VoicePath™ API-II

CSLAC Reference Guide

Part Number:	**Le71SDK0002**
Document Id#:	**081301**
Revision Number:	**20**
Issue Date:	**June 5, 2014**

Microsemi®

This page left intentionally blank

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION . . . 1
1.1 About this User's Guide . . . 1
1.1.1 Chapter Overview . . . 1
1.1.2 Frequently Used Terms . . . 2
1.1.3 Documentation Conventions . . . 2
1.2 VP-API-II Overview . . . 2
1.2.1 Features . . . 2
1.2.2 Architecture . . . 4
1.2.3 Supported Hardware Configurations . . . 5
1.2.4 VP-API-II Function Summary . . . 8
1.2.5 Basic VP-API-II Data Types . . . 10
1.2.6 VP-API-II Function Return Type . . . 11
1.3 API-II Source Version Number . . . 11
1.4 Technical Support . . . 12

CHAPTER 2 PROFILES . . . 13
2.1 Overview . . . 13
2.2 Profile Types . . . 13
2.3 Profile Tables . . . 14
2.4 Profile Functions . . . 15
2.5 Important Notes About CSLAC and Profiles . . . 15

CHAPTER 3 OPTIONS . . . 17
3.1 Overview . . . 17
3.2 Option Summary . . . 17
3.3 Option Descriptions . . . 19
3.3.1 VP_DEVICE_OPTION_ID_CRITICAL_FLT . . . 19
3.3.2 VP_OPTION_ID_SWITCHER_CTRL . . . 19
3.3.3 VP_OPTION_ID_DEBUG_SELECT . . . 19
3.3.4 VP_OPTION_ID_PULSE_MODE . . . 20
3.3.5 VP_DEVICE_OPTION_ID_PULSE . . . 20
3.3.6 VP_DEVICE_OPTION_ID_PULSE2 . . . 22
3.3.7 VP_OPTION_ID_TIMESLOT . . . 22
3.3.8 VP_OPTION_ID_CODEC . . . 23
3.3.9 VP_OPTION_ID_PCM_HWY . . . 25
3.3.10 VP_OPTION_ID_PCM_TXRX_CNTRL . . . 25
3.3.11 VP_OPTION_ID_RING_CNTRL . . . 26
3.3.12 VP_OPTION_ID_LOOPBACK . . . 27
3.3.13 VP_OPTION_ID_LINE_STATE . . . 28
3.3.14 VP_OPTION_ID_EVENT_MASK . . . 29
3.3.15 VP_DEVICE_OPTION_ID_DEVICE_IO . . . 30
3.3.16 VP_OPTION_ID_ABS_GAIN . . . 31
3.3.17 VP_OPTION_ID_DCFEED_PARAMS . . . 33
3.3.18 VP_OPTION_ID_RINGING_PARAMS . . . 34

CHAPTER 4 DESIGN CONSIDERATIONS . . . 35
4.1 Return Code Handling . . . 35
4.2 Initialization of the VP-API-II . . . 35

4.2.1 Basic Initialization of the VP-API-II . . . 35
4.2.2 Calibration Bypass . . . 36
4.3 TickRate Requirements . . . 38
4.3.1 Dial Pulse Detection . . . 38
4.3.2 FSK Caller ID . . . 39
4.4 Cadence Timing . . . 39
4.5 Soft Profile Tables . . . 40
4.6 Multi-Tasking Applications . . . 41
4.6.1 Multiple Device and Line Contexts . . . 41
4.6.2 Protected Memory Requirements . . . 42
4.7 Code Size Management . . . 43
4.7.1 Common Code Size Reductions . . . 44
4.7.2 VE880 Code Size Reductions . . . 45
4.7.3 VE890 Code Size Reductions . . . 47

CHAPTER 5 SYSTEM CONFIGURATION FUNCTIONS . . . 49
5.1 Overview . . . 49
5.2 Objects and Contexts . . . 49
5.3 Function Descriptions . . . 51
5.3.1 VpMakeDeviceObject() . . . 51
5.3.2 VpMakeLineObject() . . . 52
5.3.3 VpMakeDeviceCtx() . . . 53
5.3.4 VpMakeLineCtx() . . . 53
5.3.5 VpFreeLineCtx() . . . 54
5.3.6 VpMapLineId() . . . 54

CHAPTER 6 INITIALIZATION FUNCTIONS . . . 55
6.1 Overview . . . 55
6.2 Function Descriptions . . . 56
6.2.1 VpInitDevice() . . . 56
6.2.2 VpInitLine() . . . 58
6.2.3 VpConfigLine() . . . 59
6.2.4 VpInitCustomTermType() . . . 60
6.2.5 VpCal() . . . 61
6.2.6 VpCalLine() . . . 63
6.2.7 VpInitRing() . . . 64
6.2.8 VpInitCid() . . . 65
6.2.9 VpInitMeter() . . . 65
6.2.10 VpInitProfile() . . . 66

CHAPTER 7 INTERRUPT AND EVENT HANDLING . . . 67
7.1 Overview . . . 67
7.2 Handling Interrupts from CSLAC Devices . . . 67
7.2.1 VP-API-II Time Base . . . 67
7.2.2 CSLAC Device Interrupt Architecture Variations . . . 68
7.3 Function Descriptions . . . 73
7.3.1 VpApiTick() . . . 73
7.3.2 VpVirtualISR() . . . 73
7.3.3 VpGetEvent() . . . 74
7.3.4 VpFlushEvents() . . . 75
7.3.5 VpGetResults() . . . 76

CHAPTER 8 EVENTS . . . 77
8.1 Overview . . . 77
8.2 Event Summary . . . 78
8.3 Fault Events . . . 80

8.3.1 VP_DEV_EVID_BAT_FLT . . . 80
8.3.2 VP_DEV_EVID_CLK_FLT . . . 80
8.3.3 VP_LINE_EVID_THERM_FLT . . . 80
8.3.4 VP_LINE_EVID_DC_FLT . . . 81
8.3.5 VP_LINE_EVID_AC_FLT . . . 81
8.3.6 VP_LINE_EVID_RES_LEAK_FLT . . . 81
8.4 Signaling Events . . . 82
8.4.1 VP_LINE_EVID_HOOK_OFF . . . 82
8.4.2 VP_LINE_EVID_HOOK_ON . . . 82
8.4.3 VP_LINE_EVID_GKEY_DET . . . 83
8.4.4 VP_LINE_EVID_GKEY_REL . . . 83
8.4.5 VP_LINE_EVID_FLASH . . . 83
8.4.6 VP_LINE_EVID_STARTPULSE . . . 84
8.4.7 VP_LINE_EVID_EXTD_FLASH . . . 84
8.4.8 VP_LINE_EVID_DTMF_DIG . . . 84
8.4.9 VP_LINE_EVID_PULSE_DIG . . . 85
8.4.10 VP_DEV_EVID_TS_ROLLOVER . . . 85
8.5 Response Events . . . 86
8.5.1 VP_LINE_EVID_RD_OPTION . . . 86
8.5.2 VP_EVID_CAL_CMP . . . 86
8.5.3 VP_EVID_CAL_BUSY . . . 86
8.5.4 VP_LINE_EVID_RD_LOOP . . . 87
8.5.5 VP_LINE_EVID_GAIN_CMP . . . 88
8.5.6 VP_DEV_EVID_DEV_INIT_CMP . . . 89
8.5.7 VP_LINE_EVID_LINE_INIT_CMP . . . 89
8.5.8 VP_DEV_EVID_IO_ACCESS_CMP . . . 89
8.6 Process Events . . . 90
8.6.1 VP_LINE_EVID_MTR_CMP . . . 90
8.6.2 VP_LINE_EVID_MTR_ABORT . . . 90
8.6.3 VP_LINE_EVID_CID_DATA . . . 91
8.6.4 VP_LINE_EVID_RING_CAD . . . 91
8.6.5 VP_LINE_EVID_SIGNAL_CMP . . . 92
8.6.6 VP_LINE_EVID_TONE_CAD . . . 92
8.7 FXO Events . . . 93
8.7.1 VP_LINE_EVID_RING_ON . . . 93
8.7.2 VP_LINE_EVID_RING_OFF . . . 93
8.7.3 VP_LINE_EVID_LIU . . . 93
8.7.4 VP_LINE_EVID_LNIU . . . 94
8.7.5 VP_LINE_EVID_FEED_DIS . . . 94
8.7.6 VP_LINE_EVID_FEED_EN . . . 94
8.7.7 VP_LINE_EVID_DISCONNECT . . . 94
8.7.8 VP_LINE_EVID_RECONNECT . . . 95
8.7.9 VP_LINE_EVID_POLREV . . . 95
8.7.10 VP_LINE_EVID_POH . . . 95
8.7.11 VP_LINE_EVID_PNOH . . . 96
8.8 Test Events . . . 97
8.8.1 VP_LINE_EVID_TEST_CMP . . . 97

CHAPTER 9 CONTROL FUNCTIONS . . . 99
9.1 Overview . . . 99
9.2 Function Descriptions . . . 100
9.2.1 VpFreeRun() . . . 100
9.2.2 VpSetLineState() . . . 101
9.2.3 VpSetRelayState() . . . 103

9.2.4 VpSetOption() . . . 104
9.2.5 VpSetLineTone() . . . 105
9.2.6 VpSetRelGain() . . . 107
9.2.7 VpSendSignal() . . . 108
9.2.8 VpSendCid() . . . 109
9.2.9 VpContinueCid() . . . 110
9.2.10 VpStartMeter() . . . 110
9.2.11 VpDtmfDigitDetected() . . . 111
9.2.12 VpDeviceIoAccess() . . . 112

CHAPTER 10 STATUS AND QUERY FUNCTIONS . . . 115
10.1 Overview . . . 115
10.2 Function Descriptions . . . 115
10.2.1 VpGetLoopCond() . . . 115
10.2.2 VpGetLineStatus() . . . 116
10.2.3 VpGetDeviceStatus() . . . 117
10.2.4 VpGetLineInfo() . . . 118
10.2.5 VpGetDeviceInfo() . . . 119
10.2.6 VpGetOption() . . . 120
10.2.7 VpGetLineState() . . . 121

CHAPTER 11 SYSTEM SERVICES . . . 123
11.1 Overview . . . 123
11.2 Function Descriptions . . . 123
11.2.1 VpSysWait() . . . 123
11.2.2 VpSysEnterCritical() and VpSysExitCritical() . . . 124
11.2.3 VpSysDisableInt() and VpSysEnableInt() . . . 125
11.2.4 VpSysTestInt() . . . 125
11.2.5 VpSysDtmfDetEnable() and VpSysDtmfDetDisable() . . . 126
11.2.6 VpSysDebugPrintf() . . . 126

CHAPTER 12 HARDWARE ABSTRACTION LAYER . . . 127
12.1 Overview . . . 127
12.2 Function Descriptions . . . 128
12.2.1 VpMpiCmd() . . . 128

CHAPTER 13 DEBUG FUNCTIONS . . . 131
13.1 Overview . . . 131
13.2 Types of Debug Output . . . 131
13.3 Debug Output Selection at Compile Time . . . 132
13.4 Debug Output Specificity . . . 132
13.5 Debug Output Selection at Run Time . . . 132
13.6 Portability Concerns . . . 133
13.6.1 Static Variable Coherency . . . 133
13.6.2 VpSysDebugPrintf() . . . 133
13.6.3 ANSI X3.64 Colors . . . 133
13.6.4 Displaying VpDeviceIdType and VpLineIdType . . . 134

APPENDIX A GLOSSARY . . . 135

APPENDIX B FUNCTION INDEX . . . 137

APPENDIX C RELAY CONFIGURATIONS . . . 143

APPENDIX D REVISION HISTORY . . . 147
Rev 14 - 2/09/2011 . . . 147
Rev 15 - 7/15/2011 . . . 147
Rev 16 - 11/15/2011 . . . 147
Rev 17 - 11/13/2012 . . . 147

Rev 18 - 03/20/13 147
Rev 19 - 08/23/13 147
Rev 20 - 06/05/2014 148

CHAPTER

1 INTRODUCTION

1.1 ABOUT THIS USER'S GUIDE

This document describes the Microsemi VoicePath™ Application Program Interface (VP-API-II). The VP-API-II is used to control Microsemi's telephony Voice Termination Devices (VTDs). This chapter highlights the document structure and summarizes the VP-API-II architecture and features.

1.1.1 Chapter Overview

This user's guide consists of the following chapters:

Chapter 2, Profiles: Explains the concept of the VP-API-II profiles.

Chapter 3, Options: Describes the VP-API-II options.

Chapter 4, Design Considerations: Provides some design considerations when using the VP-API-II to avoid potential errors and to make the most efficient use of the VP-API-II.

Chapter 5, System Configuration Functions: Describes the VP-API-II functions used to configure the VP-API-II for the correct device and line type.

Chapter 6, Initialization Functions: Describes VP-API-II functions used to initialize and configure the devices and lines.

Chapter 7, Interrupt and Event Handling: Describes configuration and management of the VP-API-II for one of the four (device) interrupt modes and functions for managing VP-API-II Events.

Chapter 8, Events: Describes the VP-API-II event details.

Chapter 9, Control Functions: Describes the VP-API-II functions used to set the line state, generate line tones, generate Caller ID and other line control operations.

Chapter 10, Status and Query Functions: Describes the VP-API-II functions used to retrieve device and line information and status (e.g., hook, ground key, clock fault, etc.).

Chapter 11, System Services: Describes the user implemented functions to provide OS abstraction and support functions required for (device level) interrupt management.

Chapter 12, Hardware Abstraction Layer: Describes the VP-API-II hardware abstraction layer functions required to provide access to the underlying silicon.

Chapter 13, Debug Functions: Describes debug capabilities of the VP-API-II.

Appendix A, Glossary: Defines uncommon terminology used throughout this document.

Appendix B, Function Index: Provides a summary of all VP-API-II functions.

Appendix C, Relay Configurations: Defines relay configurations for specified termination types.

Appendix D, Revision History: Revision history of this document.

1.1.2 Frequently Used Terms

The following terms are used extensively throughout this document:

Device: The term *device* refers to a Microsemi VTD. Examples of a device include Microsemi's conventional SLAC™ devices (CSLAC) and Voice Control Processor (VCP) devices. The VP-API-II primarily configures and controls a device. A device provides services for one or more channels to perform functions of termination lines.

Channel: The term *channel* refers to resources associated with a device in performing the functions of a termination line. Thus, if a device has resources to support four termination lines, the device is said to have four channels. Note that a channel by itself does not represent all the blocks that are necessary to implement a termination line; it merely represents part of the resources that are provided by a device.

Line: The term *line* means *termination line* in this document. Line refers to the complete system solution (application software, VP-API-II software, Microsemi VTDs, other hardware) that implements an FXS or FXO termination line. The line uses the channel resources of a device in order to realize the features of the line.

Refer to Appendix A, Glossary for the definition of other uncommon terms used in this document.

1.1.3 Documentation Conventions

The *VP-API-II User's Guide* uses the formatting conventions shown in Table 1–1.

Table 1–1 Documentation Conventions

Format	Usage
`Bold Courier New`	Indicates a VP-API-II function or data type.
`Plain Courier New`	Indicates computer code or a file name.
Bold Blue Underlined Text	Indicates a hyper link cross-reference or a web site.
Bold Blue Text	Indicates a hyper link cross-reference.
Italic	Emphasizes an important term.

1.2 VP-API-II OVERVIEW

The VP-API-II library is a C source code module that provides a standard software interface for controlling, testing, and passing digitized voice through a set of subscriber lines using Microsemi Voice Termination Devices. The VP-API-II hides the details of controlling Microsemi VTDs and allows software developers to focus on the application instead of the hardware.

1.2.1 Features

Listed below are some of the key features of the CSLAC VP-API-II library. Note that some features depend on support from the underlying hardware.

- Provides a common software interface.
- OS agnostic (used with any OS and Kernel/User mode)
- Supports dynamic and static memory models.
- Supports various interrupt modes.
- Implemented in portable ANSI C compliant code.
- Seamlessly integrates with the Microsemi Line Test SW (VeriVoice™ Auditor or Professional Line Test Packages).

1.2.1.1 Profiles

Profiles are used by the VP-API-II to configure the device and line(s) to meet the customer specific requirements. They are C strings of `uint8` data generally device specific, so each profile should be applied only to the device type it was created for. All profiles used by the VP-API-II should be created by Microsemi's VoicePath™ Profile Wizard.

The profile types generated by Profile Wizard and functions to which they apply are:

Profile Type	Function	Comment
Profiles used with Initialization Functions		
Device	`VpInitDevice()`	Device Profile is Required in this function. It contains critical device configuration and CSLAC software parameters to maintain timing and operate the `VpApiTick()`.
AC, DC, Ringing, FXO/Dialing	`VpInitDevice()`	Optional. Must be provided prior to putting line into service.
	`VpInitLine()`	Optional. Must be provided prior to putting line into service.
	`VpConfigLine()`	Optional. Must be provided prior to putting line into service.
FXS Control	`VpInitCustomTermType()`	This is used to configure the CODEC I/O Lines to the SLIC State/Det Mapping. Used for VE580 only.
Profile Definitions		
AC	AC transmission and VE890 FXO B-Filter Calibration. AC Profiles are also used with function `VpCal()` for this purpose.	
DC	Feed and Loop Supervision Parameters. For VE890 will also contain the Floor Voltage. On the VE880 device, the Floor Voltage is in the Device Profile.	
Ringing	Ringing configuration (frequency, type, amplitude, DC bias). For VE890 will also contain the Ringing Voltage (hidden). On the VE880 device, the Ringing Voltage is in the Device Profile.	
FXO/Dialing	FXO Line Detection and Generation Parameters	

Profile Type	Function	Comment
Profiles used with Control Functions		
Tone	`VpSetLineTone()`	Contains call progress tone frequency/amplitude.
Tone/Ringing Cadence	`VpSetLineTone()`	Cadence Profiles contain sets of cmd/data operated on by the VP-API-II sequencer. In general, Ringing On/Off is provided to `VpInitRing()` and Tone On/Off is provided to `VpSetLineTone()`, but there are very few restrictions. Providing ringback tone in a Ringing Cadence is possible.
	`VpInitRing()`	
Caller ID	`VpSendCid()`	Similar to Cadence Profiles, generates a wide range of line control sequences including DTMF or FSK tone generation in support of Type I or Type II Caller ID.
	`VpInitRing()`	
Metering	`VpInitMeter()`	Initializes Metering Parameters in the silicon, but does not start Metering on the line. To generate Metering on the line refer to function `VpStartMeter()`.

