestroy()</i> ]                      | 1372                         | 28                  | 22                                          | 393                                | 1.181                              |
| <b>User application allocates<br/>memory</b> [does not include<br><i>CallerIDCreate()</i> and<br><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**.

$$\frac{8000[2934 + N \times 19]}{N \times 10^6}$$

This library is described in more detail in the document **Caller ID Library**.

## 7.4.6 Common Tone Generation (CTG) Library

The CTG library can generate any tone with any number of simultaneous frequencies. These frequencies can have different starting and ending times and can be repeated for varying numbers of cycles.

### 7.4.6.1 Performance

[Table 7-31](#) details basic performance information for CTG on the DSP5685x for Small Memory Model.

**Table 7-31. Common Tone Generation Memory and MIPS Requirements for Internal Memory on the DSP5685x 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 |
|---------------------------------------------------------------------------------------------------------|------------------------|------------------|------------------------------------|-----------------------------------|------------------------------|
| <b>Library allocates memory</b> [includes <i>ctgCreate()</i> and <i>ctgDestroy()</i> ]                  | 768                    | 0                | 12                                 | 20 + 16 * (number of frequencies) | 2.95                         |
| <b>User application allocates memory</b> [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 DSP5685x for Large Memory Model.

**Table 7-32. Common Tone Generation Memory and MIPS Requirements for Internal Memory on the DSP5685x 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 |
|---------------------------------------------------------------------------------------------------------|------------------------|------------------|------------------------------------|-----------------------------------|------------------------------|
| <b>Library allocates memory</b> [includes <i>ctgCreate()</i> and <i>ctgDestroy()</i> ]                  | 768                    | 0                | 12                                 | 22 + 20 * (number of frequencies) | 2.95                         |
| <b>User application allocates memory</b> [does not include <i>ctgCreate()</i> and <i>ctgDestroy()</i> ] | 568                    | 0                | 12                                 | 22 + 20 * (number of frequencies) | 2.95                         |

**Note:**

- **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. 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 DSP5685x are shown in [Table 7-33](#).

**Table 7-33. DTMF Detection Memory and MIPS for Internal Memory on the DSP5685x**

|                                                                                                         | Program Memory (words) | Data ROM (words) | Software Stack Per Channel (words) | Data RAM Per Channel (words) | MIPS at 8000Hz Sampling Rate |
|---------------------------------------------------------------------------------------------------------|------------------------|------------------|------------------------------------|------------------------------|------------------------------|
| <b>Library allocates memory</b> [includes <i>DTMFDetCreate()</i> and <i>DTMFDetDestroy()</i> ]          | 1048                   | 48               | 29                                 | 186                          | 0.659                        |
| <b>User application allocates memory</b> [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](#) is 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 DSP5685x for Small Memory Model.



**Table 7-34. MIPS and Memory Requirements for Internal Memory on the DSP5685x 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 |
|----------------------------------------------------------------------------------------------------------|------------------------|------------------|------------------------------------|------------------------------|------------------------------|
| <b>Library allocates memory</b><br>[includes <i>dtmfCreate()</i> and <i>dtmfDestroy()</i> ]              | 273                    | 0                | 10                                 | 23                           | 0.1456                       |
| <b>User application allocates memory</b> [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 DSP5685x for Large Memory Model.

**Table 7-35. MIPS and Memory Requirements for Internal Memory on the DSP5685x 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 |
|----------------------------------------------------------------------------------------------------------|------------------------|------------------|------------------------------------|------------------------------|------------------------------|
| <b>Library allocates memory</b><br>[includes <i>dtmfCreate()</i> and <i>dtmfDestroy()</i> ]              | 273                    | 0                | 10                                 | 23                           | 0.1456                       |
| <b>User application allocates memory</b> [doesn't include <i>dtmfCreate()</i> and <i>dtmfDestroy()</i> ] | 244                    | 0                | 10                                 | 23                           | 0.1456                       |

- MIPS in [Table 7-34](#) and [Table 7-35](#) are for an input buffer length of **40** samples.
- For an input buffer length of “N” (N samples in the input buffer, where  $N \leq 40$ ), 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 further 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 DSP5685x.

**Table 7-36. G.165 Memory and MIPS Requirements for Internal Memory on the DSP5685x**

|                                                                                                         | Program Memory (words) | Data ROM (words) | Software Stack Per Channel (words) | Data RAM Per Channel (words)         | MIPS at 8000Hz Sampling Rate (echo tail (msec))                       |
|---------------------------------------------------------------------------------------------------------|------------------------|------------------|------------------------------------|--------------------------------------|-----------------------------------------------------------------------|
| <b>Library allocates memory</b><br>[includes <i>g165Create()</i> and <i>g165Destroy()</i> ]             | 2211                   | 234              | 20                                 | 282 + 3 * Echo-Span (number of taps) | 5.2429 (8)<br>6.0077(16)<br>7.2685 (24)<br>8.5613 (32)<br>9.8541 (40) |
| <b>User application allocates memory</b> [does not include <i>g165Create</i> and <i>g165Destroy()</i> ] | 1716                   | 234              | 20                                 | 282 + 3 * Echo-Span (number of taps) | 5.2429 (8)<br>6.0077(16)<br>7.2685 (24)<br>8.5613 (32)<br>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 further 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 DSP5685x.

**Table 7-37. G.168 Memory and MIPS Requirements for Internal Memory on the DSP5685x**

|                                                                                                           | Program Memory (words) | Data ROM (words) | Software Stack Per Channel (words) | Data RAM Per Channel (words)         | MIPS at 8000Hz Sampling Rate (echo tail (msec))                                                                       |
|-----------------------------------------------------------------------------------------------------------|------------------------|------------------|------------------------------------|--------------------------------------|-----------------------------------------------------------------------------------------------------------------------|
| <b>Library allocates memory</b> [includes <i>g168Create()</i> and <i>g168Destroy()</i> ]                  | 2211                   | 234              | 20                                 | 282 + 3 * Echo-Span (number of taps) | 5.2429 (8)<br>6.0077(16)<br>7.2685 (24)<br>8.5613 (32)<br>9.8541 (40)<br>11.1469 (48)<br>12.4397 (56)<br>13.7325 (64) |
| <b>User application allocates memory</b> [does not include <i>g168Create()</i> and <i>g168Destroy()</i> ] | 1716                   | 234              | 20                                 | 282 + 3 * Echo-Span (number of taps) | 5.2429 (8)<br>6.0077(16)<br>7.2685 (24)<br>8.5613 (32)<br>9.8541 (40)<br>11.1469 (48)<br>12.4397 (56)<br>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  $\mu$ 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 DSP5685x for Small Memory Model.

**Table 7-38. G.711 Memory and MIPS Requirements for Internal Memory on the DSP5685x 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 DSP5685x for Large Memory Model.

**Table 7-39. G.711 Memory and MIPS Requirements for Internal Memory on the DSP5685x 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 further detail in the document **G.711 Library**.

## 7.4.12 G.723.1A Speech Codec Library

The ITU-T Recommendation G.723.1 Dual Rate Speech Coder provides a coded representation used for compressing speech in either of two very low bit rates. The higher 6.3 Kbits/s rate produces a greater quality reproduction, while the lower 5.3 Kbits/s rate provides system designers additional flexibility. Both rates are mandatory, and it is possible to switch between the rates on any 30ms frame boundary. Annex A of this Recommendation (G.723.1A) provides an optional silence compression system used to reduce the transmitted bit rate during silent intervals of speech.

This coder was optimized to represent speech with a high quality of the two rates using a limited amount of complexity. Music and other audio signals are not represented as faithfully as speech, but can be compressed and decompressed using this coder.

### 7.4.12.1 Performance

Memory requirements for the G.723.1A Speech Codec library for internal memory on the DSP5685x are shown in [Table 7-40](#).

**Table 7-40. G.723.1A Memory and MIPS Requirements for Internal Memory on the DSP5685x**

| Program Memory (words) | Data ROM (words) | Software Stack Overlay Per Channel (words) | Data RAM Per Channel (words) | MIPS at 8000Hz Sampling Rate  |
|------------------------|------------------|--------------------------------------------|------------------------------|-------------------------------|
| 7674                   | 9466             | 1366                                       | 940                          | 18.5 (typical)<br>23.6 (peak) |

A test that verifies the operation of the G.723.1A library is described in [Chapter 8](#).

This library is described in more detail in the document **G.723.1A Library**.

## 7.4.13 G.726 Library

The ITU-T Recommendation G.726 Adaptive Differential Pulse Code Modulation Speech Codec converts a 64 Kbits/s A-Law or  $\mu$ Law pulse code modulation (PCM) channel to and from 40, 32, 24, 16 Kbits/s PCM stream using an ADPCM transcoding technique.

### 7.4.13.1 Performance

Memory requirements for the G.726 ADPCM Codec library for internal memory on the DSP5685x are shown in [Table 7-41](#).

**Table 7-41. G.726 Memory and MIPS Requirements for Internal Memory on the DSP5685x**

|                                                                                                                                                                                   |                              | Program<br>Memory<br>(words) | Data<br>ROM<br>(words) | Software<br>Stack Per<br>Channel<br>(words) | Data<br>RAM<br>Per<br>Channel<br>(words) | MIPS at<br>8000Hz Sampling<br>Rate                                                           |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------|------------------------------|------------------------|---------------------------------------------|------------------------------------------|----------------------------------------------------------------------------------------------|
| <b>Library allocates<br/>memory</b> [includes<br><i>G726EncCreate()</i><br><i>G726EncDestroy()</i><br><i>G726DecCreate()</i> and<br><i>G726DecDestroy()</i> ]                     | <b>Encoder</b>               | 1060                         | 220                    | 16                                          | 73                                       | 7.272 @40kbps<br>7.272@32kbps<br>7.208@24kbps<br>7.176@16kbps                                |
|                                                                                                                                                                                   | <b>Decoder</b>               | 1216                         | 220                    | 16                                          | 73                                       | 8.304 @40kbps<br>8.304@32kbps<br>8.24@24kbps<br>8.208@16kbps                                 |
|                                                                                                                                                                                   | <b>Encoder +<br/>Decoder</b> | <b>1311</b>                  | <b>220</b>             | <b>16</b>                                   | <b>146</b>                               | <b>15.576@40kbps</b><br><b>15.576@32kbps</b><br><b>15.448@24kbps</b><br><b>15.384@16kbps</b> |
| <b>User application<br/>allocates memory</b><br>[does not include<br><i>G726EncCreate()</i><br><i>G726EncDestroy()</i><br><i>G726DecCreate()</i> and<br><i>G726DecDestroy()</i> ] | <b>Encoder</b>               | 1110                         | 220                    | 16                                          | 73                                       | 7.272 @40kbps<br>7.272@32kbps<br>7.208@24kbps<br>7.176@16kbps                                |
|                                                                                                                                                                                   | <b>Decoder</b>               | 1141                         | 220                    | 16                                          | 73                                       | 8.304 @40kbps<br>8.304@32kbps<br>8.24@24kbps<br>8.208@16kbps                                 |
|                                                                                                                                                                                   | <b>Encoder +<br/>Decoder</b> | <b>1205</b>                  | <b>220</b>             | <b>16</b>                                   | <b>146</b>                               | <b>15.576@40kbps</b><br><b>15.576@32kbps</b><br><b>15.448@24kbps</b><br><b>15.384@16kbps</b> |

**Note:** Data RAM 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 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 further detail in the document **G.726 Library**.

### 7.4.14 G.729AB Vocoder Library

The ITU-T G.729 Recommendation describes an algorithm for speech coding at 8kbps using the Conjugate-Structure Algebraic-Code-Excited Linear-Prediction model. The coder operates on speech frames of 80 samples at a sampling rate of 8000 samples per second (each frame corresponds at 10ms of speech).

The ITU-T G.729 Annex A is a less-complicated version of the G.729 vocoder. The block diagram of the G.729 Annex A vocoder is similar to that of the original G.729 Recommendation. Most blocks are identical; only a few blocks have different implementations.

The ITU-T G.729 Annex B is used to reduce the transmission rate during silent periods of speech. Each frame first passes a voice activity detection (VAD) block that decides if the frame is “speech” or “silent”. This decision is transmitted to the encoder and the decoder via the bit-stream, activating either the “Active Voice” block or “Non-Active Voice” block in both the encoder and decoder.

### 7.4.14.1 Performance

Processing load<sup>1</sup> and memory requirements for the G.729AB Vocoder library are shown in [Table 7-42](#).

**Table 7-42. Performance Figures for the G.729AB Vocoder Library**

|                | Code Size<br>(words) | Data Size           |                            |                  |                       | Processing<br>Load<br>(MCPS) |
|----------------|----------------------|---------------------|----------------------------|------------------|-----------------------|------------------------------|
|                |                      | Tables<br>(words)   | Channel<br>Data<br>(words) | Stack<br>(words) | Total Data<br>(words) |                              |
| <b>Encoder</b> | 6,960                | shared <sup>1</sup> | 708                        | 1,108            | 4,481                 | 10.53                        |
| <b>Decoder</b> | 2,920                | shared <sup>1</sup> | 430                        | 690              | 3,785                 | 2.44                         |
| <b>Vocoder</b> | 8,516                | 2,665               | 1,138                      | 1,108            | 4,911                 | 12.97                        |

1. The tables are shared by encoder and decoder

### 7.4.15 MFCR2 Detection Library

The MFCR2 signal consists of two frequencies, taken either from 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.