1.2.1.2 Options

Options are similar to Profiles in they are used to configure device and line parameters. There are two main differences between Options and Profiles:

- Device and Line parameters controlled by Options can be determined at run-time and almost all can be retrieved from the VP-API-II (*).
- Device and Line parameters controlled by Profiles must be pre-determined and only one Profile can be retrieved from the VP-API-II (using `VpCal()`).

Options are set by `VpSetOption()` and retrieved by `VpGetOption()`. See Chapter 3, Options for a list of the device and line options.

1.2.2 Architecture

Figure 1–1 on page 4 illustrates a typical software block diagram of a system incorporating the VP-API-II. The VP-API-II module provides services to the Application Layer and requires services from the System Services Layer and Hardware Abstraction Layer. The following sections describe each of the blocks shown in Figure 1–1.

Figure 1–1 Software Block Diagram

1.2.2.1 VP-API-II

This block is implemented and provided by Microsemi. It is the set of code described in this document. It runs on the host microprocessor controlling one or more Microsemi VTDs. The VP-API-II is provided in two versions:

- VP-API-II ("full"): Supports all operations described in this document.
- VP-API-II Lite: Supports all operations described in this document excluding Tone and Ringing cadence, `VpSendSignal()`, Metering Cadence, and Caller ID. Note that the tone, ringing and metering signals may be applied to the line using the appropriate functions when using VP-API-II Lite, they simply do not cadence (i.e., when using VP-API-II lite all cadencing must be done by the application).

1.2.2.2 Customer Application

This block is implemented by the user. It represents the line management tasks such as initializing the system, configuring lines, changing line states, responding to events, etc.

1.2.2.3 Operating System

This block is implemented by the user. It represents the operating system functions that the user is running on the host microprocessor. The VP-API-II accesses this layer through the functions described in Chapter 11, System Services. Note that the implementation of these functions does not require an OS.

1.2.2.4 Hardware Abstraction Layer

This block is implemented by the user. It provides access to Microsemi devices through the Microprocessor Interface (MPI) for CSLAC devices. The VP-API-II accesses this layer through the functions described in Chapter 12, Hardware Abstraction Layer.

1.2.2.5 System Services Layer

This block is implemented by the user. It abstracts platform-specific functions such as interrupt control and timing services. This layer derives the functions required by the VP-API-II from the facilities provided by the underlying hardware or operating system. The VP-API-II accesses this layer through the functions described in Chapter 11, System Services. Note that the implementation of these functions does not require an OS.

1.2.3 Supported Hardware Configurations

The VP-API-II supports any combination of Microsemi CSLAC devices and line termination types. The specific CSLAC device being used is identified by a `VpDeviceType` and the specific termination type is identified by `VpTermType`. Note that the `VpDeviceType` refers only to the device connected directly to the Host Processor, not to devices that are subtended from other Microsemi devices.

Important characteristics of the CSLAC VP-API-II

- CSLAC silicon contains very little real-time operation and therefore requires an external timer provided by the Application. The Application **MUST CALL** the `VpApiTick()` function at a pre-determined rate (known as the "tick rate") in order for the VP-API-II to operate correctly.
- An interrupt from the CSLAC silicon does not necessarily correspond to a VP-API-II event. The CSLAC VP-API-II provides an extensive amount of silicon to VP-API-II interface processing in order to simplify the application and to provide a common interface.
- CSLAC silicon contains very little internal memory. In order to be memory friendly (i.e., let the Application determine how much memory is needed) the VP-API-II in many cases relies on the Application to maintain the required memory rather than having the VP-API-II copy data from

one location to another. Since CSLAC VP-API-II memory is the same as the host memory, the copy operation would simply be a waste of host processor memory.

Table 1–2 on page 6 lists the device configurations supported by the VP-API-II. The *Part Numbers* column indicates which Microsemi parts or device families apply to each configuration. The *Configuration Name* column lists the terminology used throughout this document to refer to a specific device configuration. The VP-API-II supports simultaneously any combination of the configurations shown in Table 1–2 subject to the limitations of the target hardware.

Table 1–2 Supported CSLAC Device Configurations

VpDeviceType	Part Numbers	Configuration Names
VP_DEV_880_SERIES	Le88xxx	CSLAC-880
VP_DEV_890_SERIES	Le89xxx	CSLAC-890
VP_DEV_790_SERIES	Le79Q224x Le7922x Le79228x	CSLAC-790
VP_DEV_580_SERIES	Le58QLxxx Le58083	CSLAC-580

Each of the device configurations described above supports one or more of the line termination types listed in Table 1–3.

Table 1–3 Supported CSLAC Termination Types

VpTermType	Devices	Description
FXS Termination Types		
VP_TERM_FXS_GENERIC	All	Generic FXS termination
VP_TERM_FXS_ISOLATE	CSLAC-880, CSLAC-890	FXS termination with SLIC driver isolation relay. Unpowered relay connects (customer line) T/R to SLIC.
VP_TERM_FXS_ISOLATE_LP	CSLAC-880, CSLAC-890	FXS termination with SLIC driver isolation relay. Unpowered relay connects (customer line) T/R to SLIC. Uses Low Power technology of the 880 series device.
VP_TERM_FXS_SPLITTER	CSLAC-880, CSLAC-890	FXS termination with SLIC driver isolation relay. Unpowered relay isolates (customer line) T/R from SLIC.
VP_TERM_FXS_75181	CSLAC-790	SLIC Le792x2/ LCAS Le75181; External ringing bus access through the LCAS.
VP_TERM_FXS_75282	CSLAC-790	SLIC Le792x2 / LCAS Le75282; Test-out bus access relay; Test-in, external ringing bus access through the LCAS.
VP_TERM_FXS_TO_TL	CSLAC-790	SLIC Le792x2 with test-out relay and test load
VP_TERM_FXS_CUSTOM	CSLAC-580	Custom FXS termination. Case where the user specifies the SLAC, SLIC, and I/O connections required to set Line States and Detector status.
VP_TERM_FXS_LOW_PWR	CSLAC-880, CSLAC-890	Termination specific to 880/890 architecture implementing low power algorithms.
VP_TERM_FXS_SPLITTER_LP	CSLAC-880, CSLAC-890	Low Power FXS termination with Splitter. Unpowered relay isolates (customer line) T/R from SLIC.

Table 1–3 Supported CSLAC Termination Types

FXO Termination Types		
VP_TERM_FXO_GENERIC	CSLAC-580, CSLAC-880, CSLAC-890	Generic FXO termination
VP_TERM_FXO_DISC	CSLAC-880	FXO termination with additional circuitry for improved Disconnect detection.

CSLAC Device System Architecture

The system architecture for CSLAC products is shown below.

Figure 1–2 VoicePath System Architecture Using CSLAC™ Devices

The VP-API-II supports one or more SLAC devices connected to and controlled by a host processor that uses one or more MPI interfaces. Each line derived from the SLAC device can be configured for FXS or FXO termination provided supporting circuitry is available. The number of lines depends only on the SLAC device used in the design. The VP-API-II permits the combination of more than one type of SLAC device and more than one type of termination in a given design.

Communication to the SLAC is done through the HAL layer which provides the basic MPI access function (**`VpMpiCmd()`**). The function **`VpMpiCmd()`** is passed a `deviceId` (of user defined type `VpDeviceIdType`) initialized by the application to uniquely identify a particular SLAC. Refer to Chapter 12 for details about the HAL function required by the CSLAC devices.

1.2.4 VP-API-II Function Summary

This section provides a brief overview of each of the VP-API-II functions.

1.2.4.1 System Configuration Functions

The VP-API-II uses the concept of *device objects* and *line objects* to manage run-time support for different types of VTDs and line terminations. An instance of a device object represents a physical VTD controlled by the VP-API-II. A device object can represent any type of VTD as long as support for that type of VTD is included in the VP-API-II at compile-time (see `vp_api_cfg.h`).

Device contexts are essentially handles to device objects. There must be exactly one instance of a device object per VTD in the system, but there can be many device contexts referring to a single device object. Similarly, an instance of a line object represents one physical line managed by the VP-API-II. *Line contexts* are basically handles to line objects. There must be exactly one instance of a line object per physical line in the system, but there can be many line contexts referring to a single line object.

The System Configuration functions manage device objects, line objects, device contexts, and line contexts.

- **`VpMakeDeviceObject()`** – Initializes a device object to control a physical device. Also initializes a device context that refers to the new device object. This function is required by all Applications.
- **`VpMakeLineObject()`** – Initializes a line object and associates it with a device object. Also initializes a line context that refers to the new line object. This function is required by all Applications.
- **`VpFreeLineCtx()`** – Tells the API that the application no longer needs a particular line context. This function is required only when dynamic memory methods are used.
- **`VpMakeDeviceCtx()`** – Allows creating more than one device context referring to the same device object. This function is required only in multi-threaded Applications.
- **`VpMakeLineCtx()`** – Allows creating more than one line context referring to the same line object. This function is required only in multi-threaded Applications.
- **`VpMapLineId()`** – Allows application mapping of generic line id to the line. This function is optional.

1.2.4.2 Initialization

These functions initialize and configure the system before a particular feature can be used.

- **`VpInitDevice()`** – Initializes all FXS and FXO lines of a device and applies the specified profiles to those lines.
- **`VpInitLine()`** – Initializes an individual FXS or FXO line and applies the specified profiles to that line.
- **`VpConfigLine()`** – Sets the AC, DC, and Ring Profiles for an individual FXS line.
- **`VpInitCustomTermType()`** – CSLAC-580 only. Configures the SLAC to Line I/O Connections used to set line state and detect line conditions.
- **`VpCal()`** – Calibrates the line to external parameters and provides methods for managing system calibration coefficients.
- **`VpInitRing()`** – Sets the ringing parameters such as the ringing cadence and Caller ID profile for an individual FXS line.
- **`VpInitCid()`** – Prepares a FXS line for a Caller ID ring sequence.
- **`VpInitMeter()`** – Configures the metering signal generator of an individual FXS line.
- **`VpInitProfile()`** – Initializes the device's profile tables.

1.2.4.3 Interrupt and Event Handling

The Interrupt and Event handling functions address management of the device interrupts and VP-API-II events. The functions in this section are provided by Microsemi.

- `VpApiTick()` – Called periodically to provide a timing basis for the VP-API.
- `VpVirtualISR()` – Services CSLAC device interrupts.
- `VpGetEvent()` – Returns events corresponding to a device.
- `VpFlushEvents()` – Flushes all outstanding events.
- `VpGetResults()` – Reads the data associated with an event.

1.2.4.4 Control

The control functions manage the current line state and set options that may change during run-time.

- `VpFreeRun()` – Puts the device/lines into and takes it out of PLL Free Run mode.
- `VpSetLineState()` – Sets a line to the requested state.
- `VpSetRelayState()` – Sets the line relay configuration.
- `VpSetOption()` – Sets various device and line specific options.
- `VpSetLineTone()` – Generates a cadenced call progress tone on a FXS line.
- `VpSetRelGain()` – Sets the relative transmit or receive gain for a line.
- `VpStartMeter()` – Starts metering on a FXS line.
- `VpSendSignal()` – Generates message waiting pulse on FXS lines, or pulse and DTMF digits on FXO lines.
- `VpSendCid()` – Starts a Caller ID sequence on a FXS line without waiting for a ring state change.
- `VpContinueCid()` – Refreshes the Caller ID buffer for a FXS line during message transmission.
- `VpDtmfDigitDetected()` – Reports a DTMF digit detected outside the scope of the VP-API. Used for implementing Type-II Caller ID.
- `VpDeviceIoAccess()` – Controls device input/output pins.

1.2.4.5 Status and Query

These functions get information and events from the VTD.

- `VpGetLoopCond()` – Reads loop conditions for an FXS line and returns parameters such as voltage, current, and resistance.
- `VpGetLineStatus()` – Returns the state of a particular status flag for one line.
- `VpGetDeviceStatus()` – Returns the state of a particular status flag for the first 32 lines of the device.
- `VpGetLineInfo()` – Retrieves line-specific information from a device or line context. This function is optional.
- `VpGetDeviceInfo()` – Retrieves device-specific information from a device or line context. This function is optional.
- `VpGetOption()` – Returns the current setting of an option.
- `VpGetLineState()` – Reads the current line state.

1.2.4.6 System Services

The system support functions are platform-specific and must be implemented for the target host processor.

- `VpSysWait()` – Implements a software delay
- `VpSysEnterCritical() and VpSysExitCritical()` – Blocks/Opens entry into a critical section of VP-API code through some user-defined method.
- `VpSysDisableInt() and VpSysEnableInt()` – Required by the Level-Triggered interrupt mode to disable/re-enable the CSLAC device interrupt in the host microprocessor.
- `VpSysTestInt()` – Required for all interrupt modes except for Simple Polled mode to test the CSLAC device interrupt line for active interrupts.
- `VpSysDtmfDetEnable() and VpSysDtmfDetDisable()` – These functions are used by the VP-API to control a DTMF digit decoding resource that is outside the scope of the VP-API. This is used for Type-II Caller ID implementation.
- `VpSysDebugPrintf()` – Print mechanism used by VP-API-II Debug features.

1.2.4.7 Hardware Abstraction Layer

The Hardware Abstraction Layer (HAL) functions contain the code used to directly access the Microsemi silicon. These functions are platform-specific and must be implemented by the user.

- `VpMpiCmd()` – Implements an MPI transaction.

1.2.5 Basic VP-API-II Data Types

Table 1–4 lists the basic data types used by the VP-API-II. They exist in: vp_api_types.h. file and are platform specific. The user must review/modify this file as needed.

Table 1–4 Basic VP-API-II Data Types

Type	Description
`VpDeviceIdType`	Application-dependent device ID, user defined type.
`VpLineIdType`	Application-dependent line ID, user defined type.
`bool`	Boolean variable assigned `TRUE` (1) or `FALSE` (0).
`uint8`	8-bit unsigned integer.
`uint16`	16-bit unsigned integer.
`uint32`	32-bit unsigned integer.
`int8`	8-bit signed integer.
`int16`	16-bit signed integer.
`int32`	32-bit signed integer.
`uint8p`	Pointer to 8-bit unsigned integer.
`uint16p`	Pointer to 16-bit unsigned integer.
`uint32p`	Pointer to 32-bit unsigned integer.
`VpProfilePtrType`	Pointer to profile data.

1.2.6 VP-API-II Function Return Type

Most VP-API-II functions return a result code indicating whether the function executed successfully, and if not, what type of error occurred. The enumeration type `VpStatusType` is defined for this purpose. All `VpStatusType` codes are listed in the table below.

Table 1–5 VP-API-II Result Codes

Type	Description
`VP_STATUS_SUCCESS`	Function executed successfully.
`VP_STATUS_FAILURE`	Function execution failed due to unspecified error.
`VP_STATUS_FUNC_NOT_SUPPORTED`	Function not supported for the device.
`VP_STATUS_INVALID_ARG`	One or more arguments to the function are invalid. No command is issued to the VTD.
`VP_STATUS_ERR_VTD_CODE`	Unsupported device type or termination type requested in call to **`VpMakeDeviceObject()`**, **`VpMakeLineObject()`**, **`VpMakeDeviceCtx()`**, or **`VpMakeLineCtx()`**.
`VP_STATUS_OPTION_NOT_SUPPORTED`	Unsupported option in call to **`VpSetOption()`** or **`VpGetOption()`**.
`VP_STATUS_DEVICE_BUSY`	Resources required to perform the requested function are not available.
`VP_STATUS_DEV_NOT_INITIALIZED`	The specified device object is not yet initialized via **`VpInitDevice()`**.
`VP_STATUS_ERR_PROFILE`	VP-API-II detected an error in the format of a profile. This error is also returned if the application attempts to use an uninitialized profile from the profile table.
`VP_STATUS_DEDICATED_PINS`	Can be returned when Application is trying to configure or write to I/O pins that are dedicated to the termination type. When this error returns, all pins NOT dedicated to the termination type will be changed. Pins that are dedicated will be unchanged.
`VP_STATUS_INVALID_LINE`	VP-API-II detected that the physical line corresponding to the line context does not exist.
`VP_STATUS_LINE_NOT_CONFIG`	Returned when the requested action requires an AC, DC, or Ringing profile not previously provided.
`VP_STATUS_INPUT_PARAM_OOR`	Returned when one or more of the input values exceeds the device range. The result depends on the function/option that generated this return value.
`VP_STATUS_ERR_SPI`	Returned when **`VpInitDevice()`** detects a communication error.

1.3 API-II SOURCE VERSION NUMBER

An Application may at runtime want to determine the current VP-API-II release version. This can be accomplished by appropriate parsing of the VP-API-II version tag:

```
#define VP_API_VERSION_TAG (0x021500)
```

Note that the specific value in the source may be different than shown above.

The example shown is for Major release 02 (0x02), Minor release 21 (0x15), Revision 0 (0x00).

- Major Number: Interface level compatibility. Always = 02 for the VP-API-II.
- Minor Number: Functional change or significant addition. The following are modifications to the VP-API-II that warrant a "Minor" number increase:
  - Addition of a device or Termination Type.
  - Addition of Options or Events the Application SHOULD use. Addition of Options or Events that are considered as features - not expected to be used by most Applications would only be considered a Revision Number change.
- Backward compatibility is ensured in a Minor Number change, excluding new functionality.
- Revision Number: Bug fix or minor Option or Event addition, or minor change to existing Event (e.g., adding information to `eventData` that was previously ignored). The following are modifications to the VP-API-II that warrant a "Revision" number increase:

  - Bug fix requiring "extensive" changes to the source. If a bug change is made that affects one line (or very few), such changes are released a "Patch" releases.
  - Minor Option or Event change that is not necessary for the Application, but could be useful under certain circumstances.
- Functional compatibility is guaranteed for all Revision Number (and Patch) releases.

1.4 TECHNICAL SUPPORT

For technical support, logon to http://www.microsemi.com/cmpg then follow the links to "Products" and "Comm-Med Product Directory". Then log into your MyCMPG account and follow the links to the "Product Issue Tracker".

CHAPTER

2 PROFILES

2.1 OVERVIEW

Profiles are structures that contain design data to meet specific system requirements. Many VP-API-II functions take profiles as one or more arguments. There are several different types of profiles. Each defines a different set of parameters for a service aspect of the device. Table 2–1 provides a summary of all the profiles that can be used by the VP-API-II. Some profile types are not utilized by certain device types. Also, the content of some profiles may vary according to the device type.

2.2 PROFILE TYPES

Table 2–1 VP-API-II Profile Types

Profile Type	Description
Device Profile	Contains values required to initialize the device.
AC Profile	Used for programming the transmission characteristics of the system, the AC Profile holds the VTD programmable gain and filter coefficients and data. Each AC Profile is designed to address the specific AC transmission requirements of a given design.
DC Profile	Holds the VTD DC feed commands and data. Each DC Profile is designed to address the specific DC feed requirements of a given design.
Ringing Profile	Configures the ringing signal parameters such as waveform type, frequency, amplitude, and DC offset.
Metering Profile	Contains metering signal parameters such as pulse type, current limits, voltage limits and frequency (if 12Khz/16Khz metering signal).
Tone Profile	Specifies tone frequency and level for up to 4 simultaneous tones (number of tones depending on device).
Ringing Cadence Profile	Defines the ringing on/of durations used when ringing the line. Also specifies when CID is generated for Ringing associated CID.
Tone Cadence Profile	Defines the tone generator on/of durations used when sending tones on the line. Note that each tone specified in the tone profile may be turned on/off individually in a single tone cadence.
Caller ID Profile	Defines the signal type and parameters (frequency and level) used when generating CID as well as the signaling protocol used hen performing a CID sequence. The CID profile is used for both Type I and Type II CID.
FXO Configuration Profile	Contains the FXO configuration, including ringing detection frequency window, ringing detection amplitude, and loop open disconnect voltage.
Custom Termination Profile	Used for VE580 Applications only. Contains an I/O mapping of the SLAC to SLIC (or FXO) interface used to set the line state and detect line conditions. Note that a custom termination must be initialized with `VpInitCustomTermType()` prior to initialization of the line.

2.3 PROFILE TABLES

The VP-API-II provides *profile tables* that allow one or more instances of each type of profile to pre-loaded into the system and accessed during normal operation by an index rather than pointer. Figure 2–1 illustrates the concept of profile tables.

Figure 2–1 Profile Tables

Each box in Figure 2–1 represents one instance of a profile. Each stack of boxes represents a table of that specific type of profile. The number of profile table entries available per profile type is specified in Table 2–2.

The application refers to an individual profile within a profile table by passing a *profile table index* into VP-API-II functions. Profile table indices are simply C macros in the form of `VP_PTABLE_INDEXx` where $1 \le x \le 15$. `VP_PTABLE_NULL` is a special value indicating that no profile argument is specified.

The application can load data into a profile table entry at any time after calling **`VpMakeDeviceObject()`** (i.e., before or after device and line initialization). *However, overwriting a profile table entry while that profile is in use could result in unusual behavior.*

Alternatively, the application can bypass the profile tables and load profiles directly into the device by passing a pointer to a profile instead of a profile table index. Any VP-API-II function profile argument that is not a valid profile table index is automatically interpreted as a pointer to a profile in memory.

Table 2–2 Profile Table Capacity

Profile Type	Number of Profiles
Device Profile	1
AC Profile	2
DC Profile	2
Ringing Profile	2
Ringing Cadence Profile	4
Tone Profile	10
Metering Profile	2
Caller ID Profile	2
Tone Cadence Profile	11
FXO Configuration Profile	1

2.4 PROFILE FUNCTIONS

Below is a list of the VP-API-II functions that use profiles. Please refer to the appropriate function descriptions for more information about these functions.

- *VpInitProfile()*, on page 66
- *VpInitDevice()*, on page 56
- *VpInitLine()*, on page 58
- *VpConfigLine()*, on page 59
- *VpCal()*, on page 61
- *VpInitRing()*, on page 64
- *VpInitMeter()*, on page 65
- *VpSetLineTone()*, on page 105
- *VpSendCid()*, on page 109

2.5 IMPORTANT NOTES ABOUT CSLAC AND PROFILES

- As mentioned previously, CSLAC devices contain no memory for storing profiles. Therefore, the application must retain the memory for all profiles while in use except for those profiles provided to `VpInitDevice()` and `VpInitLine()` (e.g., Device, AC-FXS, AC-FXO, DC, Ringing, and FXO/Dialing Profiles). These profiles do not need to be maintained by the application because the VP-API-II **BY DESIGN** saves all relevant profile information (from these profiles only) to either the silicon or to the device/line objects.
- `VpInitProfile()` has limited use in a CSLAC application. It was provided/supported only to be compatible with other device families supported by the VP-API-II. When loading a profile table entry by calling `VpInitProfile()` the VP-API-II simply saves the profile pointer passed into the device object. When the application later calls a function using the profile table entry, the VP-API-II simply refers to the profile pointer stored in the device object. This means all profiles pointed to by the profile table entries **MUST BE** maintained by the application while the profile is being accessed. Given this, it should be just as easy for the application to pass the profile pointer to the function that will use it rather than using the profile tables. The host requirements are the same whether or not the profile table is used, and there is no host processor memory savings by using the profile tables in a CSLAC application.

CHAPTER

3 OPTIONS

3.1 OVERVIEW

This chapter covers the VP-API-II Options. Each option is described using the following format:

DESCRIPTION	This is a summary description of the option.
DEFAULT	This field contains the default setting for the option.
DEVICES	This field lists the devices that support the option.
TERMINATIONS	This field lists the termination types (FXS, FXO, All) that support the option. Termination type "All" means either all termination types supported by the applicable devices, or the termination type is not relevant to the option.

This chapter discusses the individual option types that are accessed through the *VpSetOption()*, on page 104 and *VpGetOption()*, on page 120 functions.

3.2 OPTION SUMMARY

`VpOptionIdType` defines the set of options that `VpSetOption()` can write and `VpGetOption()` can read. Table 3–1 lists all CSLAC supported options for each device and termination type. Each option is described in detail later in this chapter. Note that most options are automatically set to default values by the VP-API-II when the VTD is initialized.

Some options apply to individual lines, while other options apply to an entire VTD and all lines controlled by it. Device option names begin with `VP_DEVICE_OPTION_ID_`, line option names begin with `VP_OPTION_ID_`. The type of option (device-specific or line-specific) combined with the `pLineCtx` and `pDevCtx` arguments determine which line's configuration is accessed

Table 3–1 CSLAC Supported Options

Options	Devices	Terminations	Page
VP_DEVICE_OPTION_ID_CRITICAL_FLT	CSLAC-790, CSLAC-880, CSLAC-890	FXS	19
VP_OPTION_ID_SWITCHER_CTRL	CSLAC-890	FXS	19
VP_OPTION_ID_DEBUG_SELECT	All	All	19
VP_OPTION_ID_PULSE_MODE	All	FXS	20
VP_DEVICE_OPTION_ID_PULSE	All	FXS	20
VP_DEVICE_OPTION_ID_PULSE2	CSLAC-880, CSLAC-890	FXS	22
VP_OPTION_ID_TIMESLOT	All	All	22
VP_OPTION_ID_CODEC	All	All	23
VP_OPTION_ID_PCM_HWY	All	All	25
VP_OPTION_ID_PCM_TXRX_CNTRL	All	All	25
VP_OPTION_ID_RING_CNTRL	All	FXS	26
VP_OPTION_ID_LOOPBACK	All	All	27
VP_OPTION_ID_LINE_STATE	All	FXS	28
VP_OPTION_ID_EVENT_MASK	All	All	29
VP_DEVICE_OPTION_ID_DEVICE_IO	All	All	30
VP_OPTION_ID_ABS_GAIN	CSLAC-880, CSLAC-890	All	31
VP_OPTION_ID_DCFEED_PARAMS	CSLAC-890	FXS	33
VP_OPTION_ID_RINGING_PARAMS	CSLAC-890	FXS	34

3.3 OPTION DESCRIPTIONS

3.3.1 VP_DEVICE_OPTION_ID_CRITICAL_FLT

DESCRIPTION This option determines whether or not a line is automatically forced into `VP_LINE_DISCONNECT` state when a critical fault is detected on that line. This option is device-specific and applies to all lines controlled by the VTD. Critical fault option parameters are passed through the **`VpOptionCriticalFltType`** structure shown below.

```
typedef struct {
    bool acFltDiscEn;
    bool dcFltDiscEn;
    bool thermFltDiscEn;
} VpOptionCriticalFltType;
```

Setting `acFltDiscEn`, `dcFltDiscEn`, or `thermFltDiscEn` to `TRUE` enables automatic disconnect when an AC, DC, or thermal fault is detected, respectively.

Notes:

1. *The CSLAC-880 and CSLAC-890 configurations only support thermal faults*
2. *The CSLAC-890-FXO reports line status using the Event VP_LINE_EVID_DC_FLT, but is not affected by this option.*

DEFAULT `VpOptionCriticalFltType::acFltDiscEn= TRUE;` (`FALSE` for 880 and 890 devices)
`VpOptionCriticalFltType::dcFltDiscEn= TRUE;` (`FALSE` for 880 and 890 devices)
`VpOptionCriticalFltType::thermFltDiscEn= TRUE;`

DEVICES CSLAC-790, CSLAC-880, CSLAC-890

TERMINATIONS FXS

3.3.2 VP_OPTION_ID_SWITCHER_CTRL

DESCRIPTION This option provides control over switcher based designs and is provided a bool value.

```
bool autoShutdownEn;
```

The `autoShutdownEn` parameter determines whether the line will automatically shutdown upon detection of a battery fault as indicated by a `VP_DEV_EVID_BAT_FLT`. When set to `TRUE` a battery fault will cause the affected line to be set to `VP_LINE_DISCONNECT`.

DEFAULT `bool autoShutdownEn = FALSE;`

DEVICES CSLAC-890

TERMINATIONS FXS

3.3.3 VP_OPTION_ID_DEBUG_SELECT

DESCRIPTION This option enables and disables the debug capabilities of the VP-API-II as described in Chapter 13. See Chapter 13 for details.

DEFAULT `uint32 debugSelect = (VP_DBG_WARNING | VP_DBG_ERROR);` (If debug is compiled in)

DEVICES All

TERMINATIONS All

3.3.4 VP_OPTION_ID_PULSE_MODE

DESCRIPTION The pulse mode option determines whether automatic flash and pulse-digit decode is enabled for a particular line. This option is line-specific. This option is passed through a variable of type **VpOptionPulseModeType**, shown below.

```
Enumeration Data Type: VpOptionPulseModeType:
    VP_OPTION_PULSE_DECODE_OFF
    VP_OPTION_PULSE_DECODE_ON
```

DEFAULT VP_OPTION_PULSE_DECODE_OFF

DEVICES All

TERMINATIONS FXS

3.3.5 VP_DEVICE_OPTION_ID_PULSE

DESCRIPTION The option applies only if pulse detection is enabled on the line using VP_OPTION_ID_PULSE_MODE.

The pulse options allow the application to set the timing limits used by the VP-API-II/VTD to decode pulse digits and hook-switch flashes. All of the times are in units of 125us. This option is device-specific and applies to all lines controlled by the VTD. Pulse option parameters are passed through the **VpOptionPulseType** structure shown below.

```
typedef struct {
    uint16 breakMin;                        /* Minimum pulse break time */
    uint16 breakMax;                        /* Maximum pulse break time*/
    uint16 makeMin;                         /* Minimum pulse make time */
    uint16 makeMax;                         /* Maximum pulse make time */
    uint16 interDigitMin;                   /* Minimum pulse interdigit time. */
    uint16 flashMin;                        /* Minimum flash break time */
    uint16 flashMax;                        /* Maximum flash break time */

#ifdef EXTENDED_FLASH_HOOK
    uint16 onHookMin;                       /* Minimum on-Hook time */

#endif
} VpOptionPulseType;
```

The timing limits set by the application must conform to the following relationships:

1. breakMin < breakMax < flashMin < flashMax < onHookMin
2. makeMin < makeMax < interDigitMin

Figure 3–1 shows an example of a typical setting for this option. It also shows the timing relationship between the dialed digit and the events generated.

Notes:

1. *Parameter* onHookMin *is compiled out of the VP-API-II by default. To enable* onHookMin*, the value* EXTENDED_FLASH_HOOK *must be changed to* #define *in file vp_api_cfg.h.*

Figure 3–1 Typical VP Option Pulse Timing Diagrams

On-Hook Event is generated if "break" is longer than *onHookMin* time

DEFAULT

```
VpOptionPulseType::breakMin      = 33 * 8;      /* 33 milliseconds */
VpOptionPulseType::breakMax      = 100 * 8;     /* 100ms */
VpOptionPulseType::makeMin       = 17 * 8;      /* 17ms */
VpOptionPulseType::makeMax       = 75 * 8;      /* 75ms */
VpOptionPulseType::interDigitMin = 250 * 8;     /* 250ms */
VpOptionPulseType::flashMin      = 250 * 8;     /* 250ms */
VpOptionPulseType::flashMax      = 1300 * 8;    /* 1300ms */
VpOptionPulseType::onHookMin     = 1300 * 8;    /* 1300ms */
```

DEVICES All

TERMINATIONS FXS

3.3.6 VP_DEVICE_OPTION_ID_PULSE2

DESCRIPTION

This option behaves identically to **VP_DEVICE_OPTION_ID_PULSE** and provides a second mask to be used for dial pulse detection. See **VP_DEVICE_OPTION_ID_PULSE** for input argument details.

This option is intended to provide the ability to uniquely detect dial pulses with different set of breakMin/breakMax and makeMin/makeMax requirements from those set using **VP_DEVICE_OPTION_ID_PULSE**. A case where this could be used is for NTT Edition 5 when detecting 10pps and 20pps. When using this option however, the values interDigitMin, flashMin/flashMax, and onHookMin must be identical to those set in **VP_DEVICE_OPTION_ID_PULSE**. Only the breakMin/breakMax and makeMin/makeMax values can be different.

When performing dial pulse detection, the VP-API-II evaluates the parameters of **VP_DEVICE_OPTION_ID_PULSE** before evaluating the parameters of **VP_DEVICE_OPTION_ID_PULSE2**. Therefore, when only one set of dial pulse parameters is required the application must use **VP_DEVICE_OPTION_ID_PULSE** and set the values in **VP_DEVICE_OPTION_ID_PULSE2** to 0. Using the reverse settings (i.e., setting parameters in **VP_DEVICE_OPTION_ID_PULSE** to 0 and setting parameters in **VP_DEVICE_OPTION_ID_PULSE2** to the desired settings) will causes unpredictable events to be generated.

DEFAULT

```
VpOptionPulseType::breakMin      = 0;          /* 0ms */
VpOptionPulseType::breakMax      = 0;          /* 0ms */
VpOptionPulseType::makeMin       = 0;          /* 0ms */
VpOptionPulseType::makeMax       = 0;          /* 0ms */
VpOptionPulseType::interDigitMin = 0;          /* 0ms */
VpOptionPulseType::flashMin      = 0;          /* 0ms */
VpOptionPulseType::flashMax      = 0;          /* 0ms */
VpOptionPulseType::onHookMin     = 0;          /* 0ms */
```

DEVICES CSLAC-880, CSLAC-890

TERMINATIONS FXS

3.3.7 VP_OPTION_ID_TIMESLOT

DESCRIPTION

The timeslot option selects the PCM transmit and receive timeslots for the given line. PCM timeslots are numbered from 0 to *max_num_timeslots*-1, where *max_num_timeslots* equals f_{PCLK} KHz / 8 KHz / 8 bits. Setting this option with either tx or rx timeslots that exceed (*max_num_timeslots*-1) will result in return VP_STATUS_INPUT_PARAM_OOR from **VpSetOption()**.