#### 7.4.15.1 Performance

The MFCR2 Detection memory and MIPS requirements for internal memory on the DSP5685x are shown in [Table 7-43](#).

1. The processing load is defined as the execution time (in cycles) per frame (usually worst case is used) multiplied by the number of frames per second (100 in the G.729 vocoders suite), and divided by 1,000,000. The measurement unit used is MCPS (million cycles per second).



**Table 7-43. MFCR2 Detection Memory and MIPS for Internal Memory on the DSP5685x**

|                                                                                                                         | Program Memory (words) | Data ROM (words) | Software Stack Per Channel (words) | Data RAM Per Channel (words) | MIPS at 8000Hz Sampling Rate |
|-------------------------------------------------------------------------------------------------------------------------|------------------------|------------------|------------------------------------|------------------------------|------------------------------|
| <b>Library allocates memory</b><br>[includes <i>MFCR2DetectCreate()</i> and <i>MFCR2DetectDestroy()</i> ]               | 986                    | 250              | 5                                  | 216                          | 4.1                          |
| <b>User application allocates memory</b> [does not include <i>MFCR2DetectCreate()</i> and <i>MFCR2DetectDestroy()</i> ] | 816                    | 250              | 5                                  | 216                          | 4.1                          |

**Notes:**

- Data memory figures in [Table 7-43](#) 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 7-43](#) 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**.

## 7.4.16 Noise Suppression (NS) Library

[Table 7-44](#) details basic performance of noise suppression on the DSP5685x.

**Table 7-44. NS Memory and MIPS Requirements for Internal Memory on the DSP5685x**

|                                                                                                       | Program Memory (words) | Data ROM (words) | Software Stack Per Channel (words) | Data RAM Per Channel (words) | MIPS at 8000Hz Sampling Rate |
|-------------------------------------------------------------------------------------------------------|------------------------|------------------|------------------------------------|------------------------------|------------------------------|
| <b>Library allocates memory</b><br>[includes <i>nsCreate()</i> and <i>nsDestroy()</i> ]               | 1724                   | 357              | 40                                 | 653                          | 3.063                        |
| <b>User application allocates memory</b> [does not include <i>nsCreate()</i> and <i>nsDestroy()</i> ] | 1294                   | 357              | 40                                 | 653                          | 3.063                        |

**Notes:**

- Data RAM figures in [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 Motorola's DSP architecture, which has strict memory alignment requirements.

- MIPS in [Table 7-44](#) are for **80 input samples per call**

This library is described in further detail in the document **Noise Suppression Library**.

## 7.4.17 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.

### 7.4.17.1 Performance

[Table 7-45](#) details the basic performance of VAD for internal memory on the DSP5685x for Small Memory Model.

**Table 7-45. VAD Memory and MIPS Requirements for Internal Memory on the DSP5685x for Small Memory Model**

|                                                                                                                | Program<br>Memory<br>(words) | Data<br>ROM<br>(words) | Software<br>Stack Per<br>Channel<br>(words) | Data RAM<br>Per Channel<br>(words) | MIPS at<br>8000Hz<br>Sampling Rate |
|----------------------------------------------------------------------------------------------------------------|------------------------------|------------------------|---------------------------------------------|------------------------------------|------------------------------------|
| <b>Library allocates memory</b><br>[includes <i>vadCreate()</i> and<br><i>vadDestroy()</i> ]                   | 793                          | 0                      | 38                                          | 84                                 | 1.645                              |
| <b>User application allocates<br/>memory</b> (does not include<br><i>vadCreate()</i> and <i>vadDestroy()</i> ) | 708                          | 0                      | 38                                          | 84                                 | 1.645                              |

[Table 7-46](#) details the basic performance of VAD for internal memory on the DSP5685x for Large Memory Model.



**Table 7-46. VAD Memory and MIPS Requirements for Internal Memory on the DSP5685x 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 |
|---------------------------------------------------------------------------------------------------------|------------------------|------------------|------------------------------------|------------------------------|------------------------------|
| <b>Library allocates memory</b> [includes <i>vadCreate()</i> and <i>vadDestroy()</i> ]                  | 793                    | 0                | 38                                 | 88                           | 1.645                        |
| <b>User application allocates memory</b> (does not include <i>vadCreate()</i> and <i>vadDestroy()</i> ) | 708                    | 0                | 38                                 | 88                           | 1.645                        |

**Notes:**

- **Table 7-45** and **Table 7-46** 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 7-45** and **Table 7-46** 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 further 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 7-47** shows VRLite-1 Memory and MIPS figures for internal memory on the DSP5685x.

**Table 7-47. VRLite-1 Memory and MIPS Requirements for Internal Memory on the DSP5685x**

|                                                                                                                              | Program<br>Memory<br>(words) | Data ROM<br>(words) | Software<br>Stack Per<br>Channel<br>(words) | Data RAM<br>Per Channel<br>(words)           | MIPS                                            |
|------------------------------------------------------------------------------------------------------------------------------|------------------------------|---------------------|---------------------------------------------|----------------------------------------------|-------------------------------------------------|
| <b>Library allocates<br/>memory</b> [includes<br><i>vrlite1Create()</i> and<br><i>vrlite1Destroy()</i> ]                     | 6958                         | 540                 | 51                                          | 4670<br>(only one<br>channel is<br>possible) | 13.5<br>(front end only, which<br>is real-time) |
| <b>User application<br/>allocates memory</b><br>[does not include<br><i>vrlite1Create()</i> and<br><i>vrlite1Destroy()</i> ] | 6859                         | 540                 | 51                                          | 4670<br>(only one<br>channel is<br>possible) | 13.5<br>(front end only, which<br>is real-time) |

**Table 7-48** shows the Instruction Cycle Count figures for internal memory on the DSP5685x (which equals Oscillator Cycle Count) for Training, Rejection Analysis, and Recognition, non-real time modules.

**Table 7-48. Instruction Cycle Counts for Internal Memory on the DSP5685x**

| Module                                                       | Instruction Cycle Count |
|--------------------------------------------------------------|-------------------------|
| Training <b>of a word</b>                                    | 1000336                 |
| Rejection Analysis <b>per trained word</b> in the vocabulary | 318114                  |
| Recognition <b>per trained word</b> in the vocabulary        | 442708                  |

**Notes:**

- Memory resources in **Table 7-48** 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 7-48** 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 further 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 7-49** 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 7-49**. 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 7-49**.

**Table 7-49. Generic Echo Cancellor Memory and MIPS Requirements**

|                               | Program<br>Memory<br>(ROM)<br>(16 bit<br>words) | Data<br>Memory<br>(ROM)<br>(16 bit<br>words) | Data RAM<br>(16 bit<br>words) | Data RAM<br>Per Instance<br>(16 bit<br>words) | MIPS  |
|-------------------------------|-------------------------------------------------|----------------------------------------------|-------------------------------|-----------------------------------------------|-------|
| GEC Library (Internal Memory) | 680                                             | NA                                           | NA                            | 580 + buffer <sup>1</sup>                     | 10.88 |
| GEC Library (External Memory) | 680                                             | NA                                           | NA                            | 580 + buffer <sup>1</sup>                     | 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 7-50](#) details the memory and MIPS requirements for the Type 1 and 2 Telephony Parser Library.

**Table 7-50. 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 7-51](#) 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 7-51](#). 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** man-

ual should also be added to the memory requirement stated in [Table 7-51](#). The MIPS shown in [Table 7-51](#) is for a tail length of 24ms for the acoustic echo canceller.l

**Table 7-51. 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 <sup>1</sup>            | 12.96 |
| Full Duplex Speakerphone Library (External Memory) | 2203                                | NA                               | NA                      | 603 + Buffer <sup>1</sup>            | 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 7-52](#) details the memory and MIPS requirements for the Type 1 Telephony Features Library.

**Table 7-52. 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).



### 7.6.6 Features and Performance

**Table 7-53** details the memory and MIPS requirements for the Type 1 and 2 Telephony Features Library.

**Table 7-53. Type 1 and 2 Telephony Features Memory and MIPS Requirements**

|                                            | Program<br>Memory<br>(ROM)<br>(16 bit<br>words) | Data ROM<br>(16 bit<br>words) | Data RAM<br>1st Instance<br>(16 bit<br>words) | Data RAM<br>Additional<br>Instance<br>(16 bit<br>words) | MIPS<br>(When all program<br>and data is internal<br>memory) |
|--------------------------------------------|-------------------------------------------------|-------------------------------|-----------------------------------------------|---------------------------------------------------------|--------------------------------------------------------------|
| Type 1 and 2 Telephony<br>Features Library | 2.8 K                                           | 20                            | 257                                           | 192                                                     | 9.6                                                          |



## Chapter 8

# Library Tests

The SDK for the DSP5685x 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\dsp56858evm\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 **DSP56858 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\dsp56858evm\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 **DSP56858 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\dsp56858evm\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 **DSP56858 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 **DSP56858 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 **DSP56858 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 **DSP56858 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 **DSP56858 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 **DSP56858 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 **DSP56858 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 **DSP56858 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 56000 bps
- 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, namely, 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 **DSP56858 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 56000 bps
- 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  $\pm$ 12 bps

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 **DSP56858 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 56000 bps 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 **DSP56858 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 56000 bps 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 **DSP56858 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 56000 bps
- 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 CTG test is intended to confirm the accuracy of the CTG algorithm.

In this test for Common Tone Generation, 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\ctg\test\ref_outputs
```

### 8.5.6.1 Set-up for Common Tone Generation Test

#### EVM Jumper Settings:

Use default settings as shown in the **DSP56858 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 **DSP56858 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 56000 bps
- 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 DSP56858 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 the 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 **DSP56858 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 56000 bps
- 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  $2100 \pm 21$ 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:

...\\dsp56858evm\\nos\\telephony\\g165\\.

#### 8.5.9.1 Set-up for G.165 Test

##### EVM Jumper Settings:

Use default settings as shown in the **DSP56858 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 56000 bps 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  $2100 \pm 21\text{Hz}$ , 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:

...\dsp56858evm\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 **DSP56858 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 56000 bps
- 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:

`...dsp56858evm\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 **DSP56858 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 56000 bps
- 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.723.1A Test

This test is intended to confirm the accuracy of G.723.1A algorithms provided by the SDK.

Input to the application is passed through a file, *tstboth.bin*. This input file for running this test is read from the following directory:

`...\nos\telephony\g723\g723_test\ref_data\`

The testing is performed with this configuration :

- `CoderRate = 63 /* The algorithm runs for 6.3 kbps*/`
- `BitErrors = 0`

The test runs in loopback so that once the encoder has run, the encoded data is passed to the decoder. The decoder output is compared back to *tstboth.bin*.

This test is run for 860 frames.



### 8.5.12.1 Set-up for G.723.1A Test

#### EVM Jumper Settings:

Use default settings as shown in the **DSP56858 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.723.1A 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\g723\g723\_test\test\_g723.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

### 8.5.13 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 16 kbps.

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, 16 kbps data rates. The standard output test vectors are available in the folder **...\nos\telephony\g726\test\test\_dec\** in *files.out*, where *files* refers 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.

### 8.5.13.1 Set-up for G.726 Test

#### EVM Jumper Settings:

Use default settings as shown in the **DSP56858 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 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 56000 bps
- 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

### 8.5.14 G.729AB Test

Three test applications are available to verify the bit-exact implementation of the G.729AB algorithm when running on hardware. The applications allow testing in single- or multiple-channel implementations.

This chapter describes both the set-up required to run the tests and how to run them.

To be able to run test applications, an evaluation module (EVM) and a serial extension cable<sup>1</sup> are needed. The tester is composed of two applications: one runs on the EVM and the other runs on a PC.

#### 8.5.14.1 Set-up for G.729AB Test

##### EVM Jumper Settings:

Use default settings as shown in the **DSP56858 Evaluation Module User's Manual**.

##### PC Setup:

This test requires PC applications for reading/writing data to the serial port. The applications that run on the PC can be found in:

`.../x86/win32/applications/g729ab_file_server2`

1. The serial extension cable does not twist the Tx/D, Rx/D signals at its terminals.

2. The `.../x86/win32/applications/g729ab_file_server` will be referred in the remainder of the document as {g729ab\_FS}

It reads input test vectors from the PC; sends them, frame-by-frame to the EVM; receives processed frames from the EVM and stores them on the PC.

The test cases from subfolders of *{g729ab\_FS}/test\_cases/reference* are those provided by ITU with the C reference code for Recommendation G.729, Annex AB.

Connect the serial cable between the EVM and the COM1 or COM2 port of the computer.

### 8.5.14.2 Procedure for G.729AB Encoder Test

- Launch *{g729ab\_FS}/encoder/encoder\_xxxx.exe*, where “xxxx” is COM1 or COM2, corresponding to the port where the serial cable is connected
- Using CodeWarrior, open:  
*.../nos/telephony/g729ab/test/test\_g729ab.mcp*
- Select target *encoder* from *test\_g729ab.mcp*
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- To pass the bit-exact test, the *.bit* files in the *{g729ab\_FS}/test\_cases/result* folder must be identical to the files from the *{g729ab\_FS}/test\_cases/reference/ref\_bit* folder
- Test cases can be chosen via the *{g729ab\_FS}/encoder/g729ab\_encoder.ini* file; requirements for writing this *.ini* file are:
  - The first line contains the root directory of the reference vectors ('reference')
  - The second line contains the directory where the results will be placed
  - Each of the next lines contains the name of the input vector (e.g. speech or pitch), the number of frames from that input vector, and, only for encoder, the VAD switch (can be 0 or 1)
  - Every line that begins with “#” is considered comment
  - Every blank line is avoided
  - The last line must start with “@”

### 8.5.14.3 Procedure for G.729AB Decoder Test

- Launch *{g729ab\_FS}/decoder/decoder\_xxxx.exe*, where “xxxx” is COM1 or COM2, corresponding to the port where the serial cable is connected
- Using CodeWarrior, open:  
*.../nos/telephony/g729ab/test/test\_g729ab.mcp*