Timeslot option parameters are passed through the **VpOptionTimeslotType** structure shown below.

```
typedef struct {
    uint8 tx;     /* 8-bit timeslot used for data in the A-D direction */
    uint8 rx;     /* 8-bit timeslot used for data in the D-A direction */
} VpOptionTimeslotType;
```

The number of timeslots used depends on the CODEC mode (see **VP_OPTION_ID_CODEC**). In all cases **VP_OPTION_ID_TIMESLOT** option defines the first timeslots used by the channel.

Notes:

1. The application should assign timeslots before setting the line to a state that will enable transmission on the PCM highway. *See* *VpSetLineState(), on page 101* *for more information on which line states activate the PCM highway.*

DEFAULT None

DEVICES All

TERMINATIONS All

3.3.8 VP_OPTION_ID_CODEC

DESCRIPTION

The codec option selects the PCM encoding algorithm for the given line. This option is line-specific. This option is passed through a variable of type **VpOptionCodecType**, shown below.

```
Enumeration Data Type: VpOptionCodecType:
    VP_OPTION_ALAW              /* G.711 A-law PCM encoding 8kHz sample rate */
    VP_OPTION_MLAW              /* G.711 Mu-law PCM encoding 8kHz sample rate */
    VP_OPTION_LINEAR            /* Linear 16-bit encoding 8kHz sample rate */
    VP_OPTION_LINEAR_WIDEBAND   /* Linear 16-bit encoding 16kHz sample rate */
    VP_OPTION_ALAW_WIDEBAND     /* G.711 A-law PCM encoding 16kHz sample rate */
    VP_OPTION_MLAW_WIDEBAND     /* G.711 Mu-law PCM encoding 16kHz sample rate */
```

Wideband Application Note:

`VP_OPTION_LINEAR_WIDEBAND` *is supported for CSLAC-880 and CSLAC-890 devices only, and operates at the device level. Meaning both lines must operate in Wideband mode at the same time, or Narrowband mode at the same time. This is a silicon restriction enforced in software.*

In order to minimize the impact this has on the Application, the VP-API-II will manage Wideband to/from Narrowband changes as follows:

Rule: The CODEC Mode applied to both channels when changing to/from Wideband mode will be the CODEC mode applied to the most recent channel changed. Example below:

Changing Wideband/Narrowband Mode:

If the application changes the `CODEC` *mode of one line to* `VP_OPTION_LINEAR_WIDEBAND` *mode the VP-API-II will automatically change the other line (if exists) also to* `VP_OPTION_LINEAR_WIDEBAND`*. Calling* **`VpGetOption()`** *with* `VP_OPTION_ID_CODEC`*) for either line at this point will return* `VP_OPTION_LINEAR_WIDEBAND`*.*

If the application changes one of the lines to (e.g.) `VP_OPTION_ALAW`*, the VP-API-II will automatically change the other line (if exists) to* `VP_OPTION_ALAW`*. At this point, the lines are (once again) independent and can be programmed to different CODEC modes.*

The changes to/from `VP_OPTION_LINEAR_WIDEBAND` *on the other line occur inside* **`VpApiTick()`**, *so excluding the call to* **`VpApiTick()`** *the following describes the VP-API-II behavior for* `VP_OPTION_LINEAR_WIDEBAND` *mode:*

Given: *"pLineCtx1" = a line context pointer for channel 1, "pLineCtx" = a line context pointer for channel 2 both on the same device pointed to by "pDevCtx"*

```
VpOptionCodecType codecMode = VP_OPTION_ALAW;

/* Set line 1 to A-Law, line 2 to u-Law */
VpSetOption(pLineCtx1, VP_NULL, VP_OPTION_ID_CODEC, &codecMode);
codecMode = VP_OPTION_MLAW;
VpSetOption(pLineCtx1, VP_NULL, VP_OPTION_ID_CODEC, &codecMode);

/* Now Set line 1 to Wideband */
codecMode = VP_OPTION_LINEAR_WIDEBAND;
VpSetOption(pLineCtx1, VP_NULL, VP_OPTION_ID_CODEC, &codecMode);

/* Line 1 is in Wideband now, but Line 2 is not yet in Wideband mode. At
   least one call to VpApiTick() is required to set Line 2. However, even
   though the call to VpApiTick() puts Line 2 in Wideband mode, it is highly
   recommended the Application use VpSetOption() also to force Line 2 to
   Wideband mode. */
VpApiTick(pDevCtx, pEvent);
VpSetOption(pLineCtx2, VP_NULL, VP_OPTION_ID_CODEC, &codecMode);

/* Line 2 is now in VP_OPTION_LINEAR_WIDEBAND mode, as well as Line 1 */

/* Put line 1 back to A-Law and follow the same sequence mentioned above
   (except in this case we're taking the lines out of Wideband Mode. Note that
   it is NOT necessary to wait for next VpApiTick() when changing Wideband
   states. The following is perfectly legal. */
codecMode = VP_OPTION_ALAW;
VpSetOption(pLineCtx1, VP_NULL, VP_OPTION_ID_CODEC, &codecMode);
VpSetOption(pLineCtx2, VP_NULL, VP_OPTION_ID_CODEC, &codecMode);
                      -------- OR --------
VpSetOption(VP_NULL, pDevCtx, VP_OPTION_ID_CODEC, &codecMode);
/* CODEC Mode is set on both lines already. Nothing for VpApiTick() to do */
VpApiTick(pDevCtx, pEvent);
```

Timeslot Management For CODEC Modes:

The timeslots consumed by the line will depend on the values provided in **`VP_OPTION_ID_TIMESLOT`** *and the* `CODEC` *mode. For all* `CODEC` *settings the values provided with* **`VP_OPTION_ID_TIMESLOT`** *refer to the lowest/starting timeslot number.*

`VP_OPTION_ALAW` *and* `VP_OPTION_MLAW`*: When using 8-bit compressed mode only a single timeslot is required.*

`VP_OPTION_LINEAR`*: When using 16-bit Linear mode two consecutive timeslots are required.*

`VP_OPTION_LINEAR_WIDEBAND`*: When using 16-bit Linear Wideband Mode (sampled at 16kHz - twice the frame rate), 4 timeslots are required. The first two correspond to the first Linear data sample in the frame, the second two correspond to the second Linear data sample in the frame. Since the sample rate is 2x the frame rate, the samples are separated by 1/2 a frame.*

For a PCLK = 8.192MHz and timeslots of 0 (set by **`VP_OPTION_ID_TIMESLOT`***) the figure below shows the timeslots consumed for the different* `CODEC` *modes:*

DEFAULT `VP_OPTION_ALAW`

DEVICES All

TERMINATIONS All

3.3.9 VP_OPTION_ID_PCM_HWY

DESCRIPTION The PCM highway option selects the PCM highway for the given line. This option is line-specific. This option is passed through a variable of type **`VpOptionPcmHwyType`**, shown below.

```
Enumeration Data Type: VpOptionPcmHwyType:
    VP_OPTION_HWY_A    /* Select the "A" PCM Highway */
    VP_OPTION_HWY_B    /* Select the "B" PCM Highway - VE790 and VE580 only */
```

Notes:

1. *CSLAC-880 and CSLAC-890 devices support* `VP_OPTION_HWY_A` *option only.*

DEFAULT `VP_OPTION_HWY_A`

DEVICES All

TERMINATIONS All

3.3.10 VP_OPTION_ID_PCM_TXRX_CNTRL

DESCRIPTION This line-specific option enables or disables the PCM transmit and receive paths in line states that use the PCM highway, including: `VP_LINE_TALK`, `VP_LINE_TALK_POLREV`, `VP_LINE_OHT`, and `VP_LINE_OHT_POLREV`. This option can take any of the following values:

```
Enumeration Data Type: VpOptionPcmTxRxCntrlType:
    VP_OPTION_PCM_BOTH          /* Enable PCM transmit and receive paths */
    VP_OPTION_PCM_RX_ONLY       /* Enable PCM receive path only */
    VP_OPTION_PCM_TX_ONLY       /* Enable PCM transmit path only */
    VP_OPTION_PCM_ALWAYS_ON     /* PCM Path enabled for all line states */
    VP_OPTION_PCM_OFF           /* PCM Path disabled for all line states */
```

Notes:

1. *`VP_OPTION_PCM_ALWAYS_ON` and `VP_OPTION_PCM_OFF` supported by 880 and 890 devices only.*

DEFAULT `VP_OPTION_PCM_BOTH`

DEVICES All

TERMINATIONS All

3.3.11 VP_OPTION_ID_RING_CNTRL

DESCRIPTION

This option configures the ring-trip attributes of a line. This option is line-specific. The ring-trip control option is passed though the **VpOptionRingControlType** structure shown below.

```
typedef struct {
    VpOptionZeroCrossType zeroCross;
    uint16 ringExitDbncDur;
    VpLineStateType ringTripExitSt;
} VpOptionRingControlType;
```

The zeroCross parameter controls the relay behavior used to exit ringing near the zero cross point. Possible values are:

```
Enumeration Data Type: VpOptionZeroCrossType:
    VP_OPTION_ZC_M4B        /* Zero-Cross control - make-before-break */
    VP_OPTION_ZC_B4M        /* Zero-Cross control - break-before-make */
    VP_OPTION_ZC_NONE       /* Turn Zero-Cross Control OFF */
```

Make-before-Break provides a continuous feed to the line, Break-before-Make provides a slight disconnect duration while exiting ringing, and None removes ringing immediately (i.e., does not wait for zero crossing).

VpOptionZeroCrossType Notes:

1. *The VTD relay setting must be in VP_RELAY_NORMAL for the VTD to control either variation of the Zero-Cross Control Option. If the relay setting is not set to VP_RELAY_NORMAL, then the proper timing and control will not occur (no operation of the relay will occur at all) and any operation pertaining to the relay control for ring entry or exit will be the host's responsibility.*
2. *The VP_OPTION_ZC_NONE zero-cross option is not supported for the CSLAC-790 class of devices.*

The ringExitDbncDur variable sets the minimum time to ignore hook activity when exiting ringing, specified in units of 125us. The ring-exit debounce period starts when the line state is changed by the application not (necessarily) when ringing is removed from the line, which may be up to 1/2 period from the time the application stops ringing.

This debounce time helps filter false hook events during transitions from the *ringing-on* state to the *ringing-off* state caused by the physical characteristics of the line.

The ringTripExitSt parameter determines which state the line is automatically switched to when ring-trip occurs. Auto-Ring Trip can be effectively disabled by setting ringTripExitSt to the active ringing state (i.e. VP_LINE_RINGING or VP_LINE_RINGING_POLREV).

A line that is detecting loop close or ground key at the time of being set to VP_LINE_RINGING or VP_LINE_RINGING_POLREV will instead be immediately set to the ringTripExitSt.

Notes:
The default ringTripExitSt is VP_LINE_TALK which by default will enable the PCM Highway. This is desirable as long as the Application has control over the PCM Data to the far end while the call is being connected (e.g., via a DSP). If on the other hand, the call connection requires some amount of time, ringTripExitSt should be changed to VP_LINE_ACTIVE and the line should only be set to VP_LINE_TALK until AFTER call connection has been established and Ringing Removed from Tip-Ring (depends on ringing frequency in the Ringing Profile).

DEFAULT

```
VpOptionRingControlType::zeroCross = VP_OPTION_ZC_M4B;
VpOptionRingControlType::ringExitDbncDur = 100 * 8; /* 100ms */
VpOptionRingControlType::ringTripExitSt = VP_LINE_TALK;
```

DEVICES All

TERMINATIONS FXS

3.3.12 VP_OPTION_ID_LOOPBACK

DESCRIPTION

The loopback option controls the loop back mode of the given line. This option is line-specific. This option is passed through a variable of type **VpOptionLoopbackType**, shown below.

```
Enumeration Data Type: VpOptionLoopbackType:
    VP_OPTION_LB_OFF            /* All loopbacks off */
    VP_OPTION_LB_TIMESLOT       /* Perform a timeslot loopback */
    VP_OPTION_LB_DIGITAL        /* Perform a full-digital loopback */
    VP_OPTION_LB_CHANNELS       /* Perform a timeslot loopback */
```

VP_OPTION_LB_OFF will remove all loopbacks. Note however this has no impact on the Line State or Timeslot settings.

VP_OPTION_LB_TIMESLOT loopbacks the line at the device internal timeslot manager. Digital signals output from the TX path will match those provided in the RX path. Signals provided in the Receive PCM Highway will not be sent to the customer's line in Timeslot Loopback mode.

VP_OPTION_LB_DIGITAL loopbacks the line in the A/D-D/A section. Digital signals output from the TX path will not match those provided in the RX path. Signals provided in the Receive PCM Highway will be sent to the customer's line in Digital Loopback mode.

VP_OPTION_LB_CHANNELS is a timeslot connection between the FXO and FXS line of the device. This selection is useful for connecting the PSTN (on the FXO line) to the FXS line without imposing delay caused by upstream processing. This selection is supported on CSLAC-890 device only that have both an FXS and an FXO channel.

Notes:

1. *The CSLAC-880 and CSLAC-890 configurations do not support* VP_OPTION_LB_DIGITAL *in the VP-API-II However, the equivalent loopback for 880/890 devices is supported in the Line Test Software (Software Packages: Le880SLVVP and Le890SLVVP - VeriVoice Professional).*
2. *When performing a loopback test (i.e., set* VP_OPTION_LB_TIMESLOT *mode and send data into the PCM receive path while monitoring the PCM transmit path) the line must be in a state where the A/D and D/A converters are enabled. Refer to* **VpSetLineState()** for states that meet these conditions.

DEFAULT VP_OPTION_LB_OFF

DEVICES All

TERMINATIONS All

3.3.13 VP_OPTION_ID_LINE_STATE

DESCRIPTION

This line specific option controls battery levels used during normal off-hook states and defines the polarity reversal method (abrupt or smooth) used when changing feed polarity. These settings are passed through the **`VpOptionLineStateType`** structure shown below.

```
typedef struct {
    bool battRev;                /* Smooth/abrupt Bat reversal; TRUE=abrupt */
    VpOptionBatType bat;         /* Battery selection for active line states */
} VpOptionLineStateType;
```

When the `battRev` variable is set to `FALSE`, it forces a smooth voltage change when the line state is changed from any feed state to any other feed state of opposite polarity. The smooth ramp time varies with the device and is not programmable.

The `bat` battery modifier option can be any of the following enumeration constants:

```
Enumeration Data Type: VpOptionBatType:
    VP_OPTION_BAT_AUTO           /* Automatic battery selection */
    VP_OPTION_BAT_HIGH           /* Use high and pos batt for active line states */
    VP_OPTION_BAT_LOW            /* Use low battery for active line states */
    VP_OPTION_BAT_BOOST          /* Utilize positive batt for active states */
```

Notes:

1. *The* `battRev` *(i.e., smooth) battery reversal is used to minimize noise on the line.*
2. *CSLAC-880 and CSLAC-890 devices only support* `VP_OPTION_BAT_AUTO`*.*

DEFAULT

```
VpOptionLineStateType::battRev = FALSE;
VpOptionLineStateType::bat = VP_OPTION_BAT_AUTO;
```

DEVICES All

TERMINATIONS FXS

3.3.14 VP_OPTION_ID_EVENT_MASK

DESCRIPTION

This option determines which events are reported for a given line. This option is line-specific. The event mask option is passed though the **VpOptionEventMaskType** structure shown below.

```
typedef struct {
    uint16 faults;              /* Fault event category masks */
    uint16 signaling;           /* Signaling event category masks */
    uint16 response;            /* Mailbox response event category masks */
    uint16 test;                /* Test events */
    uint16 process;             /* Call process event category masks */
    uint16 fxo;                 /* FXO event category masks */
} VpOptionEventMaskType;
```

Masking an event is done by setting the corresponding event bit to '1', unmasking is done by setting the corresponding event bit to'0'. Masking an event prevents the event from being reported to the application (via **VpApiTick()** and **VpGetEvent()**).

The composite masks for each event category are created by *bitwise OR'ing* the event ID constants of the individual events within that category. When building the composite event mask for a particular event category the application should only sum individual event ID constants belonging to that event category. The following enumeration types define the event ID constants in the source code:

- **VpFaultEventType**
- **VpSignalingEventType**
- **VpResponseEventType**
- **VpTestEventType**
- **VpProcessEventType**
- **VpFxoEventType**

These event types are described in Chapter 8. Some event are device-specific, and some events are line-specific. Setting a *device-specific* event mask for an individual *line* effectively changes that mask for all other lines controlled by that VTD. To avoid confusion, the application should always set device-specific event masks for individual lines to the same value across all lines of a VTD. The application must take care not to accidentally change a device-specific event mask when modifying other event masks for an individual line. Refer to *VpSetOption(), on page 104* for more information on how device-specific and line-specific options are applied based on the values of `pDevCtx` and `pLineCtx`.

To support future expansion of the event categories, the application should set all bits in an instantiated event structure to '1' prior to setting the individual bits to user defined values. This can be done using a standard memset function as follows:

```
VpOptionEventMaskType eventMask;
memset(&eventMask, 0xFF, sizeof(VpOptionEventMaskType));
```

Notes:

1. *Event masking takes affect immediately. Unmasked events that occur will not be reported if masked prior to calling* **VpApiTick()**. *Event unmasking however behaves differently. Some events that occur while masked will be reported if they are unmasked prior to calling* **VpApiTick()**, *some events will not. This occurs for events that when masked result in silicon level interrupt masking. In those cases, the VP-API-II has no record that the event has occurred when it is unmasked.*
2. *Some events are non-maskable. The mask bits corresponding to these events are ignored. Refer to Chapter 8 for details.*

DEFAULT

All events are masked except for non-maskable events.