- Select target *decoder* from *test\_g729ab.mcp*
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- To pass the bit-exact test, the *.out* files in the *{g729ab\_FS}/test\_cases/result* folder must be identical to the files from the *{g729ab\_FS}/test\_cases/reference/ref\_out* folder
- Test cases can be chosen via the *{g729ab\_FS}/decoder/g729ab\_decoder.ini* file; requirements to write this *.ini* file are identical to those for Encoder testing

#### 8.5.14.4 Procedure for G.729AB Multi-Channel Test

- Launch `{g729ab_FS}/multi_channel/multi_channel_xxxx.exe`, where “xxxx” is COM1 or COM2, corresponding to the port where the serial cable is connected.
- Using CodeWarrior, open:  
`.../nos/telephony/g729ab/test/test_g729ab.mcp`
- Select target `vocoder_multi_channel` from `test_g729ab.mcp`
- Build and download the project; when the debugger reaches `main()`, it will stop
- Select *Run* in the debugger to continue executing the test
- To pass the bit-exact test, the `.bit` files resulted in `{g729ab_FS}/test_cases/result` folder must be identical to the files from the `{g729ab_FS}/test_cases/reference/ref_bit` folder, and the `.out` files in the `{g729ab_FS}/test_cases/result` folder must be identical to the files from the `{g729ab_FS}/test_cases/reference/ref_out`
- Test cases can be chosen via the `{g729ab_FS}/multi_channel/g729ab_vocoder_multichannel.ini` file; the requirements for writing this `.ini` file are:
  - The first line of this file contains the number of channels; if the number exceeds MAX\_CHANNELS as defined by `vocoder_multi_channel.c`, it is automatically overwritten with MAX\_CHANNELS
  - The second line contains the root directory of the reference vectors (“reference”)
  - The third line contains the directory where the results will be placed
  - Each of the next lines contains the name of the input vector (e.g., speech or pitch), the number of frames from that input vector, and the VAD switch (which can be 0 or 1)
  - Every line that begins with “#” is considered comment
  - Every blank line is avoided
  - The last line must start with “@”

#### 8.5.15 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`

##### 8.5.15.1 Set-up for MFCR2 Detection Test

###### EVM Jumper Settings:

Use the default settings shown in the **DSP56858 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 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 56000bps
- 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.

**8.5.16 Noise Suppression (NS) Test**

The Noise Suppression test is intended to demonstrate the accuracy of the algorithm.

The input to noise suppression 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

A file, *ns.out*, will be generated by the test and stored in the same directory; its contents 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 *ns.out*.

**8.5.16.1 Set-up for Noise Suppression Test****EVM Jumper Settings:**

Use the default settings shown in the **DSP56858 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.

**8.5.16.2 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*.

## 8.5.17 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:

...\dsp56858evm\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:

...\dsp56858evm\nos\telephony\vad\test\_vad\outputs\

### 8.5.17.1 Set-up for VAD Test

#### EVM Jumper Settings:

Use the default settings shown in the **DSP56858 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.

### 8.5.17.2 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 56000 bps
- 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 **DSP56858 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.

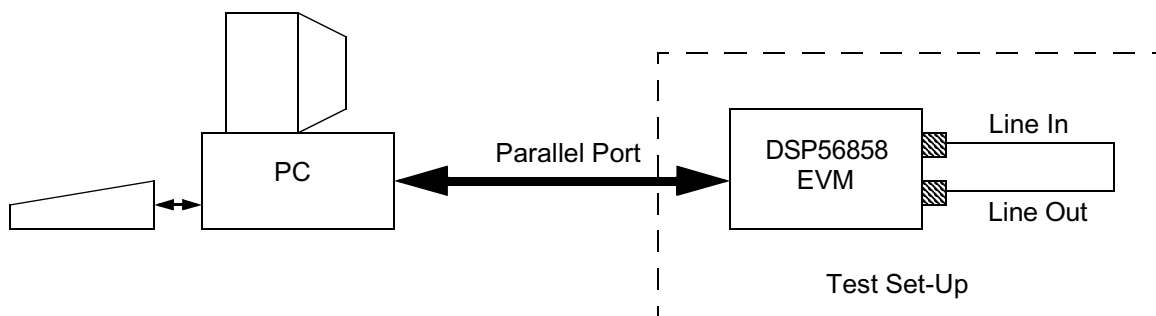


Figure 8-1. Test Set-up for V.21

#### 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 DSP56858EVM should be closed (it is open by default). All other jumpers use the default settings shown in the **DSP56858 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](#).



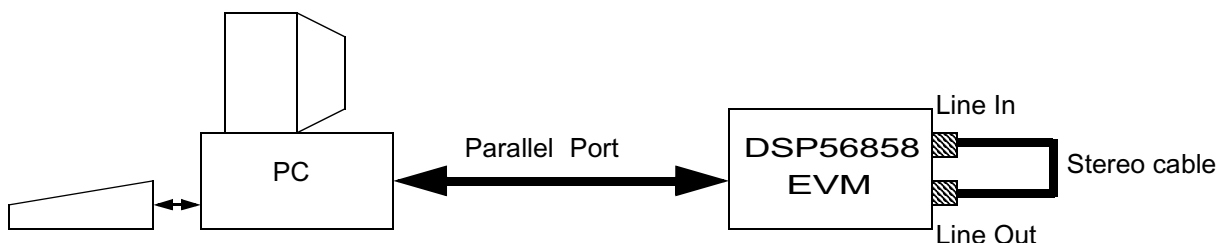


Figure 8-2. V.22bis Set-up for Digital and Analog Loopback Tests

### 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 DSP56858EVM should be closed (it is open by default). All other jumpers use the default settings shown in the **DSP56858 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**.

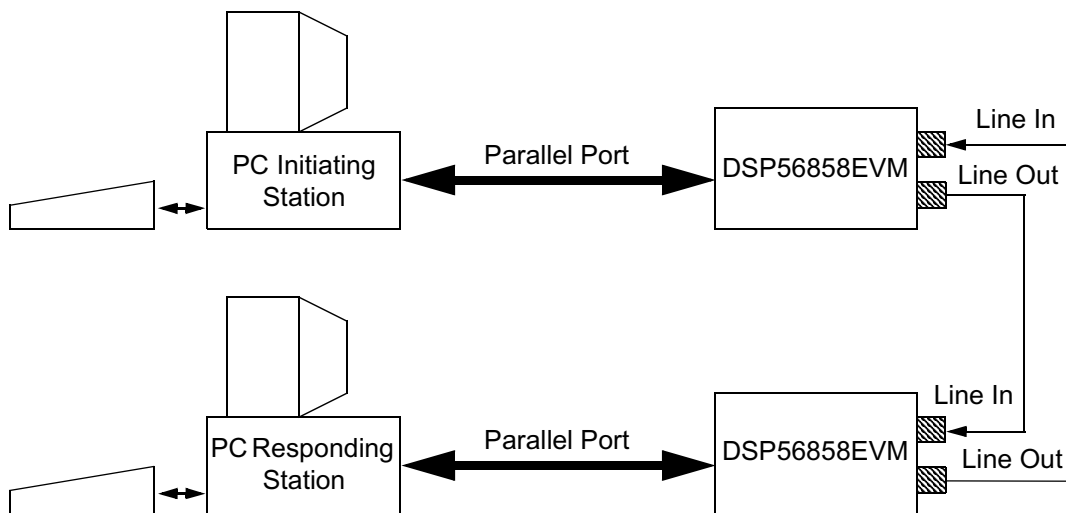


Figure 8-3. V.8bis Test Set-up

### 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 DSP56858EVM board. Use the ACS90 Computer Speaker System or its equivalent for best results.

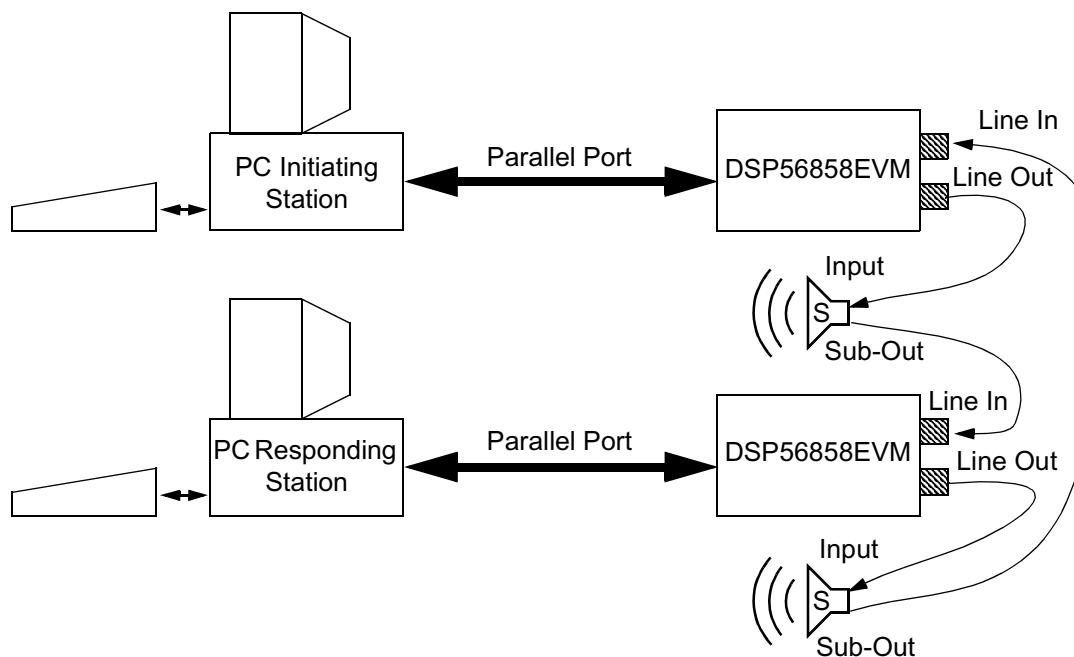


Figure 8-4. V.8bis Test Set-up with Speakers

## 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\vr lite1\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\vr lite1\test\io\`
  - The global noise mean values are directly entered in the GlobalStats[] buffer of the *pConfig* structure, in the test file *test\_vr lite1\_training.c*:  
`...\nos\speech\vr lite1\test\c_sources`
- For **Recognition**, one utterance from a speaker is stored in the file *utt\_recog.in*:  
`...\nos\speech\vr lite1\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\vr lite1\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 **DSP56858 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 56000 bps
- 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 Celler Library

Two example applications are discussed in this section; a speakerphone application and a test application.

#### 8.8.1.1 Generic Echo Celler Library 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. *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 Cancellor 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 Library 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 precaptured 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 Generic Echo Canceller 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 Chapter 3 of the **Full Duplex Speakerphone Library** manual.

#### 8.8.3.1 Full Duplex Speakerphone Application

The Full Duplex Speakerphone application has been provided for use on a DSP56858EVM 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 Chapter 3 of the **Full Duplex Speakerphone Library** manual.

#### 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 given 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. 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.aecdtHang = -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 the **Full Duplex Speakerphone Library** manual's Chapter 3.

## 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*.

#### 8.8.4.1.1 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 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;
 }
#endif

 #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

# DSP5685x Applications

The SDK for the DSP5685x 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 DSP56858EVM board. For details on building and executing this application, see [Section 6.2.7](#).

#### 9.1.2 Codec DMA Demo

The Codec DMA Demo demonstrates the use of the Codec DMA driver provided by the SDK. It uses the on-chip DMA to exercise the codec hardware on the DSP56858EVM board. For details on building and executing this application, see [Section 6.3.7](#).

#### 9.1.3 COP Demo

The COP Demo demonstrates the use of the COP driver provided by the SDK by exercising the COP on the DSP5685x. For details on building and executing this application, see [Section 5.1.4](#).

#### 9.1.4 Host Interface Demo

The Host Interface Demo demonstrates the use of the Host Interface driver provided by the SDK by exercising the Host Interface on the DSP56858EVM board. For details on building and executing this application, see [Section 5.6.7](#).



## 9.1.5 PC Master Demo

The PC Master Demo demonstrates the use of the PC Master driver provided by the SDK. For details on building and executing this application, see [Section 6.8.6](#).

## 9.1.6 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 DSP5685x to generate timed interrupts that flash LEDs; for details on how to build and execute the quad timer application, see [Section 5.8.7](#).

## 9.1.7 SCI Demo

The SCI demo demonstrates the use of the SCI Driver provided by the SDK by exercising the SCI on the DSP5685x to read and write data to a PC application using a serial port. See [Section 5.9.6.1](#) for more information on the SCI application.

## 9.1.8 SCI Echo Demo

The SCI Echo demo demonstrates the use of the SCI Driver provided by the SDK by exercising the SCI on the DSP5685x to echo data to the HyperTerminal using a serial port. See [Section 5.9.7.2](#) for more information on the SCI application.

## 9.1.9 TDC1 Demo

The TDC1 Demo demonstrates the use of the TDC1 driver provided by the SDK by exercising the TDC1 DAA and Codec hardware on the TDC1 board. For details on building and executing this application, see [Section 6.10.7](#).

## 9.1.10 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 5.11.3](#).

For more information about the Timer library, see **Timer Services** in the **SDK Programmer's Guide**.

## 9.1.11 TOD Demo

The TOD Demo demonstrates the use of the TOD driver provided by the SDK by exercising the TOD hardware on the DSP56858EVM board. For details on building and executing this application, see [Section 5.12.7](#).

## 9.2 Security Demos

### 9.2.1 DES Demo

The DES application is demonstrated with two demos: a File I/O demo and a realtime 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.

This 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.

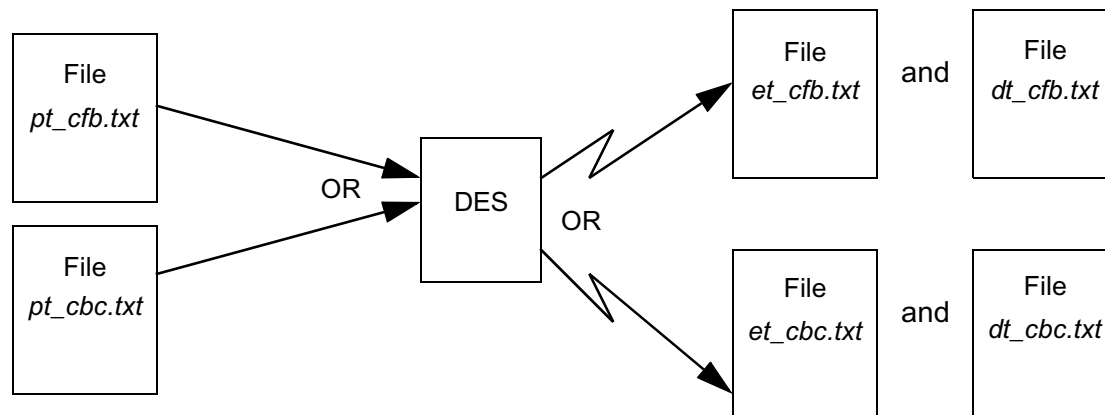


Figure 9-1. DES Demo

#### 9.2.1.1.1 Set-up for DES File I/O Demo

##### EVM Jumper Settings:

Use the default settings shown in the **DSP56858 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\\dsp56858evm\\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\dsp56858evm\nos\applications\security\des\io\`

### 9.2.1.2 DES Realtime Demo

The DES Realtime Demo illustrates 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 **DSP56858 Evaluation Module User's Manual**.

Toggle all three switches on the "Sample Rate" dip-switch (S5) to "Off", which will generate an 8000Hz sample rate.

The demo set-up is shown in [Figure 9-2](#).

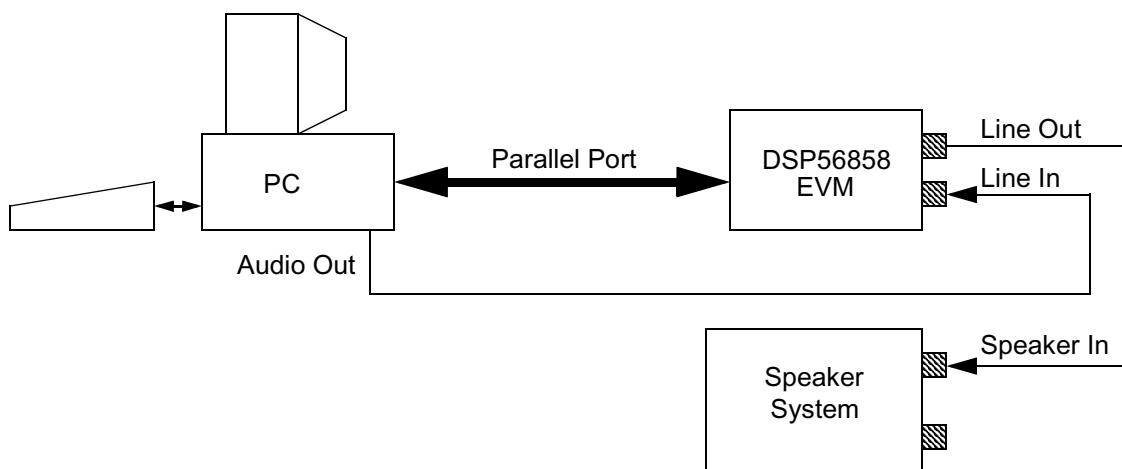


**Figure 9-2. DES Demo Set-up**

- 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 Realtime 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's Audio Out port to the EVM's Line In
- When the demo starts, the 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 3DES application is demonstrated in two demos: a File I/O demo and a realtime demo.

**9.2.2.1 Triple DES File I/O Demo**

This application uses the 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.

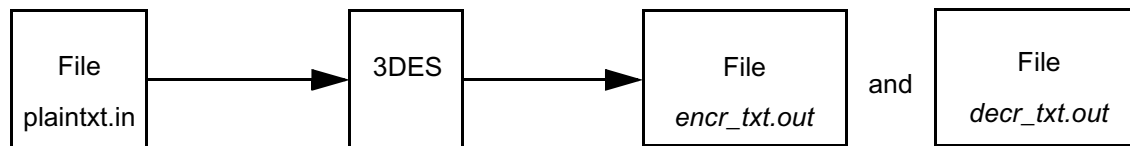


Figure 9-3. Triple DES Demo

### 9.2.2.1.1 Set-up for Triple DES Demo

#### EVM Jumper Settings:

Use the default settings shown in the **DSP56858 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 *plaintxt.in* in:  
`...\src\dsp56858evm\nos\applications\security\3des\io\`
- This file may be altered if desired
- If the user wants to change the input file *plaintxt.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\dsp56858evm\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\dsp56858evm\nos\applications\security\3des\io\`

### 9.2.2.2 Triple DES Realtime Demo

The 3DES Realtime Demo illustrates how to use the use of 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.

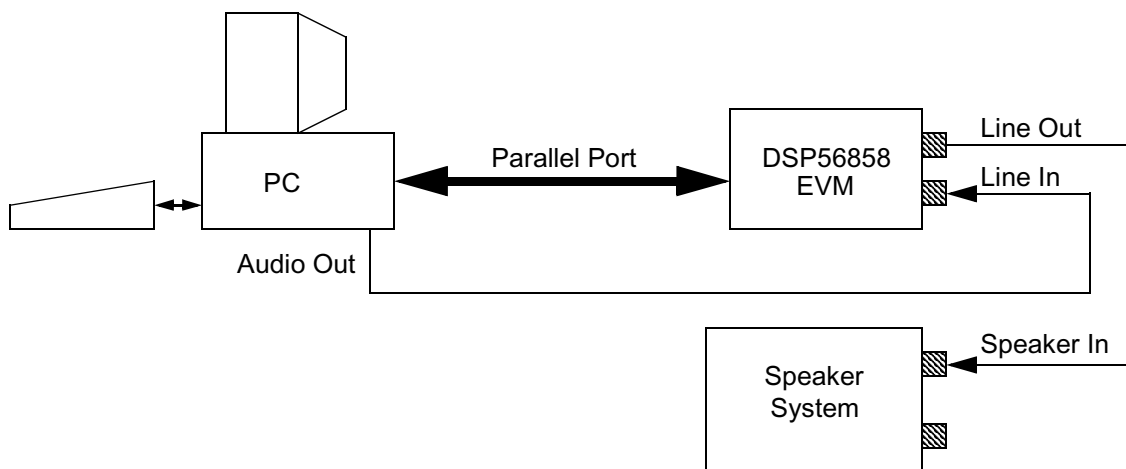
#### 9.2.2.2.1 Set-up for Triple DES Realtime Demo

##### EVM Jumper Settings:

Use the default settings shown in the **DSP56858 Evaluation Module User's Manual**.

Toggle all three switches on the "Sample Rate" dip-switch (S5) 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 the EVM board's Line Out to the speaker system's Speaker In

#### 9.2.2.2.2 Procedure for Triple DES Realtime 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 EVM's Line-In
- 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.