DEVICES

All

TERMINATIONS

All

3.3.15 VP_DEVICE_OPTION_ID_DEVICE_IO

DESCRIPTION

This option controls the device-specific input/output (I/O) pin configuration. The number of I/O pins configurable through this option is dictated by the reference design (VTD and termination type) being used. Only those I/O pins that are not dedicated per the reference design for driving termination type specific devices (i.e., LCAS, relay, etc..) can be controlled by this option.

The I/O configuration option is passed though the **`VpOptionDeviceIoType`** structure shown below.

```
typedef struct {
    uint32 directionPins_31_0;
    uint32 directionPins_63_32;
    uint32 outputTypePins_31_0;
    uint32 outputTypePins_63_32;
} VpOptionDeviceIoType;
```

The `directionPins_63_32` and `directionPins_31_0` variables are combined to make a single 64-bit `direction` field, where each bit determines whether an individual pin is an input (0) or output (1). For a configuration that supports only one user I/O pin per channel, `direction[N]` sets the direction for the I/O pin belonging to channel `N`. For a configuration that supports two user I/O pins per channel, `direction[2N]` sets the direction for I/O Pin 0 belonging to channel `N`, and `direction[2N+1]` sets the direction for I/O Pin 1 belonging to channel `N`. Unused `direction` bits that do not map to a channel/pin are ignored. The `outputTypePins_63_32` and `outputTypePins_31_0` variables are mapped in a similar fashion.

Each bit in `outputType` determines whether a single pin is configured as a TTL/CMOS output (`VP_OUTPUT_DRIVEN_PIN = 0`) or open-collector/open-drain (`VP_OUTPUT_OPEN_PIN` = 1) output.

Example 1: 2-channel device each with 2-I/O pins. Setting channel 1, I/O1 = Output-Driven, Channel 2, I/O2 = Output-Open, all other pins set to Input:

```
directionPins_31_0 = 9;    1 0 0 1
                           │ │ │ └ ch1, I/01 = Output
                           │ │ └── ch1, I/02 = Input
                           │ └──── ch2, I/01 = Input
                           └────── ch2, I/02 = Output

outputTypePins_31_0 = 8;   1 x x 0
                           │ │ │ └ ch1, I/01 = Driven
                           │ │ └── ch1, I/02 = Input (output type ignored)
                           │ └──── ch2, I/01 = Input (output type ignored)
                           └────── ch2, I/02 = Open

directionPins_63_32 = x;
outputTypePins_63_32 = x;   nnn_63_32 ignored because none of the
                            CSLAC silicon has enough I/O lines to
                            extend past 31 (32 total)
```

Example 2: 2-channel device each with 1-I/O pin. Setting channel 1, I/O1 = Output-Driven, Channel 2, I/O1 = Input:

```
directionPins_31_0 = 1; 0 1
                        │ └ ch1, I/01 = Output
                        └── ch2, I/01 = Input
outputTypePins_31_0 = 0; x 0
                         │ └ ch1, I/01 = Driven
                         └── ch2, I/01 = Input (output type ignored)
```

DEFAULT None, VTDs retain their hardware reset value.

DEVICES All

TERMINATIONS All

3.3.16 VP_OPTION_ID_ABS_GAIN

DESCRIPTION

This option is used to set the absolute gain of the transmission path in the A-to-D and D-to-A directions. Proper use of this function requires coefficients provided by Microsemi.

Absolute Gain is specified in 0.1dB steps using the following **VpOptionAbsGainType** structure:

```
typedef struct {
    int16 gain_AToD;     /* Gain in A-to-D Direction in 0.1dB steps */
    int16 gain_DToA;     /* Gain in D-to-A Direction in 0.1dB steps */
} VpOptionAbsGainType;
```

Special Values supported for gain_AToD and gain_DToA are described in the following table.

Name	Value	Description
VP_OPTION_ABS_GAIN_NO_CHANGE	32767	Leave current gain setting as-is.
VP_OPTION_ABS_GAIN_QUIET	-32767	Mute the transmission path selected. (*)
VP_OPTION_ABS_GAIN_RESTORE	-32768	Restore gain from Quiet Mode. (**)
VP_ABS_GAIN_UNKNOWN (output only)	32766	Generated by VP-API-II until ABS_GAIN option is set.

(*) Disabling the transmission path with VP_OPTION_ID_ABS_GAIN_QUIET has a different affect than using the **cutoff** methods of **VpSetLineState()** and **VP_OPTION_ID_PCM_TXRX_CNTRL**:

- **VpSetLineState()**: Voice path is removed from the PCM highway resulting in the PCM Highway being driven only by external pull-ups ("all ones").
- VP_OPTION_ABS_GAIN_QUIET: Voice path will drive the PCM highway with the quiet code corresponding to the CODEC mode selected.

(**) VP_OPTION_ABS_GAIN_RESTORE can be ONLY be used when last preceded by setting the line to quiet using VP_OPTION_ABS_GAIN_QUIET, If any other write to the gain block occurs between the quiet setting and restore operation, which includes calls to **VpSetRelGain()** and **VpConfigLine()**, the restore operation using VP_OPTION_ABS_GAIN_RESTORE will be ignored.

```
Preliminary - Initialize TX/RX Gain (0/-6dB):
{    /* main.c or other initialization function */
     VpOptionAbsGainType gain = {
         .gain_AToD = 0, .gain_DToA = -60
     };
     /* Set initial: TX Gain = 0dB, RX Gain = -6dB */
     VpSetOption(pLineCtx, NULL, VP_OPTION_ID_ABS_GAIN, &gain);
}
{     /* Middle of Call Processing. Mute Receive Path Only. */
     VpOptionAbsGainType gain = {
         .gain_AToD = VP_OPTION_ABS_GAIN_NO_CHANGE, /* Leave TX alone */
         .gain_DToA = VP_OPTION_ABS_GAIN_QUIET     /* Mute RX (D/A) Path */
     };
     VpSetOption(pLineCtx, NULL, VP_OPTION_ID_ABS_GAIN, &gain);
}

Good Scenario
VpConfigLine(pLineCtx, NULL, DC_PROF, NULL); /* OK: AC Profile = NULL */
{    /* IF Restore Here - OK! Gain NOT Changed since setting to Quiet */
     VpOptionAbsGainType gain = {
         .gain_AToD = VP_OPTION_ABS_GAIN_NO_CHANGE, /* Leave TX alone */
         .gain_DToA = VP_OPTION_ABS_GAIN_RESTORE    /* Restore RX Gain */
     };
     VpSetOption(pLineCtx, NULL, VP_OPTION_ID_ABS_GAIN, &gain);
     /* In this case, Gain is restored to previous value (-6dB) */
}

Bad Scenario
VpConfigLine(pLineCtx, AC_600, NULL, NULL); /* BAD: AC Profile != NULL */
{     /* IF Restore Here - BAD! Gain Changed since setting to Quiet */
     /* Try to restore does not work since AC Profile Changed */
     VpOptionAbsGainType gain = {
         .gain_AToD = VP_OPTION_ABS_GAIN_NO_CHANGE, /* Leave TX alone */
         .gain_DToA = VP_OPTION_GAIN_RESTORE        /* Restore Gain */
     };
     VpSetOption(pLineCtx, NULL, VP_OPTION_ID_ABS_GAIN, &gain);
     /* In this case, restore is ignored because gain block was modified
      * since last set to quiet. RX gain determined by AC_600 Profile */
}
```

Supported Ranges and Step Sizes

The Supported Ranges and Step Sizes for each Termination Type/direction is shown in Table 3–2.

Table 3–2 `VP_OPTION_ID_ABS_GAIN` Supported Ranges and Step Sizes

Termination Type	A-to-D	D-to-A	Step Size
FXS	-6dB to +6dB	-12dB to 0dB	0.5dB
FXO	0dB to +12dB	-9dB to +3dB	0.5dB

All input values in the range of Table 3–2 (+/-5 = +/-0.5dB) will be rounded to the nearest 0.5dB step. Regardless of the result from rounding, the VP-API-II in this scenario will always make an adjustment.

- If the result of the rounding is within the specified range, the VP-API-II makes the adjustment and returns `VP_STATUS_SUCCESS`.
- If the result of the rounding is outside of the specified range, the VP-API-II makes the adjustment and returns `VP_STATUS_INPUT_PARAM_OOR`.

Using `VpGetOption()`

Retrieving the current gain (i.e., calling **`VpGetOption()`** with this `optionId`) will return the current gain if the gain block has been set using this `optionId` and NOT modified by any other VP-API-II operation since (e.g., **`VpSetRelGain()`**, **`VpInitDevice()`**, **`VpInitLine()`** or **`VpConfigLine()`**). However, if any of these other VP-API-II functions are called since setting the gain (i.e., calling **`VpSetOption()`** with this `optionId`), then retrieving the current gain using this `optionId` will return `VP_ABS_GAIN_UNKNOWN`. Walk-through example shown:

`Preliminary - Initialize TX/RX Gain (0/-6dB):`

```
{   /* main.c or other initialization function */
    VpOptionAbsGainType gain = {
        .gain_AToD = 0, .gain_DToA = -60
    };
    /* Set initial: TX Gain = 0dB, RX Gain = -6dB */
    VpSetOption(pLineCtx, NULL, VP_OPTION_ID_ABS_GAIN, &gain);
}
```

`Good Scenario`

```
VpConfigLine(pLineCtx, NULL, DC_PROF, NULL); /* OK: AC Profile = NULL */
VpGetOptionImmediate(pLineCtx, NULL, VP_OPTION_ID_ABS_GAIN, &gain);
```

`gain.gain_AToD = 0, gain.gain_DToA = -60:` gain blocks have NOT been modified since last being set with this `optionId`.

`Bad Scenario`

```
VpConfigLine(pLineCtx, AC_600, NULL, NULL); /* BAD: AC Profile != NULL */
VpGetOptionImmediate(pLineCtx, NULL, VP_OPTION_ID_ABS_GAIN, &gain);
```

`gain.gain_AToD = VP_ABS_GAIN_UNKNOWN, gain.gain_DToA = VP_ABS_GAIN_UNKNOWN:` gain blocks have been modified (with **`AC_600`** Profile) since last being set with this `optionId`.

Example `gain_AToD` and `gain_DToA` values:

Target Gain	Value	Target Gain	Value
-2.0dB	-20	No Change	`VP_OPTION_ABS_GAIN_NO_CHANGE`
-3.5dB	-35	Mute	`VP_OPTION_ABS_GAIN_QUIET`
+3.0dB	30	Restore (from Mute)	`VP_OPTION_ABS_GAIN_RESTORE`

Notes:

1. *This option modifies the reference level used by* **`VpSetRelGain()`**. *If* **`VpSetRelGain()`** *is called before this option, the relative level will be based on the last AC profile provided. If* **`VpSetRelGain()`** *is called after this option, the relative level will be based on the last value provided with this option.*

DEFAULT `VP_ABS_GAIN_UNKNOWN`

DEVICES CSLAC-880, CSLAC-890

TERMINATIONS All

3.3.17 VP_OPTION_ID_DCFEED_PARAMS

DESCRIPTION

This option is used to modify some of the DC and Loop Supervision Parameters of the line that were previously set by a DC profile. This option is not intended to replace the DC Profile, but rather to provide a convenient way to modify parameters in the DC Profile that can take on a wide range of values (e.g., ILA, VOC). Without this option, Applications today have to create and modify profiles on-the-fly to adjust these parameters because creating individual profiles for all of the possible settings is impractical.

The DC Feed Parameters are passed through the **`VpOptionDcFeedParamsType`** struct shown below (parameters which apply only for other device types are omitted):

```
/* Option value struct for VP_OPTION_ID_DCFEED_PARAMS: */

typedef struct VpOptionDcFeedParamsType {
    uint32 validMask;        /* Bit-Mask of DC Parameters to Modify */
    int32 voc;               /* VOC in millivolts (mV) */
    int32 ila;               /* ILA in microamps (µA) */
    int32 hookThreshold;     /* Hook Detect Threshold in microamps (µA) */
    int32 gkeyThreshold;     /* Ground Key Detect Threshold (µA) */
    int32 battFloor;        /* DC Feed Floor Voltage (mV) - Tracker Only */
} VpOptionDcFeedParamsType;
```

Parameter `validMask` is a bit-mask set by the Application telling the VP-API-II which parameters in the **`VpOptionDcFeedParamsType`** `struct` to program. Fields corresponding to bits left as '0' will not be changed. The valid bit-masks for `validMask` are:

`validMask` bit Name	**Field Affected**
VP_OPTION_CFG_VOC	voc
VP_OPTION_CFG_ILA	ila
VP_OPTION_CFG_HOOK_THRESHOLD	hookThreshold
VP_OPTION_CFG_GKEY_THRESHOLD	gkeyThreshold
VP_OPTION_CFG_BATT_FLOOR	battFloor

When using **`VpGetOption()`**, the `validMask` result tells which fields are valid for the particular line type. Some fields do not apply for all device types.

DEFAULT

Default values are set by the DC profile.

DEVICES

CSLAC-890

TERMINATIONS

FXS

3.3.18 VP_OPTION_ID_RINGING_PARAMS

DESCRIPTION

This option is used to modify some of the main Ringing Parameters of the line that were set by the last Ringing profile. This option is not intended to replace the Ringing Profile, but rather to provide a convenient way to modify certain Ringing Parameters that of the line that are generally changed during run-time.

One restriction is the Ringing Type (Sine or Trapezoidal) must be set by providing a Ringing profile with the desired Type to the line before using this option. For ex:

```
/* Provide Base Ringing Profiles */
const VpProfileData RING_SINE[] = {.sinewave values..};
const VpProfileData RING_TRAP[] = {.sinewave values..};

/* Initialize with Sinewave */
VpInitDevice(pDevCtx, devProf, acProf, dcProf, RING_SINE, VP_NULL,
VP_NULL);

/* Bad Example - Modify Trapezoidal rise time on Sinewave Ringing Type! */
VpOptionRingingParamsType ringParam = {
    .validMask = VP_OPTION_CFG_TRAP_RISE_TIME,
    .trapRiseTime = 8000
};
VpSetOption(pLineCtx, VP_NULL, VP_OPTION_ID_RINGING_PARAMS, &ringParam);

/* Good Example - Change Ringing Type before changing rise time. */
VpOptionRingingParamsType ringParam = {
    .validMask = VP_OPTION_CFG_TRAP_RISE_TIME,
    .trapRiseTime = 8000
};
VpConfigLine(pLineCtx, VP_NULL, VP_NULL, RING_TRAP); /* Set to
Trapezoidal */
VpSetOption(pLineCtx, VP_NULL, VP_OPTION_ID_RINGING_PARAMS, &ringParam);
```

Ringing Parameters are specified using **VpOptionRingingParamsType** struct shown below:

```
typedef struct VpOptionRingingParamsType {
   uint32 validMask;        /* Bit-Mask of Ringing Parameters to Modify */
    int32 frequency;        /* Frequency in millihertz (mHz) */
    int32 amplitude;        /* Peak Amplitude in millivolts (mV) */
    int32 dcBias;           /* DC Bias in millivolts (mV) */
    int32 ringTripThreshold;/* Ring Trip Threshold in microamps (µA) */
   int32 ringCurrentLimit;  /* Ringing Current Limit in microamps (µA) */
   int32 trapRiseTime;  /* (Trapezoidal) Rise Time in microseconds (µs) */
} VpOptionRingingParamsType;
```

`validMask` is a bit-mask set by the Application telling the VP-API-II which parameters in the **VpOptionRingingParamsType** `struct` to program. Fields corresponding to bits left as '0' will not be changed. The valid bit-masks for `validMask` are:

validMask bit Name	**Field Affected**
VP_OPTION_CFG_FREQUENCY	frequency
VP_OPTION_CFG_AMPLITUDE	amplitude
VP_OPTION_CFG_DC_BIAS	dcBias
VP_OPTION_CFG_RINGTRIP_THRESHOLD	ringTripThreshold
VP_OPTION_CFG_RING_CURRENT_LIMIT	ringCurrentLimit
VP_OPTION_CFG_TRAP_RISE_TIME	trapRiseTime

When using **VpGetOption()**, the `validMask` result tells which fields are valid for the particular base profile type (e.g. `VP_OPTION_CFG_TRAP_RISE_TIME` will not be valid if the base ringing profile was not trapezoidal).

DEFAULT Default values are set by the ringing profile.

DEVICES CSLAC-890

TERMINATIONS FXS

CHAPTER

4 DESIGN CONSIDERATIONS

This section describes some of the design considerations to be taken into account when using the VP-API-II. It is intended to help the user maximize efficiency and minimize the potential for unexpected behavior.

4.1 RETURN CODE HANDLING

The VP-API-II is designed to allow a single application to be used for any `VpDeviceType` and `VpTermType` provided the application takes into account the device and line capabilities. This means that every VP-API-II function with all possible arguments may be called for any `VpDeviceType`/`VpTermType` without negatively affecting the physical device/line. There are three general results from calling any VP-API-II function:

1. If the operation can be performed **exactly** as requested, the VP-API-II returns `VP_STATUS_SUCCESS` and the device and lines are affected.
2. If the operation cannot be performed **exactly** as requested, the VP-API-II returns an error code and except for function **`VpDeviceIoAccess()`** the device/lines are not affected.