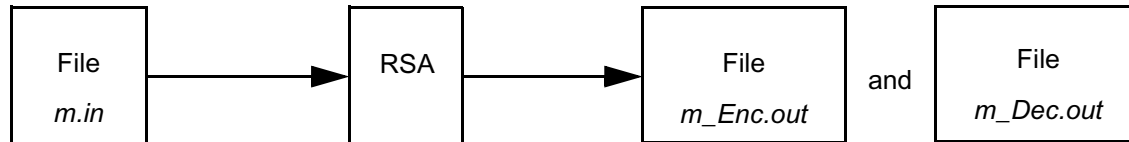


Figure 9-5. RSA Demo

### 9.2.3.1 Set-up for RSA Demo

#### EVM Jumper Settings:

Use the default settings shown in the **DSP56858 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\dsp56858evm\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\dsp56858evm\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\dsp56858evm\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 **DSP56858 Evaluation Module User's Manual**.

Toggle all three switches on the "Sample Rate" dip-switch (S5) to "Off", which will generate an 8000Hz sample rate.

The demo set-up is shown in [Figure 9-6](#).

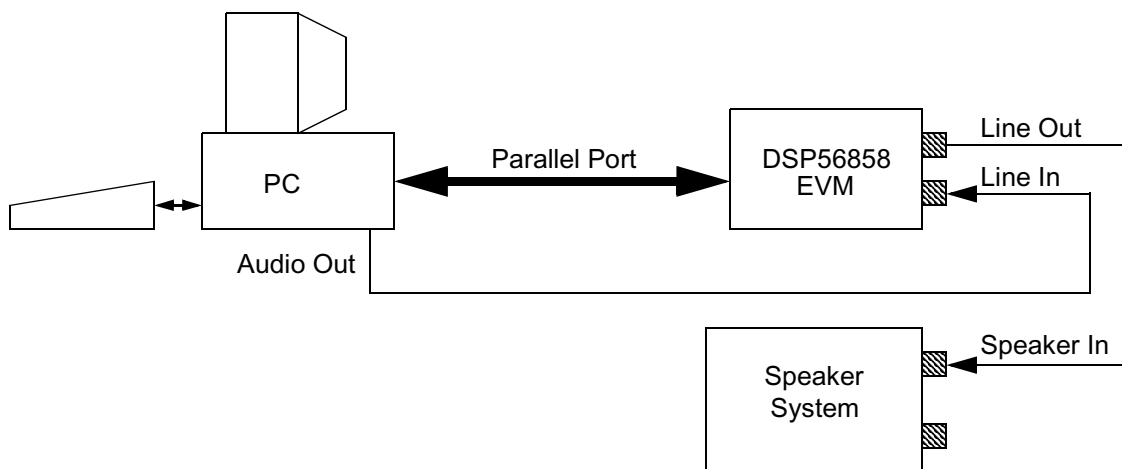


**Figure 9-6. AGC Demo Set-up**

- Connect the parallel port of the PC to the parallel port of the EVM board
- Connect PC Audio Out to the EVM board's Line In
- Connect the EVM board's Line Out to the speaker system's Speaker-In

#### 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 start, 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.

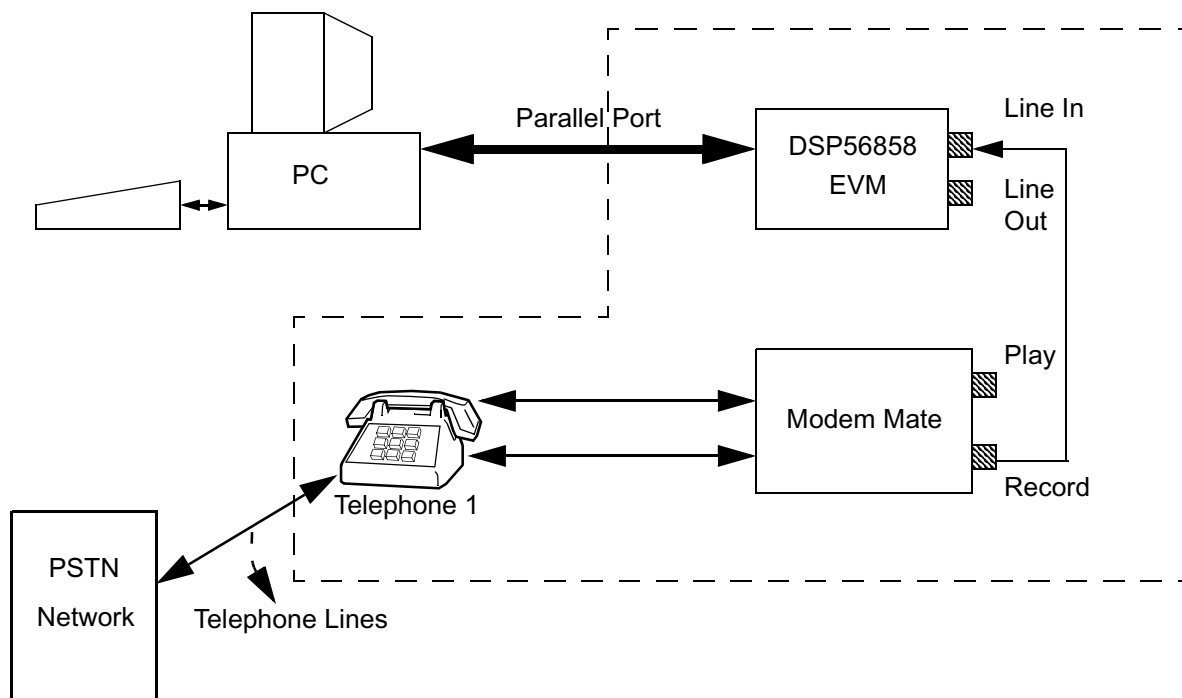
#### 9.3.2.1 Set-up for CPT Demo

##### EVM Jumper Settings:

Use the default settings shown in the **DSP56858 Evaluation Module User's Manual**.

Toggle all three switches on the "Sample Rate" dipswitch (S5) 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 (CTG) Demo

The Common Tone Generation Demo, which runs on the DSP5685x 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 the generation, the CTG 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 **DSP56858 Evaluation Module User's Manual**.

Toggle all three switches on the "Sample Rate" dipswitch (S5) to "Off", which will generate an 8000Hz sample rate.

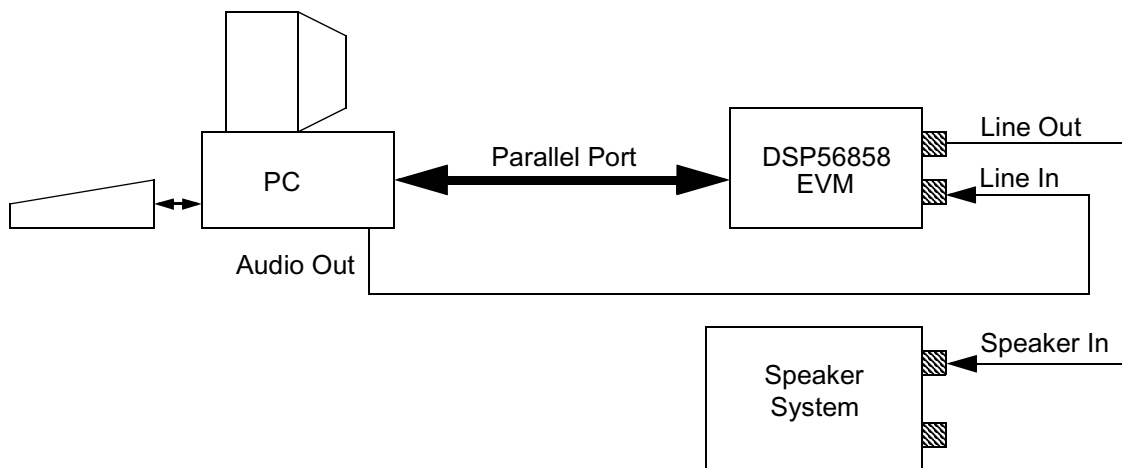


Figure 9-8. CTG Demo Set-up

- 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

### 9.3.3.2 Procedure for CTG Demo

- Using CodeWarrior, open:  
`...\nos\applications\telephony\demo_ctg\demo_ctg.mcp`
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the demo
- Call Progress Tones can now be heard from the speaker
- Once the demo is complete, CodeWarrior's console displays the message, "Demo Completed"

### 9.3.4 DTMF Detection Demo

This demo involves 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 **DSP56858 Evaluation Module User's Manual**.

Toggle all three switches on the "Sample Rate" dipswitch (S5) 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 9-9**. A device called Modem Mate from RadioShack is used in this interface.

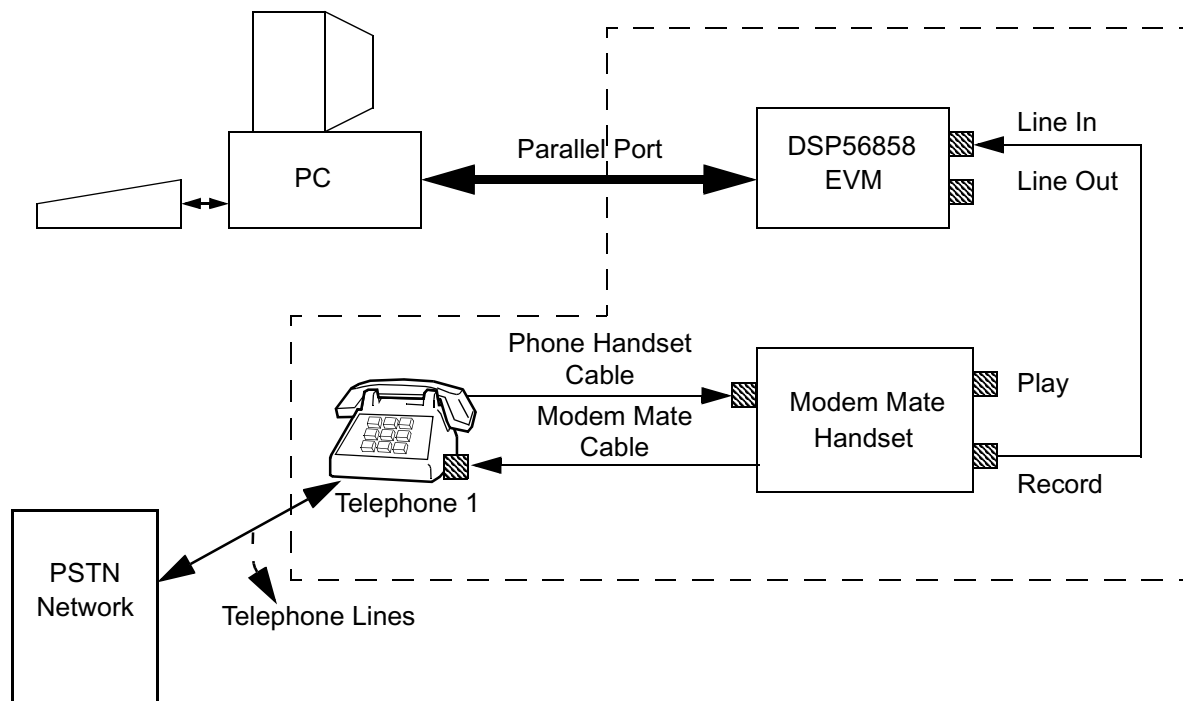


**Figure 9-9. DTMF Detection Demo Set-up**

- As shown in **Figure 9-9**, 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 DSP56858 EVM, involves 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 DSP56858EVM 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.

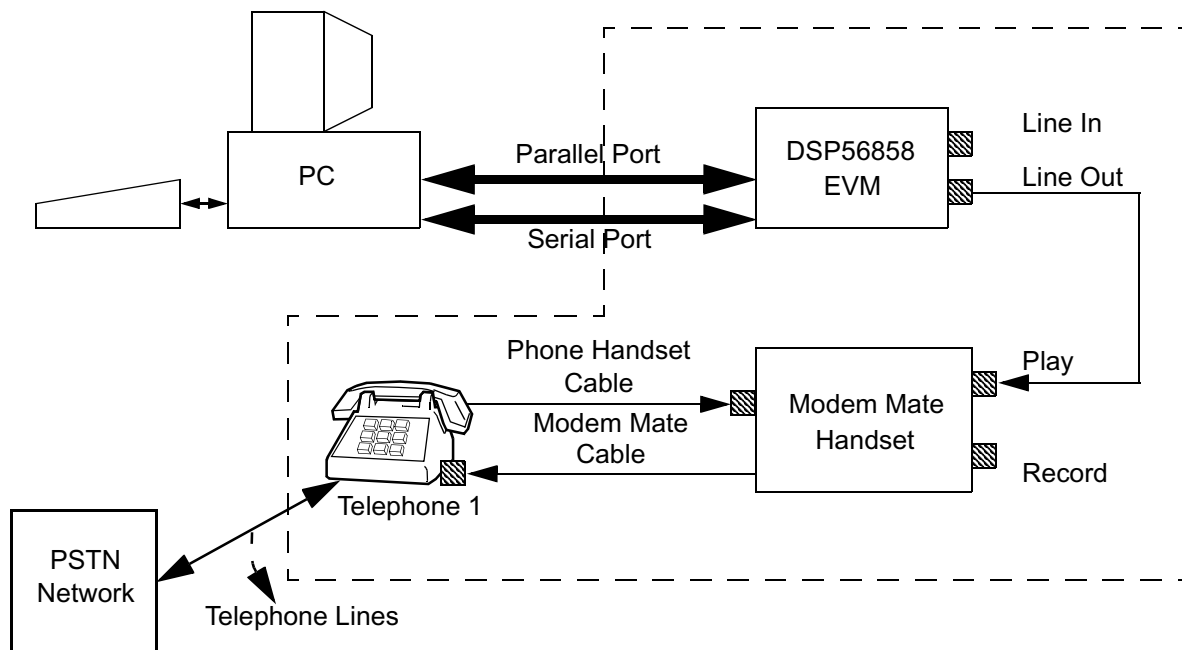
#### 9.3.5.1 Set-up for DTMF Generation Demo

##### EVM Jumper Settings:

Use the default settings shown in the **DSP56858 Evaluation Module User's Manual**.

Toggle all three switches on the "Sample Rate" dipswitch (S5) 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 9-10**. A device called Modem Mate from RadioShack is used in this interface.


**Figure 9-10. DTMF Generation Demo Set-up**

- As shown in [Figure 9-10](#), 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 DSP56858** 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 **DSP56858 Evaluation Module User’s Manual**.

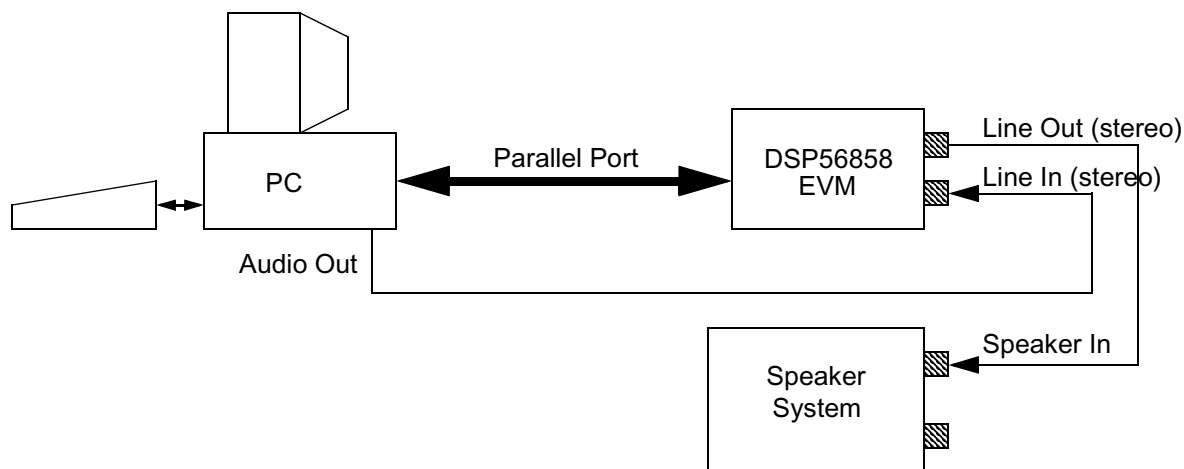


**Figure 9-11. G.165 Demo Set-up**

- 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\applications\telephony\g165\inputs`
- Play one of the speech (.wav) files in a loop (continuous) mode
- Using CodeWarrior, open:  
`..\nos\applications\telephony\g165\demo_g165.mcp`
- Set the #define BYPASS\_ECHOCANCELLER to 1, in the file *g165\_speech.c* located at :  
`..\nos\applications\telephony\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 *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

### 9.3.8 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 **DSP56858 Evaluation Module User's Manual**.



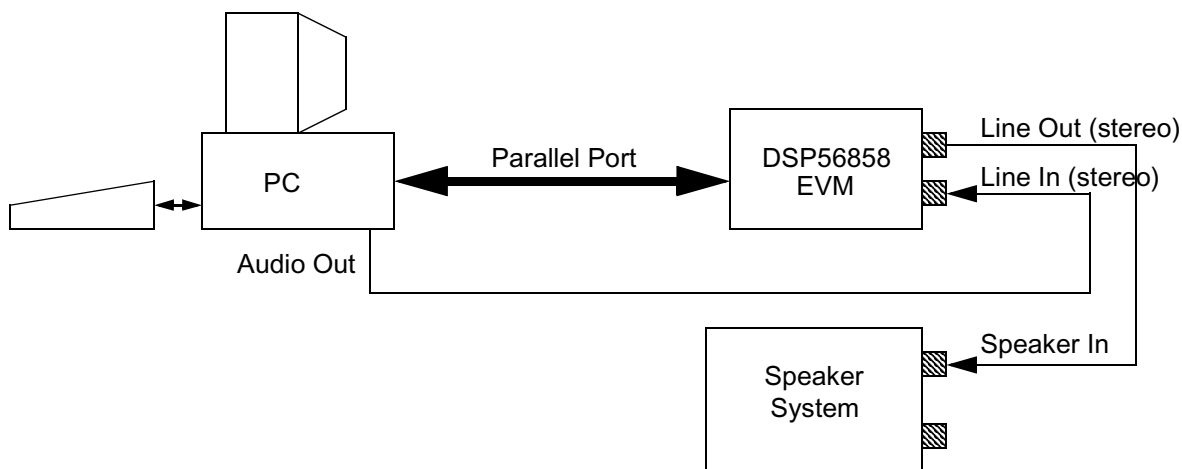


**Figure 9-12. G.168 Demo Set-up**

- 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\application\telephony\g168\inputs`
- Play one of the speech (.wav) files in a loop (continuous) mode
- Using CodeWarrior, open:  
`..\nos\application\telephony\g168\demo_g168.mcp`
- Set the `#define BYPASS_ECHOCANCELLER` to 1, in the file *g168\_speech.c* located at :  
`..\nos\application\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 **DSP56858 Evaluation Module User's Manual**.

Toggle all three switches on the "Sample Rate" dipswitch (S5) to "Off", which will generate an 8000Hz sample rate.

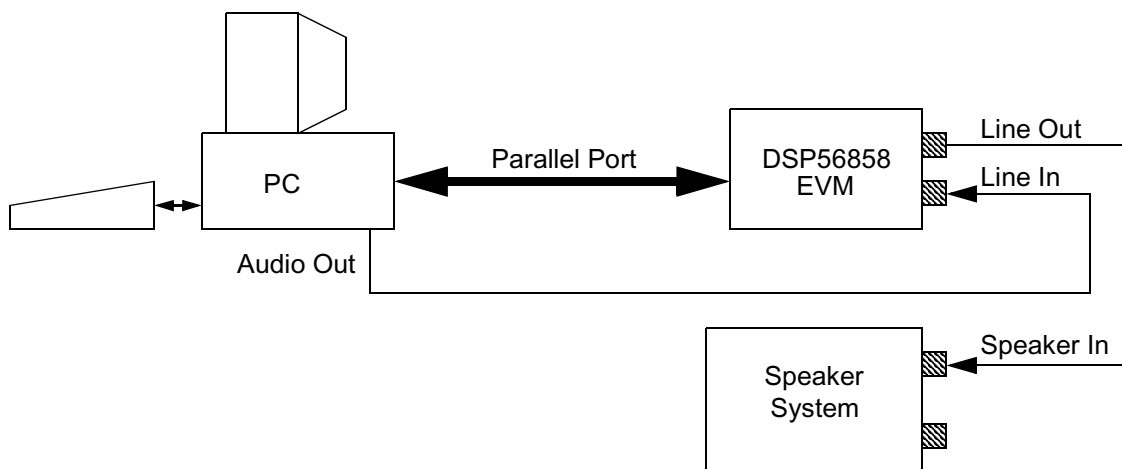


**Figure 9-13. G.711 Demo Set-up**

- 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.723.1A Demo

The G.723.1A Demo demonstrates use of the G.723.1A library provided by the SDK.

In this demo, audio is input from a PC to the G.723.1A Speech Codec (encoder and decoder) algorithm and the decoded output is played by the G.723.1A Speech Codec on a speaker system.

For more information about the G.723.1A library, see the **G.723.1A Library** manual.

### 9.3.10.1 Set-up for the G.723.1A Demo

#### EVM Jumper Settings:

Use the default settings shown in the **DSP56858 Evaluation Module User's Manual**.

Toggle all three switches on the "Sample Rate" dipswitch (S5) to "Off", which will generate an 8000Hz sample rate.

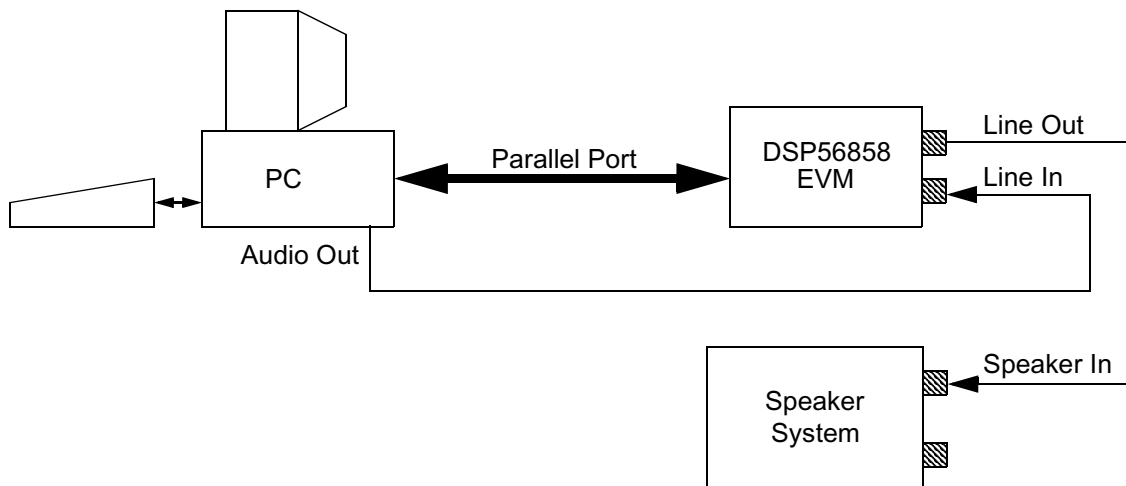


**Figure 9-14. G.723.1A ADPCM Codec Test Set-up**

- 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.723.1A Demo

- Using CodeWarrior, open:  
`...\nos\applications\telephony\g723\g723_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 board
- The same audio signal should be available at the speaker system output, with an increase in noise level when the rate decreases from 6.3 to 5.3 Kbits/s

### 9.3.11 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.

### 9.3.11.1 Set-up for the G.726 Demo

#### EVM Jumper Settings:

Use the default settings shown in the **DSP56858 Evaluation Module User's Manual**.

Toggle all three switches on the "Sample Rate" dipswitch (S5) to "Off", which will generate an 8000Hz sample rate.

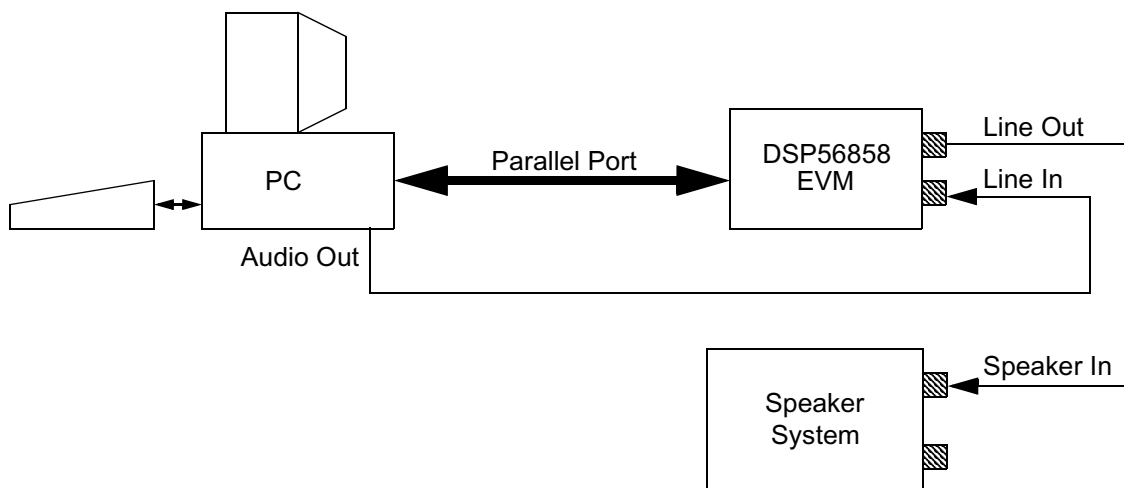


**Figure 9-15. G.726 ADPCM Codec Test Set-up**

- 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

### 9.3.11.2 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, 16 Kbits/s) decreases from 40 to 16 Kbits/s

## 9.3.12 G729 Demo

### 9.3.12.1 Vocoder Recorder Player Demo

The recorder player demo allows comparison of the quality of decoded speech using Motorola PCM, G.729AB and G.711.

In this demo, the voice is captured using a microphone, digitized by the codec, and encoded using the specified vocoder. It's then sent to the PC, which stores the sampled voice using serial communication. The encoded voice samples are then sent back from the PC to the DSP56858EVM for decoding, using the specified vocoder. The reconstructed voice samples are then played back on the speakers.

### 9.3.12.1.1 Set-up for Vocoder Recorder Player Demo

#### EVM Jumper Settings:

Use default settings as shown in the **DSP56858 Evaluation Module User's Manual**.

Toggle all three switches on the "Sample Rate" dipswitch (S5) to "Off", which will generate an 8000Hz sample rate.

Figure 9-16 shows how to set up the Recorder Player Demo.

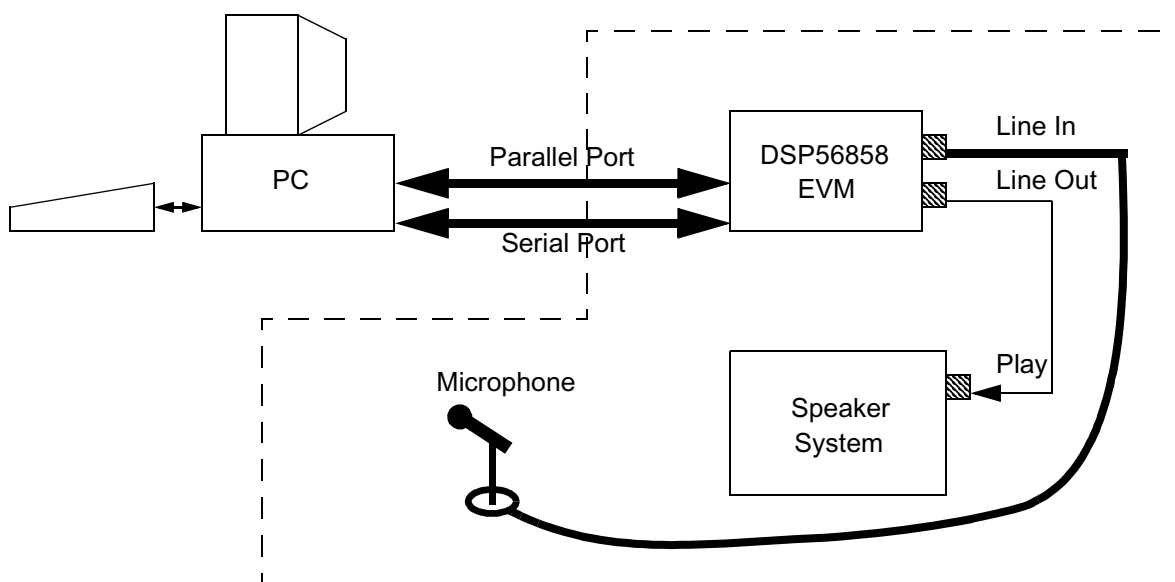


**Figure 9-16. Recorder Player Demo Set-up**

- Connect the serial cable between the EVM and the COM1, COM2, COM3, or COM4 port of the computer
- Connect the parallel port of the PC to the parallel port of the EVM board
- Connect an amplified microphone to Line In of the EVM
- Connect Line Out of the EVM to a speaker or headphone

### 9.3.12.1.2 Procedure for Vocoder Recorder Player Demo

- Run/launch the *serial\_mix.exe* found in:  
{EmbeddedSDK\src\x86\win32\applications\recorder\_serial\_player\_server\project\Release\}
- Select the correct Communication port (COM1, COM2, COM3 or COM4)
- Using CodeWarrior, open:  
.../dsp56858evm/nos/applications/telephony/recorder\_player/recorder\_player.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test

- Select from the following options shown in the PC console/screen:  
Motorola DSP5685x G.729ab and G.711 Vocoders demo  
Options  
-----  
[1] PCM (8bit)  
\* [2] G711  
[3] G729AB  
[6] Quit

Press Motorola DSP5685x EVM IRQA to record speech samples and IRQB to playback

## 9.3.12.2 Vocoder Loopback Demo

The loopback demo allows comparison of the quality of vocoders using Motorola PCM, G.729AB, G.726 (different compression modes: 16 and 40 kbits), and G.711 vocoders.

In the Loopback Demo, the voice is captured using a microphone, digitized by the codec, then encoded and decoded using the specified vocoder. It's then sent back to the codec, which converts the reconstructed voice samples and plays them back on speakers.

### 9.3.12.2.1 Set-up for Vocoder Loopback Demo

#### EVM Jumper Settings:

Use default settings as shown in the **DSP56858 Evaluation Module User's Manual**.

Toggle all three switches on the "Sample Rate" dipswitch (S5) to "Off", which will generate an 8000Hz sample rate.

Figure 9-17 shows the Vocoder Loopback demo.

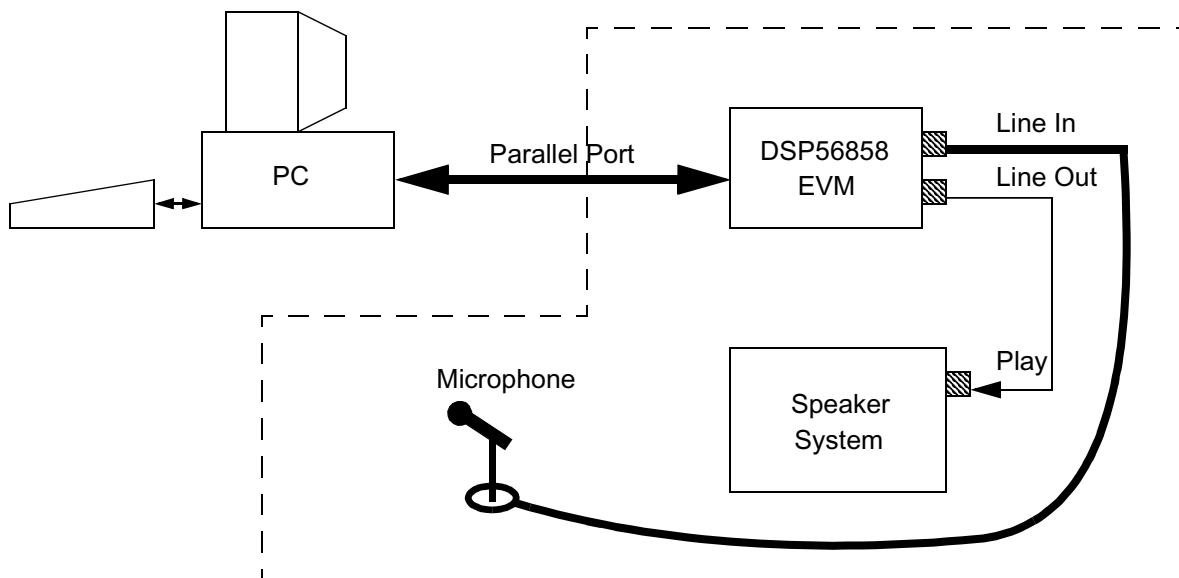


Figure 9-17. Vocoder Loopback Demo

- Connect the parallel port of the PC to the parallel port of the EVM board
- Connect an amplified microphone to Line In of the EV<
- Connect Line Out of the EVM to speaker or headphone

### 9.3.12.3 Procedure for Vocoder Loopback Demo

- Using CodeWarrior, open:  
.../dsp56858evm/nos/applications/telephony/loopback\_vocoders/loopback.mcp
- Build and download the project; when the debugger reaches *main()*, it will stop
- Select *Run* in the debugger to continue executing the test
- Use IRQA to toggle through the vocoders and IRQB to exit; an LED will blink to indicate the selection

Vocoder Demo

=====

1. PCM at 8KHz (Red LED)
2. G.711 (Yellow LED)
3. G.726 16Kbits (Green LED)
4. G.726 40Kbits (Red 2 LED)
5. G.729AB (Yellow 2 LED)
6. Exit

### 9.3.13 Noise Suppression (NS) Demo

The Noise Suppression Demo demonstrates the use of the Noise Suppression 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.

#### 9.3.13.1 Set-up for Noise Suppression Demo

##### EVM Jumper Settings:

Use the default settings shown in the **DSP56858 Evaluation Module User's Manual**.

Toggle all three switches on the "Sample Rate" dipswitch (S5) to "Off", which will generate an 8000Hz sample rate.

The demo set-up is shown in [Figure 9-18](#).

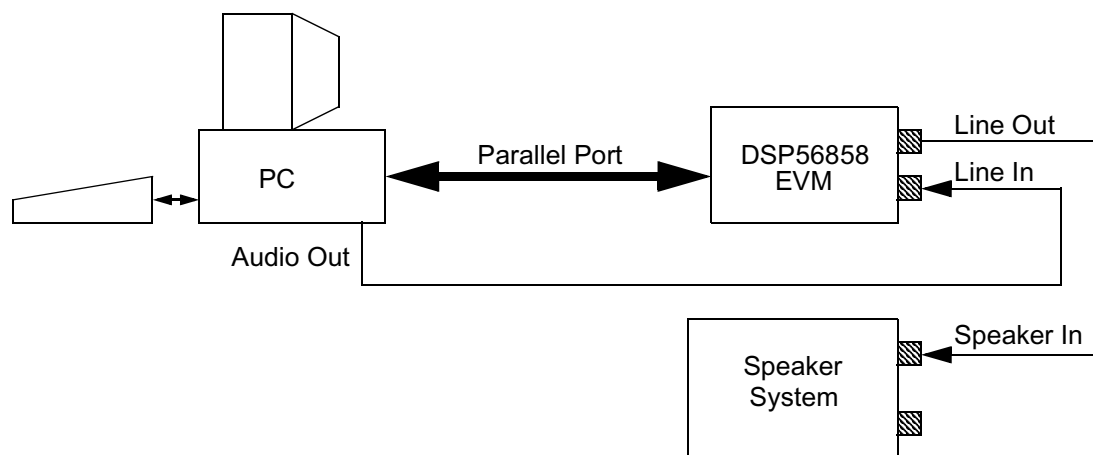


**Figure 9-18. Noise Suppression Demo Set-up**

- 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 SpeakerIn of the speaker system



### 9.3.13.2 Procedure for Noise Suppression 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 Noise Suppression is disabled, the input audio will appear unaltered at the EVM's Line Out
  - 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
  - Noise Suppression can be disabled by pressing the IRQA Button once
    - The Green LED should turn off

### 9.3.14 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.

#### 9.3.14.1 Set-up for VAD Demo

##### EVM Jumper Settings:

Use the default settings shown in the **DSP56858 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\dsp56858evm\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\dsp56858evm\nos\applications\telephony\demo_vad\inputs\`

#### 9.3.14.2 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\dsp56858evm\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\dsp56858evm\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 9-19](#). A device called Modem Mate from RadioShack is used for this interface.

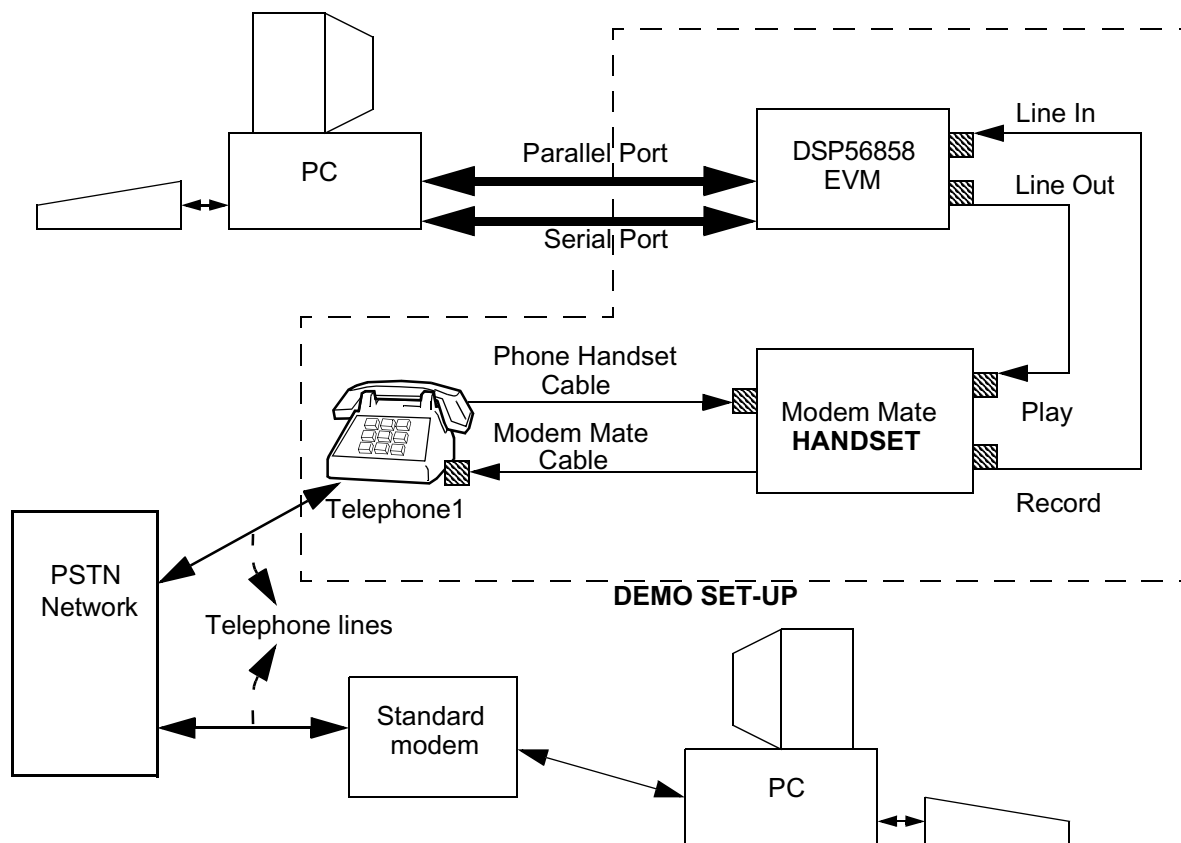


**Figure 9-19. V.22bis Demo Set-up**

- As shown in [Figure 9-19](#), 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