   Notes:

   In case of calling **`VpSetOption()`** *with `optionId = VP_OPTION_ID_ABS_GAIN`, a specific gain value can be provided, but a different gain value applied. This occurs for values that exceed the allowable min/max range of the device, resulting in a setting that is nearest the specified value. In this case,* **`VpSetOption()`** *will return `VP_STATUS_SUCCESS` since it is the specified behavior of the option.*

3. In case of **`VpDeviceIoAccess()`**, if the I/O lines being modified by the application include those dedicated to the termination type the VP-API-II returns `VP_STATUS_DEDICATED_PINS`. In this case, I/O lines NOT dedicated to the termination type are affected while I/O lines dedicated to the termination type are unaffected (i.e., application request is ignored for the purposes of the dedicated I/O lines).

4.2 INITIALIZATION OF THE VP-API-II

As of VP-API-II version P2.16.0, the VP-API-II supports "Basic Initialization" (cold start/restart) and an "Fast Initialization" (warm restart, or pre-load start). The "Basic Initialization" is same as it has been in the past, described below in Section 4.2.1 for completeness. "Calibration Bypass" (for fast initialization) occurs in one of a few different ways discussed in Section 4.2.2.

4.2.1 Basic Initialization of the VP-API-II

While each application will have it's own set of requirements (when device and line initialization will occur, when to configure ringing parameters, multi-single threaded application, etc.) the following steps can be followed for all applications:

1. Call **`VpMakeDeviceObject()`**
2. Call **`VpMakeLineObject()`** /* Repeat for each line associated with the device */
3. Call **`VpInitDevice()`**
4. Start calling **`VpApiTick()`** /* Note: **`VpApiTick()`** may be started before or after **`VpInitDevice()`**, but must be running in order to generate the Init device complete event */
5. Wait for Init Device complete event.
6. Unmask all events that will be handled by the application. Note that most events are unmasked by default. See **`VpSetOption()`** and `VP_OPTION_ID_EVENT_MASK` for details.

7. (optional) Check for FEMF or Hazardous Potential on the line. This requires use of Microsemi Line Test SW.
8. (optional) Set relay state to `VP_RELAY_NORMAL` (for termination types that contain relays and if `VP_RELAY_NORMAL` is not the default state).
9. Set all lines to operational states:
   a) FXS lines generally to `VP_LINE_STANDBY` or `VP_LINE_OHT`
   b) FXO lines generally to `VP_LINE_FXO_LOOP_OPEN` or `VP_LINE_FXO_OHT`
10. Call **`VpCalLine()`** for each line /* For devices that require calibration */
11. Wait for Calibration complete event on each line.
12. Proceed with normal configuration and call control.

If calibration is required, the initialization time due to steps #3 - #5, and #10 - #11 can be several seconds. These steps can be avoided using the "Calibration Bypass" method described in the next section.

4.2.2 Calibration Bypass

"Calibration Bypass" refers to the initialization sequence in which the VP-API-II will bypass the device/line calibration procedures performed in **`VpInitDevice()`** and **`VpCalLine()`**. The two basic methods are:

1. "Warm Reboot" detection by the VP-API-II such that the device object retains calibration data throughout the initialization sequence.
2. Pre-Load of Calibration coefficients by the application prior to **`VpInitDevice()`**. This method requires use of function **`VpCal()`** with new `calTypes [VP_CAL_GET_SYSTEM_COEFF and VP_CAL_APPLY_SYSTEM_COEFF]`.

Method 1: "Warm Reboot"

Remember from Section 1.2.4.1 that **`VpMakeDeviceObject()`** and **`VpMakeLineObject()`** initialize the device and line objects respectively. These objects are then filled in and modified by the VP-API-II during run-time with calibration and other relevant data.

During **`VpInitDevice()`** the VP-API-II checks the calibration content of the device object and makes a determination as to whether the content is valid from a previously running condition or reset from **`VpMakeDeviceObject()`**. If the VP-API-II detects that the device object has not been reset, it automatically skips the calibration processes of **`VpInitDevice()`** and **`VpCalLine()`** and applies the previously stored calibration data. The application may continue to call these functions but the VP-API-II will simply return and immediately generate the corresponding event.

To take advantage of the "Warm Reboot" method, the SW architecture should be such that **`VpMakeDeviceObject()`** and **`VpMakeLineObject()`** are called only in case of complete system initialization and not when the system performs a "warm-reboot".

Notes:

*The VP-API-II only checks the device object for previously initialized data and if determined valid, assumes the line objects are also valid. Therefore the application **must not** call* **`VpMakeLineObject()`** *in a warm-reboot scenario.*

Method 2: VpCal() with VP_CAL_GET/APPLY_SYSTEM_COEFF

VpCalType value VP_CAL_GET_SYSTEM_COEFF allows retrieval of the calibration coefficients from the VP-API-II.

VpCalType value VP_CAL_APPLY_SYSTEM_COEFF allows application of calibration coefficients to the VP-API-II. When provided with calibration coefficients per VpCal() using VP_CAL_APPLY_SYSTEM_COEFF the VP-API-II will skip the calibration procedures in all future calls to both VpInitDevice() and VpCalLine() (until call to VpMakeDeviceObject() which clears the calibration data).

Value VP_CAL_GET_SYSTEM_COEFF can be used either in a factory calibration environment or in the field (i.e., application run-time). In either case, the VP-API-II generates the required calibration coefficients per the following process:

Retrieving Calibration Coefficients from the VP-API-II

1. Call VpMakeDeviceObject()
2. Call VpMakeLineObject() /* Repeat for each line associated with the device */
3. Call VpInitDevice()
4. Start calling VpApiTick() /* Note: VpApiTick() may be started before or after VpInitDevice(), but must be running in order to generate the Init device complete event */
5. Wait for Init Device complete event.
6. Unmask all events that will be handled by the application. Note that most events are unmasked by default. See VpSetOption() and VP_OPTION_ID_EVENT_MASK for details.
7. (optional) Set relay state to VP_RELAY_NORMAL (for termination types that contain relays and if VP_RELAY_NORMAL is not the default state).
8. Set all lines to normal state:
   a) FXS lines generally to VP_LINE_STANDBY or VP_LINE_OHT
   b) FXO lines generally to VP_LINE_FXO_LOOP_OPEN or VP_LINE_FXO_OHT
9. Call VpCalLine() for each line /* For devices that require calibration */
10. Wait for Calibration complete event on each line.
11. Call VpCal() with calType = VP_CAL_GET_SYSTEM_COEFF (inputArgs = VP_NULL). Provide only one of the line contexts associated with each device in the system.
12. Wait for Calibration complete event.

    Notes:

    The event structure (e.g., VpEventType calEvent) element calEvent.hasResults = TRUE for event VP_EVID_CAL_CMP when generated as a result of VpCal() with VP_CAL_GET_SYSTEM_COEFF. This indicates that the application must call VpGetResults() with an "appropriate" pointer, which in case of VP_CAL_GET_SYSTEM_COEFF is a pointer to uint8 buffer of size calEvent.eventData.

13. Call VpGetResults() with a pointer to a uint8 buffer of size >= *calEvent.eventData*. The buffer will be filled with calibration coefficients required by the device and all lines of the device to be used at run-time. Refer to this buffer as sysCalData[*calEvent.eventData*].

Application of Calibration Coefficients ("Fast Initialization"):

1. Call VpMakeDeviceObject()
2. Call VpMakeLineObject() /* Repeat for each line associated with the device */
3. Call VpCal() with calType = VP_CAL_APPLY_SYSTEM_COEFF and inputArgs = sysCalData.

   Notes:

   Calling VpCal() with calType = VP_CAL_GET_SYSTEM_COEFF prior to calling VpInitDevice() will NOT result in event VP_EVID_CAL_CMP being generated. Events can only be generated after VpInitDevice() is called and returns VP_STATUS_SUCCESS.

4. Call `VpInitDevice()`
5. Start calling `VpApiTick()` /* Note: `VpApiTick()` may be started before or after `VpInitDevice()`, but must be running in order to generate the Init device complete event */
6. Wait for Init Device complete event (which will be almost immediate since the system has been pre-loaded with calibration coefficients, thus avoiding the device calibration procedure).
7. Unmask all events that will be handled by the application. Note that most events are unmasked by default. See `VpSetOption()` and `VP_OPTION_ID_EVENT_MASK` for details.
8. (optional) Check for FEMF or Hazardous Potential on the line. Requires use of Microsemi Line Test SW.
9. (optional) Set relay state to `VP_RELAY_NORMAL` (for termination types that contain relays and if `VP_RELAY_NORMAL` is not the default state).
10. Set all lines to operational states:
    a) FXS lines generally to `VP_LINE_STANDBY` or `VP_LINE_OHT`
    b) FXO lines generally to `VP_LINE_FXO_LOOP_OPEN` or `VP_LINE_FXO_OHT`
11. Proceed with normal configuration and call control.

4.3 TICKRATE REQUIREMENTS

The "tickRate" is the interval at which `VpApiTick()` is called. The VP-API-II performance for certain features depends on the tickrate, silicon and CODEC mode. The following sections discuss the tickrate requirements for each VP-API-II provided feature.

4.3.1 Dial Pulse Detection

CSLAC silicon falls into two categories as it concerns Dial Pulse Detection: those that contain an internal hook bit buffer, and those that do not contain an internal hook bit buffer. Table 4–1 lists the silicon both with and without the internal hook bit buffer.

Table 4–1 Internal Hook Bit Buffer

Device	Contains Hook Bit Buffer	Does Not Contain Hook Bit Buffer
VE880	Le88266, Le88276, Le88286, Le88506, Le88264	Le88010, Le881xx, Le8821x, Le8822x, Le8823x, Le8824x, Le883xx
VE890	Le891xx, Le893xx	Le89010
VE790	None	All
VE580	None	All

Silicon that does not have the internal hook bit buffer provides dial pulse detection to the resolution of the tickrate. So the timestamp reported (and used for internal dial pulse detection purposes) has an error of <= *tickRate*.

Silicon that has the internal hook bit buffer is utilized by the VP-API-II to "just fill" the buffer at each tick interval. The sample rate of the buffer depends on a device setting and CODEC mode corresponding to sample rates of [8KHz, 4KHz, 2KHz, 1KHz and 500Hz] for non-WideBand mode, and [16KHz, 8KHz, 4KHz, 2KHz and 1KHz] for WideBand mode. So for example: running a 5ms tickRate in non-WideBand mode means the buffer sample rate will be set to 1KHz for a 1ms hook resolution.

Notes:

Using a 10ms tickrate requires the buffer sample rate to be set to 500Hz, which is below what can be supported in WideBand mode. So if WideBand mode and Dial Pulse Detection are required, the application should use a 5ms tickRate.

One more consideration regarding Dial Pulse Detection: The parameters provided to the VP-API-II option `VP_DEV_OPTION_ID_PULSE` and `VP_DEV_OPTION_ID_PULSE2` must take into account the measurement accuracy of the system (tickrate, hook bit buffer, and WideBand mode included), otherwise digits will not be reported as expected.

4.3.2 FSK Caller ID

There are four categories regarding FSK Caller ID support as outlined in Table 4–2:

Table 4–2 Caller ID Capabilities

Device Family	Silicon Part Number	CID Capability
VE880	Le88010	FXO only. Does not support CID generation.
	Le881xx, Le8821x, Le8822x, Le8823x, Le8824x, Le883xx	1 1/2 byte CID buffer. Requires tickrate <= 8.33ms.
	Le88266, Le88276, Le88286, Le88506, Le88264	2 1/2 byte CID buffer. Requires tickrate <= 16.66ms.
VE890	Le89010	FXO only. Does not support CID generation.
	Le891xx, Le893xx	2 1/2 byte CID buffer
VE790	All	6-bit buffer. Requires 5ms tickrate.
VE580	All	Does not support CID generation.

Preliminary: CID FSK data runs at 1200 baud for a 1 character transmission time of 8.33ms (8-bit character = 8 bits + 1 start and 1 stop bit = 10/1200Hz = 8.33ms).

Given the FSK rate of 1200, the VE790 device requires a precise 5ms tickRate in order to keep up with the 6-bit buffer (6 * 1/1200Hz = 5ms). CID message data will be corrupted with a slower or a (non-integer divider of 5ms) faster tick rate.

The VE880 and VE890 devices on the other hand would not corrupt CID FSK data in case of sub-optimal tickrate. Running a tickrate slower than that specified in Table 4–2 for these devices will cause the device to underrun, resulting in an "all ones" signal being generated (until the next tick). For some CPE devices, short durations of "all ones" between message data will be tolerated (even if not desirable).

Notes:

CID FSK generation is not affected by the CODEC mode (WideBand vs. non-WideBand) as is the case with Dial Pulse Detection.

4.4 CADENCE TIMING

Cadence profiles generated by the Profile Wizard are run in the VP-API-II at the tickrate with one step performed per tick. This is done to maintain consistent processor load independent of the cadence profile, but must be taken into account if precise cadence timing is required.

For example (given a 10ms tick rate to generate a Ringing 2on/4off cadence):

- Line State = `VP_LINE_RINGING` /* Occurs at t=0 */
- Time Delay = 1980ms /* (+10ms) Starts at t=10ms */
- Line State = `VP_LINE_OHT` /* (+10ms + 1980ms) Occurs at t = 2000ms */
- Time Delay = 3970ms /* (+10ms) Starts at t = 2010ms */
- Branch to 0, Loop Forever /* (+10ms + 3970ms) Occurs at t = 6000ms */

4.5 SOFT PROFILE TABLES

The profile tables described in Section 2.3 for CSLAC devices are simulated in software (i.e., "soft" profile tables). These *soft profiles tables* are stored in memory allocated by the application software and referenced by the VP-API-II as necessary. The VP-API-II *does not* store a copy of the profiles internally; it only retains pointers to the profiles. The application software should take extra care not to delete or modify a soft profile as long as the VP-API-II may be using it. This restriction applies to both soft profile tables and any soft profile that is passed to the VP-API-II by reference (using a profile pointer instead of a profile table index). Figure 4–1 on page 40 illustrates the concept of soft profiles.

Figure 4–1 Soft Profile Table Example

This example shows a hypothetical tone profile table setup with the VP-API-II controlling four telephone lines through a CSLAC device. Tone profile table Index 1 (`VP_PTABLE_INDEX1`) references the application's Tone Profile A, and both Lines 1 and 2 are using tone profile table Index 1. Line 3 is accessing application Tone Profile Z via tone profile table Index 3. Line 4 is also using application Tone Profile Z but through tone profile table Index 4 instead of 3.

In this example, application Tone Profile A should not be modified because Lines 1 and 2 are linked to that profile. Application Tone Profile B may be modified because the VP-API-II has no knowledge of that profile. Application Tone Profile C may also be modified since no lines are referencing tone profile table Index 2. Finally, application Tone Profile Z should not be modified because Lines 3 and 4 are using that profile.

4.6 MULTI-TASKING APPLICATIONS

It may be desirable to have multiple tasks controlling various aspects of the voice path. One common example is one handling call control while another managing line testing. In this example, both tasks must share access to the VP-API-II.

Implementations containing multiple tasks that utilize the VP-API-II have additional requirements and constraints. This section describes aspects of the VP-API-II designed to handle this special case. This section does not apply to implementations not employing multiple tasks using the VP-API-II.

4.6.1 Multiple Device and Line Contexts

Recall that the device and line objects contain state information pertaining to the associated device or line. There must be exactly one device object for each VTD in the system and exactly one line object for each line in the system, regardless of the number of tasks. However, each task needs its own context for each line and device it controls.

Figure 4–2 Multi-Tasking Example

By default, the `VpMakeDeviceObject()` and `VpMakeLineObject()` functions create the object and one context associated with the new object. When the VP-API-II is employed by multiple tasks, one task is responsible for creating the necessary objects. All tasks that use the VP-API-II must create contexts and associate them with the single object instance using `VpMakeDeviceCtx()` and `VpMakeLineCtx()` described later in this section. Thus, each task's contexts are unique *handles* to global objects.

Note that only one task should call `VpGetEvent()` and `VpGetResults()`. If multiple tasks need to receive VP-API-II events, a centralized *event dispatcher* task should be implemented to call `VpGetEvent()` and `VpGetResults()` and forward the events to the desired tasks. Also, for CSLAC devices, only one task should perform the VP-API-II tick function.

4.6.2 Protected Memory Requirements

In multi-tasking environments with memory protection, a shared memory region is necessary to share the device and line objects between many tasks. In the example depicted in Figure 4–3 on page 42, a shared memory region is shown with two tasks (A and B) both needing access to the VP-API-II. One task is responsible for creating the shared memory region, and creating the desired device and line objects. All tasks must create device and line contexts for use with VP-API-II function calls.

For systems using "soft" profiles (see *Soft Profile Tables*, on page 40), it may be necessary to place the profile table in the shared memory region as well. This shared memory must be accessible to all processes that access the profile table and also to processes that handle `VpGetEvent()` and `VpApiTick()`.

Figure 4–3 Protected Memory Example

4.7 CODE SIZE MANAGEMENT

The VP-API-II is designed to allow all device types (`VpDeviceType`) and line configurations (`VpTermType`) to reside in the same application and allow the application to call all functions with all possible arguments. While this approach allows application developers to write a single application for all Microsemi Voice products, it tends to create a larger set of code than necessary for the specific application.

Basic code size is managed by compiling in the devices used (e.g., `VP_CC_880_SERIES`, `VP_CC_890_SERIES`, and so on..) in **vp_api_cfg.h** file. But in cases where certain application level choices have been made further reductions are possible.