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 **DSP56858 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.

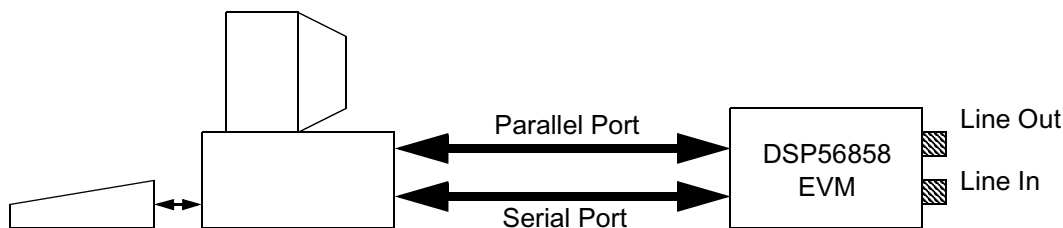


Figure 9-20. V.42bis Demo Set-up

#### 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 Operator<sup>1</sup>. 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 9-21](#). The description of each state follows.

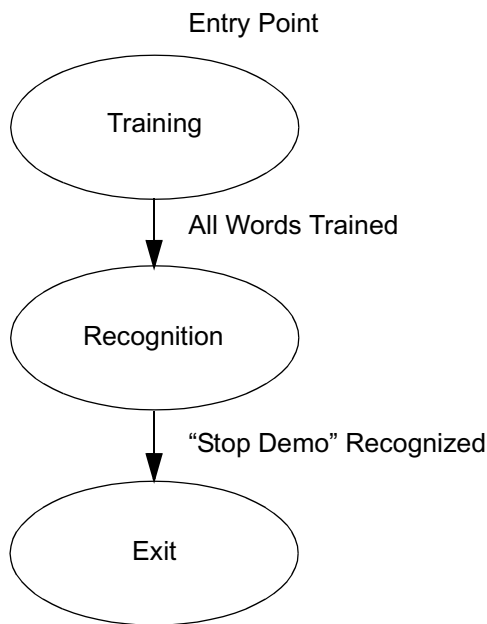


**Figure 9-21. State Diagram of VRLite-1 Demo System**

**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 9-22](#).

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

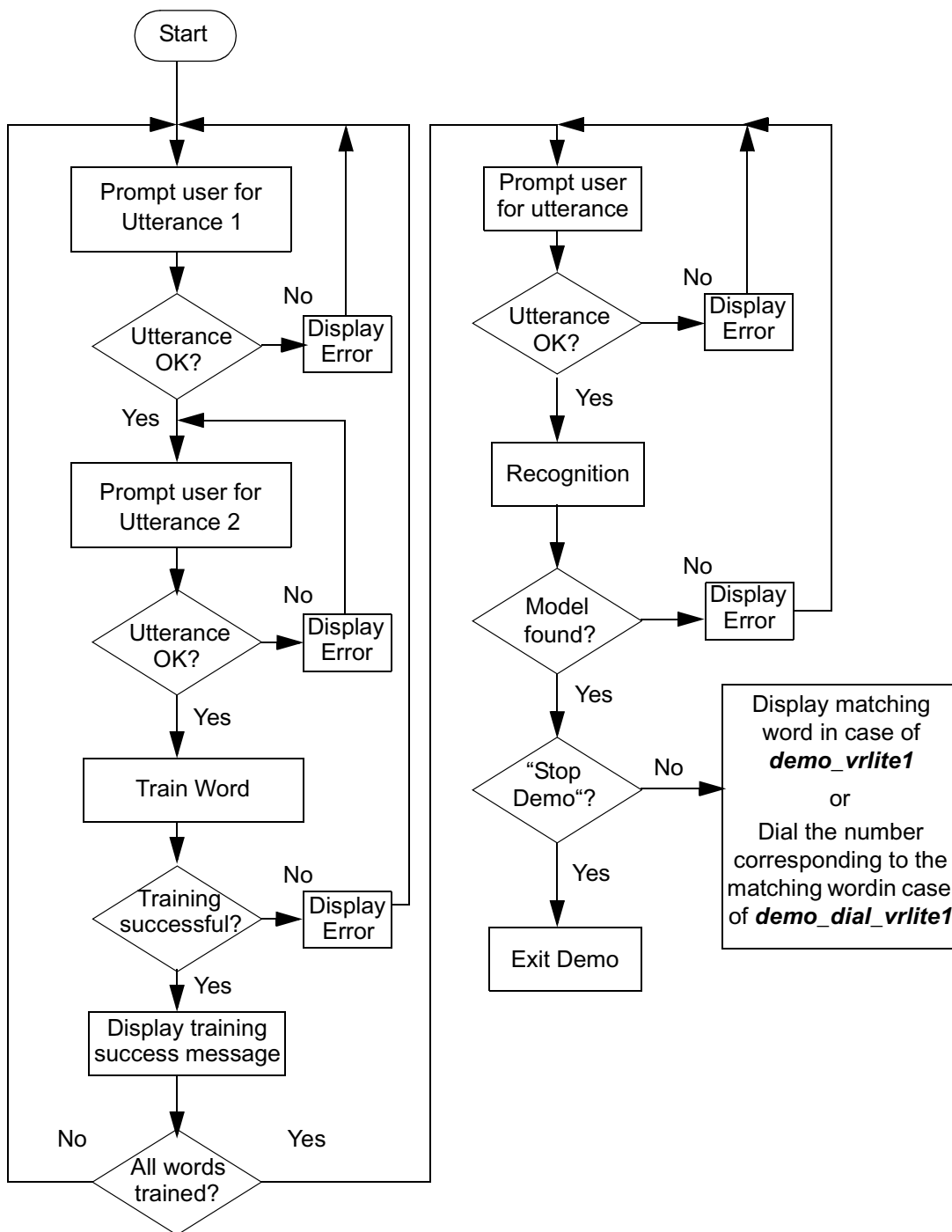


Figure 9-22. Flowchart for VRLite-1 Demo System

Demo details:

- All words defined by the application need to 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 **DSP56858 Evaluation Module User's Manual**.

Toggle all three switches on the "Sample Rate" dipswitch (S5) to "Off", which will generate an 8000Hz sample rate.

The set-up for *demo\_vrlite1* is shown in [Figure 9-23](#).

- 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.

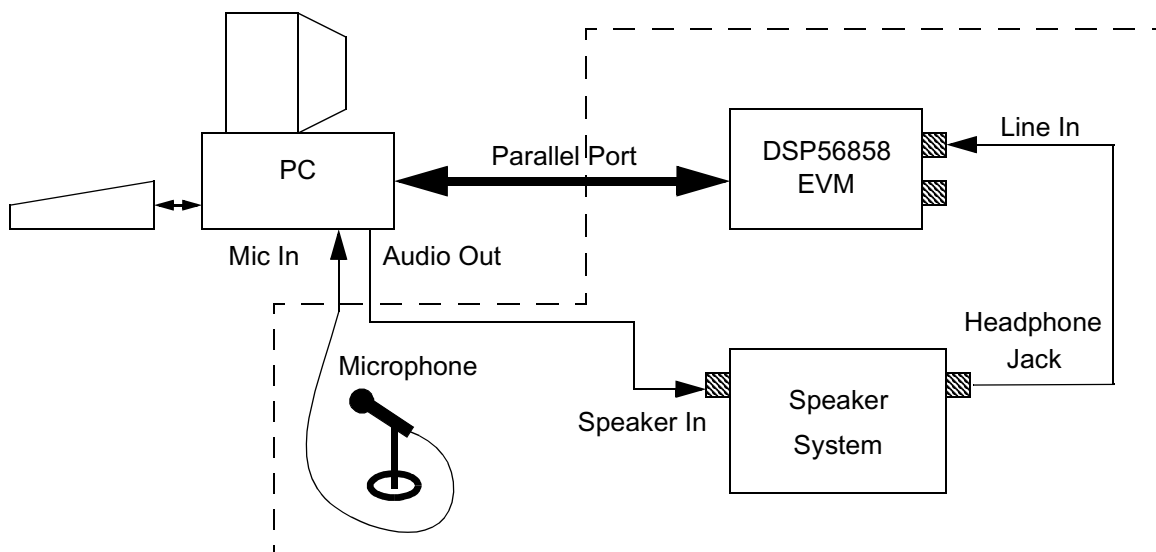


**Figure 9-23. Demo Set-up for *demo\_vrlite1***

The setup for *demo\_dial\_vrlite1* is shown in [Figure 9-24](#).

- 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.

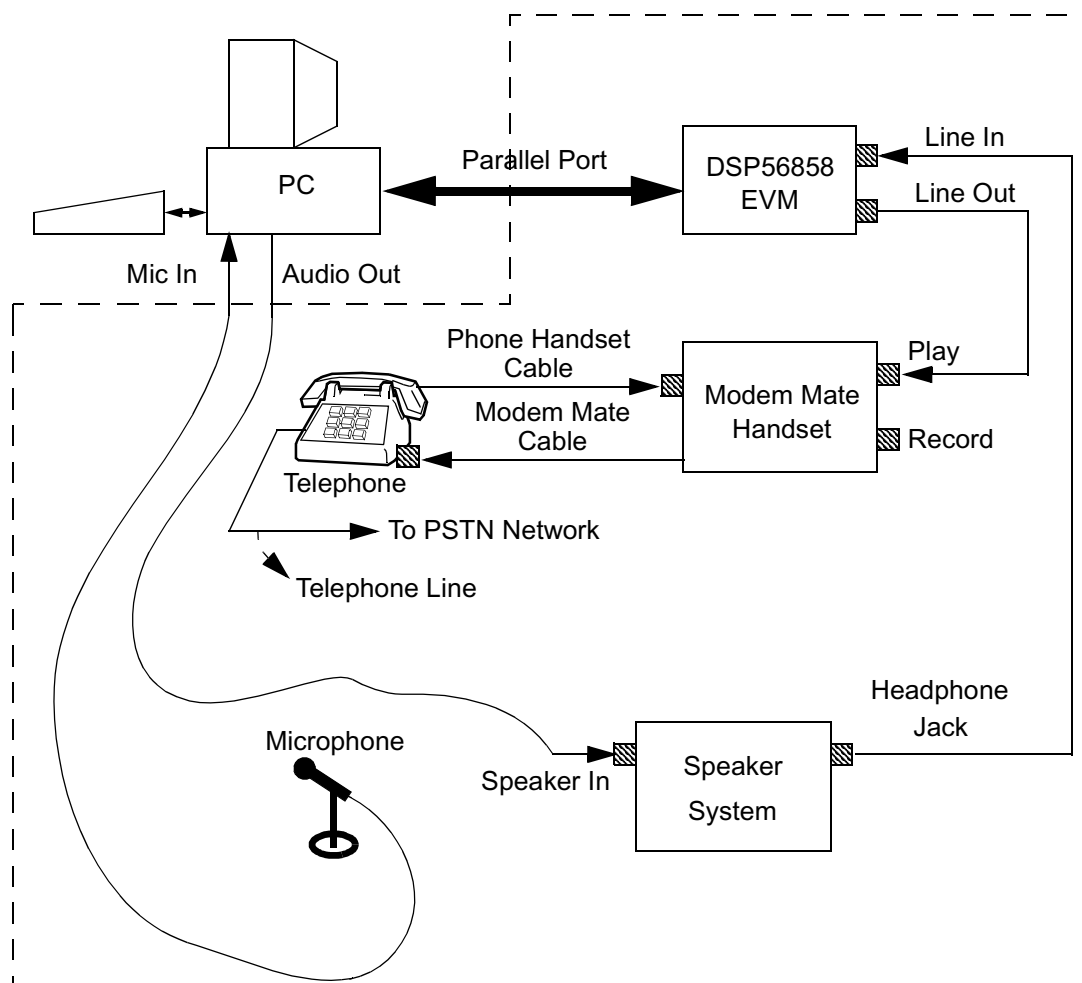


Figure 9-24. Demo Set-up for *demo\_dial\_vrlite1*

### 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 DSP56858 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 9.5.1.2.2](#), 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 [xxxv](#)

## A

ADPCM [xxxv](#)

AEC [xxxv](#)

autoCorr Performance [7-4](#)

## B

BSP [xxxv](#)

buildall.mcp [1-3](#)

## C

Caller ID Detection Test [8-8](#)

Caller ID Library [7-25](#), [7-26](#)

CAS [xxxv](#)

CBC [xxxv](#)

cbitrev Performance [7-4](#)

CFB [xxxv](#)

CFFT [xxxv](#)

CID [xxxv](#)

cid1.lib [8-39](#)

cid1test.mcp [8-39](#)

CODEC [xxxv](#)

Codec Driver [6-9](#), [6-38](#), [6-178](#), [6-199](#)

CodeWarrior

Registration [1-2](#)

Common Tone Generation Demo [9-11](#)

Conventions [xxxiv](#)

COP [xxxv](#)

CPE [xxxv](#)

CPT [xxxv](#)

CPT Detection Test [8-12](#)

## D

DCE [xxxv](#)

Decoder [8-18](#)

DES [xxxv](#)

DES Library [7-15](#)

DES Loopback test [8-5](#)

Directory Structure [2-1](#)

domain-specific libraries [7-1](#)

DSP [xxxv](#)

DSP56800E Reference Manual [xxxvi](#)

DSP5685x 16-Bit Digital Signal Processor User's  
Manual [xxxvi](#)

DTE [xxxv](#)

DTMF [xxxv](#)

DTMF Detection Library [7-27](#)

DTMF Generation Test [8-14](#)

## E

Embedded SDK Programmer's Guide [xxxvi](#)

Encoder [8-18](#)

ERLE [xxxv](#)

ESSI Driver [5-45](#), [5-79](#)

ESSI Driver Arguments - essioctl [5-76](#), [5-103](#)

ESSI Driver Arguments - ioctl [5-69](#), [5-95](#)

ESSI essioctl Commands [5-76](#)

ESSI ioctl Commands [5-69](#), [5-95](#)

EVM [xxxv](#)

## F

FFT [xxxv](#)

FILE I/O Driver [6-66](#), [6-178](#)

FIR [xxxv](#)

## G

GPIO Driver [5-106](#)

## H

HI Driver Arguments - hiioctl [5-137](#)

HI Driver Arguments - ioctl [5-128](#)

HI hiioctl Commands [5-137](#)

HI ioctl Commands [5-128](#)

Hold Release Logic [9-16](#), [9-17](#)

## I

I/O [xxxv](#)

IDE [xxxv](#)

IIR [xxxv](#)

Install CodeWarrior [1-1](#)

Install DSP56858EVM Hardware [1-2](#)

Install SDK [1-2](#)

ioctl [5-111](#)

ITCN [xxxv](#)

## L

LED Driver [6-107](#)

LED Driver Arguments - ioctl [6-112](#)

LED Ioctl Commands [6-112](#)

license key [1-2](#)

Low-Level LED Driver Arguments [6-117](#)

LSB [xxxv](#)



## M

MAC [xxxv](#)  
Memory and MIPS Requirements [7-40](#), [7-41](#), [7-43](#)  
memory and MIPS requirements [7-42](#)  
Metrowerks CodeWarrior [1-1](#)  
MIPS [xxxv](#)  
Modem Libraries [7-9](#)  
MSB [xxxv](#)

## N

NLMS [xxxv](#)

## O

OMR [xxxv](#)  
OnCE [xxxv](#)

## P

PC [xxxv](#)  
PC Master appconfig.h Settings [6-119](#)  
PC Master SCI Communication Driver [6-119](#)  
PCM [xxxv](#), [7-33](#)  
PLL [xxxv](#)  
PSTN [xxxv](#)

## Q

Quad Timer Driver [5-152](#)  
Quick Start [1-1](#)

## R

Register CodeWarrior [1-2](#)  
RFFT [xxxv](#)  
RinBuffer [8-15](#), [8-16](#)  
RSA [xxxv](#)

## S

SAS [xxxvi](#)  
SCI [xxxvi](#)  
SCI Driver [5-170](#)  
SCI Driver Arguments - ioctl [5-181](#)  
SCI Driver Arguments - scioctl [5-189](#)  
SCI ioctl Commands [5-181](#), [5-189](#)  
SDK [xxxvi](#)  
Security Libraries [7-14](#)  
Set-up for G.711 Test [8-17](#)  
Set-up for G.723.1A Test [8-18](#)  
Set-up for G.726 Test [8-19](#)  
Set-up for V.21 Loopback Test [8-24](#)  
Set-up for V.22bis Loopback Test [8-25](#)  
Set-up for V.8bis Test [8-27](#)  
Set-up for VAD Test [8-23](#)  
Set-up for Vocoder Loopback Demo [9-23](#)  
Set-up for Vocoder Recorder Player Demo [9-22](#)

Set-up for VRLite-1 Test [8-29](#)  
SIM [xxxvi](#)  
SinBuffer [8-15](#), [8-16](#)  
SP [xxxvi](#)  
SPCS [xxxvi](#)  
SPI [xxxvi](#)  
SPI Driver [5-193](#)  
SPI Driver Arguments - ioctl [5-201](#)  
SPI Driver Arguments - spiioctl [5-208](#)  
SPI ioctl Commands [5-201](#)  
SPI spiioctl Commands [5-208](#)  
SR [xxxvi](#)  
SRC [xxxvi](#)  
SSI [xxxvi](#)

## T

teldefs.h [8-32](#)  
Test Setup [8-19](#)  
Timer Demo [9-2](#)  
Timer Driver [5-210](#)  
Tone Disabler [9-16](#), [9-17](#)  
Trigonometric Math Function Performance [7-3](#)  
Triple DES Library [7-16](#)  
Type1CID [8-32](#), [8-39](#)

## V

V.21 Loopback Test [8-24](#)  
V.22bis Library [7-12](#)  
V.22bis Loopback Test [8-25](#)  
V.42bis Library [7-13](#)  
V.8bis Library [7-9](#)  
V.8bis Test [8-26](#)  
VAD [xxxvi](#)  
    Test Set-up [8-8](#), [8-22](#)  
VAD Demo [9-24](#)  
VAD Test [8-22](#), [8-23](#)  
VR [xxxvi](#)  
VR Lite-1 Demo [9-33](#)  
VRLite-1 Demo [9-33](#)



# **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](http://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](http://www.freescale.com)**

**SDK137/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