The VE880 and VE890 families provide a wide variety of features depending on the application and silicon. The following table and sections describe the compile time options available to help optimize code size for VE880 and VE890 applications (note: all values in file **vp_api_cfg.h**):

Table 4–3 Compile Time Options (Code Size Management Only)

Value	Comment	Default
Common		
`VP_REDUCED_API_IF`	If `#define` removes functions from the VP-API-II Interface.	`#undef`
`VP_DEBUG`	If `#define` compiles in run-time debug strings used by the VP-API-II. Setting this to `#define` significantly increases the size of the compiled VP-API-II and not recommended for code space limited applications. It is generally used for debug purposes only.	`#undef`
`VP_CSLAC_SEQ_EN`	If `#undef`, reduces functionality of the VP-API-II.	`Section 4.7.1.2`
VE880 and VE890		
`VP_CSLAC_RUNTIME_CAL_ENABLED`	Run-time calibration supported.	`#define`
`VP_CSLAC_HIGH_GAIN_SUPPORTED`	Set to `#define` to support line states `VP_LINE_HOWLER` and `VP_LINE_HOWLER_POLREV`.	`#define`
VE880 Only		
`VP880_FXS_SUPPORT`	VE880 FXS supported.	`#define`
`VP880_ABS_SUPPORT`	VE880 ABS silicon supported.	`#define`
`VP880_TRACKER_SUPPORT`	VE880 Tracker silicon supported.	`#define`
`VP880_LP_SUPPORT`	VE880 FXS Low Power supported.	`#define`
`VP880_FXO_SUPPORT`	VE880 FXO supported.	`#define`
`VP880_CLARE_RINGING_DETECT`	If `#define`, I/O4 from VE880 device connected to Clare Ringing Detect Output used for FXO line detection.	`#define`
VE890 Only		
`VP890_FXO_SUPPORT`	VE890 FXO supported.	`#define`
`VP890_FXS_SUPPORT`	VE890 FXS supported.	`#define`
`VP890_LP_SUPPORT`	VE890 FXS Low Power supported.	`#define`

Notes:

Compile option `VP_DEBUG` has a significant affect on code size, recommended set to `#undef` for any application limited in code space. Setting `VP_DEBUG` to `#undef` removes all debug facilities and helps minimize code

size. Selective debug mechanisms may be compiled in (rather than all) by setting `VP_DEBUG` to `#define` and properly defining `VP_CC_DEBUG_SELECT`. See Chapter 13 for debug details.

4.7.1 Common Code Size Reductions

4.7.1.1 VP_REDUCED_API_IF

When set to `#define`, functions and options are removed if not supported given other compile time settings. Functions removed are those that could only return `VP_STATUS_FUNC_NOT_SUPPORTED` if called by the application. Options removed are those that when called with **`VpSetOption()`** / **`VpGetOption()`** could only return `VP_STATUS_INVALID_ARG` or `VP_STATUS_OPTION_NOT_SUPPORTED`.

In cases where a function is compiled out of the VP-API-II due to this setting, applications containing these functions will result in compiler errors.

4.7.1.2 VP_CSLAC_SEQ_EN

The default setting for this value depends on the SW Package:

- OPN Le71SK0002 default setting: `#define`
- OPN Le71SDKAPIL default setting: `#undef`

Setting this to `#undef` removes support for cadence operations (i.e., sequencer) on both FXO and FXS termination types. Functional impact as follows:

- **`VpSetLineTone()`** - Function will accept tone and tone cadence profiles, but the cadence profile will be ignored. All tone cadence timing must be provided by the application.
- **`VpSendSignal()`** - Not supported.
- Caller ID (all related functions, including **`VpDtmfDigitDetected()`**) - Not supported.
- **`VpInitRing()`** - Not supported. All ring cadence related timing must be provided by the application.
- **`VpStartMeter()`** - Supports only "infinite on" (`onTime = 0, numMeters != 0`) and "off" (`numMeters = 0`). All metering related timing must be provided by the application.

As a result of removing these functions from the VP-API-II, none of the events in the Process category will be generated. They are:

- `VP_LINE_EVID_MTR_CMP, VP_LINE_EVID_MTR_ABORT, VP_LINE_EVID_CID_DATA, VP_LINE_EVID_RING_CAD, VP_LINE_EVID_SIGNAL_CMP, VP_LINE_EVID_TONE_CAD`

4.7.1.3 VP_CSLAC_RUNTIME_CAL_ENABLED (VE880 and VE890 only)

Setting this compile time value to `#undef` removes all source that supports run-time calibration including **`VpCalLine()`** and calibration associated with **`VpInitDevice()`**. Since the silicon may still require calibration, the calibration supported by **`VpCal()`** (i.e., FXO B-Filter, and Apply/Get System Calibration) are not removed.

When using silicon that requires calibration, the application must either enable run-time calibration or use **`VpCal()`** with `calType = VP_CAL_APPLY_SYSTEM_COEFF`.

In addition to the code size reduction, removing run-time calibration has the benefit of significantly faster initialization time and minimal line voltage dropout duration. Applications need only to have previously calibrated the system in a factory environment using the run-time calibration functions and retrieved the calibration values using **`VpCal()`** with `calType = VP_CAL_GET_SYSTEM_CEOFF` or other Manufacturing Test software.

4.7.2 VE880 Code Size Reductions

The following compile options can be used when `VP_CC_880_SERIES` is set to `#define`.

4.7.2.1 VP880_FXS_SUPPORT

When set to `#undef`, removes Product Code support for the following silicon that supports FXS termination types only:

- All ABS silicon (see `VP880_ABS_SUPPORT`)
- All Tracker Only silicon (see `VP880_TRACKER_SUPPORT`). Tracker silicon that also support FXO termination types can still be initialized.

Initialization against silicon that supports only FXS termination types is the same as `#undef VP880_ABS_SUPPORT` and `#undef VP880_TRACKER_SUPPORT`.

The following functions, options and events are affected with VE880 FXS support disabled:

- **`VpMakeLineObject()`** - Returns `VP_STATUS_VTD_ERR_CODE` for any FXS termination type provided. The line context to device context and line object association is not made.
- **`VpInitDevice()`** - Will bypass battery (device level) calibration. VE880 switchers are used to control Ringing and FXS Feed voltages, which will be disabled (i.e., never enabled in the initialization procedure).
- FXS only functions removed: **`VpInitRing()`**, **`VpInitCid()`**, **`VpInitMeter()`**, **`VpSetRelayState()`**, **`VpStartMeter()`**, **`VpSendCid()`**, **`VpContinueCid()`**, **`VpFreeRun()`**, **`VpDtmfDigitDetected()`**.
  - Calling these functions result in return code `VP_STATUS_FUNC_NOT_SUPPORTED`.
  - Note that **`VpSetRelayState()`** is not FXS specific, simply that in current VP-API-II releases only FXS lines can have relay configurations designed into the termination type.
  - Note that **`VpFreeRun()`** is not FXS specific, but it's intended purpose is to maintain relatively constant switcher performance in presence of clock fault. Since switchers are disabled with FXS support removed, there's no need to call **`VpFreeRun()`** in an FXO only application.
- FXS only configuration options/arguments removed:

  `VpSetOption()`/**`VpGetOption()`**:

  Calling **`VpSetOption()`**/**`VpGetOption()`** with any of the following options results in return code of `VP_STATUS_OPTION_NOT_SUPPORTED`:

  - `VP_DEVICE_OPTION_ID_CRITICAL_FLT`, `VP_DEVICE_OPTION_ID_PULSE`, `VP_DEVICE_OPTION_ID_PULSE2`, `VP_OPTION_ID_PULSE_MODE`, `VP_OPTION_ID_ZERO_CROSS`, `VP_OPTION_ID_RING_CNTRL`, `VP_OPTION_ID_LINE_STATE`, `VP_OPTION_ID_SWITCHER_CTRL`.

  `VP_OPTION_ID_EVENT_MASK` (events that cannot be unmasked, cannot be generated):

  - Fault Category: `VP_DEV_EVID_BAT_FLT`, `VP_LINE_EVID_THERM_FLT`, `VP_LINE_EVID_RES_LEAK_FLT`
  - Signaling Category: `VP_LINE_EVID_HOOK_OFF`, `VP_LINE_EVID_HOOK_ON`, `VP_LINE_EVID_GKEY_DET`, `VP_LINE_EVID_GKEY_REL`, `VP_LINE_EVID_FLASH`, `VP_LINE_EVID_STARTPULSE`, `VP_LINE_EVID_EXTD_FLASH`, `VP_LINE_EVID_DTMF_DIG`, `VP_LINE_EVID_PULSE_DIG`
  - Response Category: `VP_LINE_EVID_CAL_CMP`, `VP_LINE_EVID_CAL_BUSY`
  - Process Category: `VP_LINE_EVID_MTR_CMP`, `VP_LINE_EVID_MTR_ABORT`, `VP_LINE_EVID_CID_DATA`, `VP_LINE_EVID_RING_CAD`

Notes:

VE880 FXS Support requires either VE880 ABS or Tracker Support (`VP880_ABS_SUPPORT` or `VP880_TRACKER_SUPPORT` set to `#define`), otherwise an error will be generated at compile time.

4.7.2.2 VP880_ABS_SUPPORT

When set to `#undef`, removes support for the following (ABS) silicon Product Codes:

- Le88221, Le88226, Le88241, Le88246, Le88266, Le88286, Le88264

`VpInitDevice()` will return `VP_STATUS_FAILURE` if one of these devices are detected during the initialization procedure. As a result, the device object will be flagged as "not initialized".

4.7.2.3 VP880_TRACKER_SUPPORT

When set to `#undef`, removes support for the following (Tracker) silicon Product Codes:

- Le88111, Le88116, Le88131, Le88136, Le88211, Le88216, Le88231, Le88236, Le88506 (VE8820 chipset).

`VpInitDevice()` will return `VP_STATUS_FAILURE` if one of these devices are detected during the initialization procedure. As a result, the device object will be flagged as "not initialized".

4.7.2.4 VP880_LP_SUPPORT

When `VP880_FXS_SUPPORT` is enabled, support for the Low Power termination types can be enabled or disabled in the application. The Low Power termination types are:

- `VP_TERM_FXS_LOW_PWR, VP_TERM_FXS_ISOLATE_LP, VP_TERM_FXS_SPLITTER_LP`.

The following functions and events are affected with VE880 Low Power support disabled:

- **`VpMakeLineObject()`** - Returns `VP_STATUS_VTD_ERR_CODE` for any Low Power termination type provided. The line context to device context and line object association is not made.
- Event `VP_LINE_EVID_RES_LEAK_FLT` is unmaskable and cannot be generated.

Notes:

Enabling VE880 Low Power Support requires both `VP880_FXS_SUPPORT` and `VP880_TRACKER_SUPPORT` set to `#define`, otherwise an error will be generated at compile time.

4.7.2.5 VP880_FXO_SUPPORT

When set to `#undef`, removes support for the following FXO only silicon Product Codes:

- Le88010

`VpInitDevice()` will return `VP_STATUS_FAILURE` if this device is detected during the initialization procedure. As a result, the device object will be flagged as "not initialized".

The following functions and events are affected with VE880 FXO support disabled:

- **`VpMakeLineObject()`** - returns `VP_STATUS_VTD_ERR_CODE` for any FXO termination type passed.
- FXO category events are unmaskable, cannot be generated:
  - `VP_LINE_EVID_RING_ON, VP_LINE_EVID_RING_OFF, VP_LINE_EVID_LIU, VP_LINE_EVID_LNIU, VP_LINE_EVID_FEED_DIS, VP_LINE_EVID_FEED_EN, VP_LINE_EVID_DISCONNECT, VP_LINE_EVID_RECONNECY, VP_LINE_EVID_POLREV, VP_LINE_EVID_POH, VP_LINE_EVID_PNOH`.

4.7.2.6 VP880_CLARE_RINGING_DETECT

When `VP880_FXO_SUPPORT` is enabled, the type of algorithm used for detecting signals on the FXO line can be selected using compile time value `VP880_CLARE_RINGING_DETECT`. For the VE880 silicon, Microsemi supports two basic FXO detection configurations:

1. CPC Clare Ringing Detect output connected to I/O4 of the Le883xx or Le88010 silicon - used when `#define VP880_CLARE_RINGING_DETECT`.
2. CPC Clare Ringing Detect output NOT connected to I/O4 - used when `#undef VP880_CLARE_RINGING_DETECT`.

All new designs should use configuration #1 (connect I/O4 to Ringing Detect output). This configuration has several advantages over configuration #2:

- Supports Line-In-Use Detection
- Does not require Ringing peak > 65V
- Does not require Ringing signals to contain polarity reversal.
- Ringing Frequency detection programmable to within +/-125us (then accuracy limited only by device tolerance). Configuration #2 accuracy is limited by the tickrate (+/-5ms/10ms typical).

4.7.3 VE890 Code Size Reductions

The following compile options can be used when `VP_CC_890_SERIES` is set to #define.

4.7.3.1 VP890_FXO_SUPPORT

When set to `#undef`, removes Product Code support for the following FXO only silicon:

- Le89010

Disabling 890 FXO support has the same functional impact to VE890 applications as removing 880 FXO support has to VE880 applications. See Section 4.7.2.5 for VE880 FXO support details.

4.7.3.2 VP890_FXS_SUPPORT

When set to `#undef`, removes Product Code support for the following FXS only silicon:

- Le89116, Le89136, Le89156

Disabling 890 FXS support has the same functional impact to VE890 applications as removing 880 FXS support has to VE880 applications. See Section 4.7.2.1 for VE880 FXS support details.

4.7.3.3 VP890_LP_SUPPORT

Disabling 890 Low Power support has the same functional impact and to VE890 applications as removing 880 Low Power support has to VE880 applications. See Section 4.7.2.4 for VE880 Low Power support details.

Notes:

VE890 Low Power Support requires `VP890_FXS_SUPPORT` set to `#define`, otherwise an error will be generated at compile time.

CHAPTER

5 SYSTEM CONFIGURATION FUNCTIONS

5.1 OVERVIEW

The VP-API-II supports the following key features:

- A single host microprocessor can control multiple device types and multiple line termination types through a common API.
- The VP-API-II is compatible with both multi-tasking and single-threaded operating systems.

To provide the key features, the VP-API-II introduces the concept of *device objects*, *line objects*, *device contexts* and *line contexts*. The System Configuration functions described in this chapter manage these objects and contexts, and are summarized below.

Required Functions for All Applications

- **`VpMakeDeviceObject()`** – This is the MOST important function of the VP-API-II and can only be called by one thread (for each physical device) in a system. This function initializes the device object and associates it with the device context. By initializing the device object with the `deviceId`, it links the Application space device object to the physical device. The `deviceId` is then passed along to any device context that is linked to this device object in the VP-API-II interface. The `deviceType` determines the silicon register set used for controlling the device.
- **`VpMakeLineObject()`** – This function can only be called by one thread (for each physical line) in a system. This function initializes the line object and associates it with the line and device context. By initializing the line object with the `channelId`, it links the Application space line context to the physical line in the silicon (value of `channelId` being limited to the range of the silicon). The `termType` tells the VP-API-II how a line is to be physically managed (i.e., controlled and line status detectors interpreted).

Additional Functions Required Functions for Multi-Threaded Applications

- **`VpMakeDeviceCtx()`** – This function initializes the device context and links it to the device object. The `deviceType` determines the silicon register set used for controlling the device.
- **`VpMakeLineCtx()`** – This function initializes the line context provided and associates it with the device context and line object provided (device context and line object previously existing).

Optional/Conditional Functions

- **`VpFreeLineCtx()`** – (required when using dynamic memory) This function is called when the Application has previously created a line context and now no longer intends to retain the memory for this object. **The system will crash if this function is not used properly!**
- **`VpMapLineId()`** – (optional) Using the user defined type `VpLineIdType` maps a `VpLineIdType` to a line. The value of `VpLineIdType` is present in many VP-API-II functions.

5.2 OBJECTS AND CONTEXTS

Device/line objects are software representations of the physical device/line, device/line contexts are handles to these objects. So there is exactly one device object for every device in the system, and there is exactly one line object for every line in the system. However, there may be several device/line contexts "pointing" to the same device/line object.

Only one type of device context and only one type of line context is defined by the VP-API-II. Since almost all VP-API-II functions accept either a device context or line context (or both), the same application can be used with multiple device types.

Conversely, there is a unique device object and a unique line object type for each type of device in the system (note that there is only one line object type for the same device type, regardless of the termination type). The device and line object types of the VP-API-II are shown in Table 5–1.

Table 5–1 Device and Line Objects

VpDeviceType	Compile-Time Switch	Device Object	Line Object
VP_DEV_790_SERIES	VP_CC_790_SERIES	**Vp790DeviceObjectType**	**Vp790LineObjectType**
VP_DEV_880_SERIES	VP_CC_880_SERIES	**Vp880DeviceObjectType**	**Vp880LineObjectType**
VP_DEV_580_SERIES	VP_CC_580_SERIES	**Vp580DeviceObjectType**	**Vp580LineObjectType**
VP_DEV_890_SERIES	VP_CC_890_SERIES	**Vp890DeviceObjectType**	**Vp890LineObjectType**

The appropriate compile-time switches must be set to include the source code defining the desired device and line object types specified in Table 5–1. The conditional flags are set in `vp_api_cfg.h`.

Figure 5–1 illustrates the interconnections between objects and contexts in the simplest case where the VP-API-II is controlling one primary device that supports *N* number of lines. In this example there is only one context associated with each object. The device context contains a link (pointer) to the device-specific object, and it also contains a link to each line context associated with the device. Each generic line context contains a link to a device-specific line object and a link back to its parent device context.

Figure 5–1 Device and Line Objects and Contexts

To support both static and dynamic memory models, the VP-API-II itself does not allocate any storage for any type of object or context. The Application must allocate storage for these structures, then execute VP-API-II functions to initialize the device and line objects and their respective contexts. If using dynamic memory model, the Application must call **`VpFreeLineCtx()`** for a line context **before** overwriting (or freeing) the memory space previously allocated for that line context.

5.3 FUNCTION DESCRIPTIONS

5.3.1 VpMakeDeviceObject()

SYNTAX

```
VpStatusType
VpMakeDeviceObject(
 VpDeviceType deviceType,          /* Type of Device (or device object) */
 VpDeviceIdType deviceId,          /* Device chip select identity */
 VpDevCtxType *pDevCtx,            /* Pointer to the Device Context */
 void *pDevObj)                    /* Pointer to the Device Object */
```

DESCRIPTION

This function does the following:

- Initializes the device object pointed to by **pDevObj**. This includes loading the value of deviceId into the device object, creating a link between the device object and the physical device (*).

  (*) This means:
  1. There can only be one device object per physical device in a given system, and
  2. New device contexts that want to access this physical device MUST link to this specific device object (see **VpMakeDeviceCtx()***).*

- Initializes the device context pointed to by **pDevCtx**. This includes initializing the link between the device context and the device object, which allows the Application to use *this* device context to access the physical device specified by the deviceId (contained in the device object).

Notes:
The device object contains all device and line calibration data. Therefore, after calling this function the device and all lines of the device MUST be calibrated (re-calibrated if previously calibrated).

deviceId

The deviceId argument is passed through the VP-API-II to the HAL used to uniquely identify *this* device. The VP-API-II makes no assumption about the VpDeviceIdType so it may be any typedef that is convenient for the system.

deviceType

The deviceType argument tells the VP-API-II which type of device it is communicating with on *this* deviceId and must be one of the values defined in Table 5–1.

This function returns VP_STATUS_INVALID_ARG if either pDevCtx or pDevObj are VP_NULL.

Notes:

1. *If this function does not return VP_STATUS_SUCCESS, then the application must not use the device context created here for any other VP-API-II calls.*
2. *No other VP-API-II function should be called before invoking this function.*
3. *The type of the device object allocated by the application must match with deviceType. The VP-API-II has no way to check this condition and the application could fail if there is a mismatch.*

RETURNS

See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

All

5.3.2 VpMakeLineObject()

SYNTAX

```
VpStatusType
VpMakeLineObject(
   VpTermType termType,              /* Type of line termination */
   uint8 channelId,                  /* Numeric ID for this channel */
   VpLineCtxType *pLineCtx,          /* Pointer to the Line Context */
   void *pLineObj,                   /* Pointer to the Line Object */
   VpDevCtxType *pDevCtx)            /* Ptr to associated device context */
```

DESCRIPTION

This function initializes the line object pointed by `pLineObj` and the line context pointed to by `pLineCtx`. It also associates the line context with both the line object and the device context pointed to by `pDevCtx` (note that the device context does not directly point to the line object).

In any given system, there can only be one line object for each physical line.

The termination type parameter (`termType`) describes the circuitry associated with the termination. The termination types matched to the devices that support each type are defined in Table 1–3.

The value required for `channelId` for a specific tip and ring port defined by the hardware used to drive that port. Meaning, it corresponds to the physical channel of the device, so it must equal '0' when referring to the 1st physical line, '1' when referring to the 2nd physical line, and so on. Note that a device may only provide one functional line but require `channelId` = 1. For example, VE890 devices with only a single FXO channel require `channelId=1` when creating the line object for that FXO.

In case where there are multiple devices in the system, it is likely that multiple lines will have the same `channelId`. In this case, the application can assign a unique identifier for the physical line by calling function **`VpMapLineId()`**. The **`VpMapLineId()`** function accepts a `VpLineIdType` which similar to `VpDeviceIdType` (for identifying the physical device) is simply passed through the VP-API-II (to events in case of the `lineId`) so it can be any typedef convenient for the application.

This function should be called once and only once for each channel managed by each VTD.

This function returns `VP_STATUS_INVALID_ARG` if `pLineCtx`, `pDevCtx` or `pLineObj` is `VP_NULL`.

Notes:

1. *No line-specific VP-API-II functions should be called before invoking this function.*
2. *The device context pointed to by* `pDevCtx` *must be initialized with* **`VpMakeDeviceObject()`** *or* **`VpMakeDeviceCtx()`** *before calling this function.*
3. *The type of line object allocated by the application must be compatible with the device type of the given device context. The VP-API-II has no way to check this condition, and the application could fail if there is a mismatch. See Table 5–1 for a list of device types and matching line object types.*
4. *This function must be called before executing* **`VpInitDevice()`**. *See* *VpInitDevice(), on page 56* *for more information.*

FUNCTION RETURNS

See *VP-API-II Function Return Type, on page 11*

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

All

5.3.3 VpMakeDeviceCtx()

SYNTAX

```
VpStatusType
VpMakeDeviceCtx(
   VpDeviceType deviceType,          /* Type of device (or device object) */
   VpDevCtxType *pDevCtx,            /* Pointer to the device context */
   void *pDevObj)                    /* Pointer to the device object */
```

DESCRIPTION

This function is required only for multi-threaded applications. It initializes the device context pointed to by `pDevCtx` and links it to the device object pointed to by `pDevObj`.

In a multi-threaded application, only one thread should initialize the device object by calling **`VpMakeDeviceObject()`**. All other process referring to the same device object must call this function. The `pDevObj` argument provided to this function must point to the same device object that was used during its creation.

The `deviceType` argument must indicate the device type that was specified during the device object creation.

This function returns `VP_STATUS_INVALID_ARG` if `pDevCtx` or `pDevObj` is `VP_NULL`.

After calling this function the application may use the device context in other VP-API-II function calls.

RETURNS

See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

All

5.3.4 VpMakeLineCtx()

SYNTAX

```
VpStatusType
VpMakeLineCtx(
   VpLineCtxType *pLineCtx,          /* Pointer to the line context */
   void *pLineObj,                   /* Pointer to the line object */
   VpDevCtxType *pDevCtx)            /* Ptr to associated device context */
```

DESCRIPTION

This function is required only for multi-threaded applications. It initializes the line context provided and associates it with the device context and line object provided.

In a multi-threaded application, only one process should initialize the line object by calling **`VpMakeLineObject()`**. All other processes that want to refer to the same line object must call this function. The `pLineObj` argument must point to the same line object that was used during its creation.

This function returns `VP_STATUS_INVALID_ARG` if `pLineCtx`, `pDevCtx` or `pLineObj` is `VP_NULL`.

After calling this function the application may use the line context in other VP-API-II function calls.

RETURNS

See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

All

5.3.5 VpFreeLineCtx()

SYNTAX

```
VpStatusType
VpFreeLineCtx(
  VpLineCtxType *pLineCtx)        /* Pointer to line context */
```

DESCRIPTION

This function is required in dynamic memory allocation applications where previously allocated memory for a line context or associated line object may be reclaimed by the application. The VP-API-II needs to know when the memory pointed to by the line context is no longer valid in order to prevent invalid memory access and potential segmentation fault.

In case more than one line context is associated with one line object (see *Multi-Tasking Applications*, on page 41), this function must be called for all such line contexts before freeing up the memory allocated to the line object.

This function does not alter the state of the physical line it simply prevents further access to (potentially) invalid memory. Prior to calling this function, the application is expected to perform any such cleanup tasks (e.g., placing the line in Disconnect, disabling interrupts for the line, etc.).

Notes:

If using a Low Power Termination (e.g., `VP_TERM_FXS_LOW_PWR`, `VP_TERM_FXS_ISOLATE_LP` or `VP_TERM_FXS_SPLITTER_LP`) the VP-API-II requires 1 second of **`VpApiTick()`** *after setting the line to `VP_LINE_DISCONNECT` (i.e., using* **`VpSetLineState()`***) for the line to completely enter the `VP_LINE_DISCONNECT` state. So if using one of these termination types, the exact sequence for applying* **`VpFreeLineCtx()`** *is:*

1. *Set line to `VP_LINE_DISCONNECT` using* **`VpSetLineState()`**.
2. *Wait 1 second for the line to completely enter `VP_LINE_DISCONNECT` state.*
3. *Call* **`VpFreeLineCtx()`** *for the line. At this point, the line is fully in `VP_LINE_DISCONNECT` and will not be accessed by any VP-API-II function.*

RETURNS

See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

All

5.3.6 VpMapLineId()

SYNTAX

```
VpStatusType
VpMapLineId(
 VpLineCtxType *pLineCtx,         /* Pointer to line context */
 VpLineIdType lineId)             /* Value assigned as line Id */
```

DESCRIPTION

This function can be used to assign a system-wide line identification (`lineId`) to a given line. The value `lineId` is simply passed through the VP-API-II along with the event in the **`VpGetEvent()`** and **`VpGetLineInfo()`** functions.

RETURNS

See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

All

CHAPTER

6 INITIALIZATION FUNCTIONS

6.1 OVERVIEW

This chapter discusses VP-API-II functions that perform initialization. These functions are summarized below.

- `VpInitDevice()` – Initializes all FXS and FXO lines of a device and applies the specified profiles to those lines.
- `VpInitLine()` – Initializes an individual FXS or FXO line and applies the specified profiles to that line.
- `VpConfigLine()` – Sets the AC, DC, and Ring Profiles for an individual FXS line.
- `VpInitCustomTermType()` – CSLAC-580 only. Configures the SLAC to Line I/O Connections used to set line state and detect line conditions.
- `VpCal()` – Calibrates the line to external parameters and provides methods for managing system calibration coefficients.
- `VpCalLine()` – Calibrates internal line level parameters.
- `VpInitRing()` – Sets the ringing parameters such as the ringing cadence and Caller ID profile for an individual FXS line.
- `VpInitCid()` – Prepares a FXS line for a Caller ID ring sequence.
- `VpInitMeter()` – Configures the metering signal generator of an individual FXS line.
- `VpInitProfile()` – Initializes the device's profile tables.

6.2 FUNCTION DESCRIPTIONS

6.2.1 VpInitDevice()

SYNTAX

```
VpStatusType
VpInitDevice(
 VpDevCtxType *pDevCtx,            /* Pointer to device context */
 VpProfilePtrType pDevProfile,     /* Pointer to Device Profile */
 VpProfilePtrType pAcProfile,      /* Pointer to AC profile */
 VpProfilePtrType pDcProfile,      /* Pointer to DC profile */
 VpProfilePtrType
pRingProfile,                      /* Pointer to ringing profile */
 VpProfilePtrType
pFxoAcProfile,                     /* Pointer to FXO AC profile */
 VpProfilePtrType
pFxoCfgProfile)                    /* Pointer to FXO config profile */
```

DESCRIPTION

This function initializes all lines controlled by the VTD associated with the device context. This includes performing the recommended power-up sequence, calibrating device level parameters, and setting the device and all lines of the device to their default option values. Note that even though this function performs steps that could be accessed by other VP-API-II functions, when complete it only generates **VP_DEV_EVID_DEV_INIT_CMP** event. It will NOT generate other events corresponding to the equivalent VP-API-II functions being performed (e.g., **VP_LINE_EVID_LINE_INIT_CMP** and **VP_EVID_CAL_CMP**).

Requirements:

- **VpInitDevice()** must be called after creating the device via **VpMakeDeviceObject()**. It is highly recommended also to create all lines via **VpMakeLineObject()** before calling this function, but not required. Lines created after calling **VpInitDevice()** must be initialized using **VpInitLine()**.
- The `pDevProfile` parameter takes a pointer to a Device Profile and is **required**. The device profile provides basic information needed to reliably communicate with and initialize the device. This function returns an error code if `pDevProfile` does not point to a valid profile.
- Proper line operation requires complete set of valid profiles. These are: AC, DC and Ringing (FXS termination types), or AC and FXO/Dialing (for FXO termination types). The lines should be kept in either **VP_LINE_DISCONNECT** (FXS) or **VP_LINE_FXO_LOOP_OPEN** (FXO) until all valid profiles have been provided to the line.
- The profile types accepted by this function are described in *Profile Types*, on page 13. For each of these profiles, the application can supply either a pointer to a valid profile or a profile table index. Valid profiles must be loaded into the profile table before a profile table index can be used. See *Profiles*, on page 13.
- The profiles passed to this function may be removed from Application space once this function returns.
- Once this function is called, except for **VpGetEvent()** and **VpApiTick()** the Application must prevent access to the affected device until the **VP_DEV_EVID_DEV_INIT_CMP** event is generated.

VP880_INCLUDE_MPI_QUICK_TEST (VE880 only) and
VP890_INCLUDE_MPI_QUICK_TEST (VE890 only):
The beginning of **VpInitDevice()** for VE880 and VE890 devices will perform a test of the MPI interface and Hardware Abstraction Layer implementation (i.e., **VpMpiCmd()**). If an error is detected, this function will return **VP_STATUS_ERR_SPI**. Enabling **VP_DEBUG_ERROR** output will provide details about the error that can be used for debugging. Once a design is proven to work, this test can be compiled out by undefining **VP8xx_INCLUDE_MPI_QUICK_TEST** (xx = 80 for VE880, xx = 90 for VE890 series) in **vp_api_cfg.h**.

Upon completion of the device and associated line initialization as a result of calling this function, all FXS lines are placed in the `VP_LINE_DISCONNECT` line state, all FXO lines are placed in the `VP_LINE_FXO_LOOP_OPEN` line state.

The relay states are applied as stated in **`VpInitLine()`** for each supported termination type.

Notes:

1. *The application may provide* `VP_PTABLE_NULL` *pointer to the FXO and FXS specific profiles in this function, then provide valid profiles later using* **`VpInitLine()`** *or* **`VpConfigLine()`**.
2. *Only lines that have previously been created and associated with the device using* **`VpMakeLineObject()`** *will be initialized at the time this function is called.*
3. *This function performs a Hardware Reset causing all device/line activity to abruptly stop. Therefore, if calling this function on currently active lines it is recommended that the application first put all lines of the device into a steady state (i.e., Ringing Cadence, Line Test, etc. should all be stopped by the application prior to calling this function).*

See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED

VP_DEV_EVID_DEV_INIT_CMP, on page 89

DEVICES All

TERMINATIONS All

6.2.2 VpInitLine()

SYNTAX

```
VpStatusType
VpInitLine(
 VpLineCtxType *pLineCtx,          /* Pointer to line context */
 VpProfilePtrType pAcProfile,      /* Pointer to AC profile */
 VpProfilePtrType
   pDcFeedOrFxoCfgProfile,         /* Ptr to DC feed or FXO cfg profile */
 VpProfilePtrType
   pRingProfile)                   /* Pointer to ringing profile */
```

DESCRIPTION

This function resets all line parameters and options to their default values, loads the profiles (if any) provided, sets the line to default states (**VP_LINE_DISCONNECT** for FXS termination types, **VP_LINE_FXO_LOOP_OPEN** for FXO termination types) and sets the line relay state according to the list below.

FXS Line Termination Type	Relay State
VP_TERM_FXS_75282	VP_RELAY_RESET
All others	VP_RELAY_NORMAL

Notes:
*The line behavior is NOT guaranteed until all valid profiles are provided. Until all valid profiles are provided, the line should be kept in the **VP_LINE_DISCONNECT** (for FXS termination types) or **VP_LINE_FXO_LOOP_OPEN** (for FXO termination types) state.*

Requirements:

- If the target line is an FXS termination the **pDcFeedOrFxoCfgProfile** argument MUST point to a DC profile. If the target line is an FXO termination the **pDcFeedOrFxoCfgProfile** argument MUST point to a FXO/Dialing configuration profile.
- For FXO line types the **pRingProfile** argument is ignored and may be set to **VP_NULL** or **VP_PTABLE_NULL**.
- The profiles passed to this function may be removed from Application space once this function returns.
- Once this function is called, the Application must prevent access to the line affected until the **VP_LINE_EVID_LINE_INIT_CMP** event is generated.
- Once a line has been calibrated, it is not necessary to re-calibrate the same line after calling **VpInitLine()**. This function does not clear the (if previously applied or computed) calibration data.

RETURNS

See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED

VP_LINE_EVID_LINE_INIT_CMP, on page 89

DEVICES

All

TERMINATIONS

All

6.2.3 VpConfigLine()

SYNTAX

```
VpStatusType
VpConfigLine(
  VpLineCtxType *pLineCtx,          /* Pointer to line context */
  VpProfilePtrType pAcProfile,      /* Pointer to AC profile */
  VpProfilePtrType
    pDcFeedOrFxoCfgProfile,         /* Ptr to DC feed or FXO cfg profile */
  VpProfilePtrType
    pRingProfile)                   /* Pointer to ringing profile */
```

DESCRIPTION

This function re-initializes the specified line's AC, DC (or FXO/Dialing for FXO lines), and ring parameters with the profiles provided by the function's arguments. Unlike **VpInitLine()** or **VpInitDevice()**, this function does not reset the line or the device; it merely loads the given profiles.

Requirements:

- Only parameters specified by a valid profile pointer are affected by this function. Parameters specified by a VP_PTABLE_NULL or VP_NULL profile pointer are ignored leaving the line's configuration unchanged for the given parameter.
- The profiles provided to this function can be removed from Application space once this function returns.
- Once a line has been calibrated, it is not necessary to re-calibrate the same line after calling **VpConfigLine()**. This function does not clear the line calibration values if previously provided or computed.
- For FXO line types the **pRingProfile** argument is ignored and may be set to **VP_NULL** or **VP_PTABLE_NULL**.

Notes:

1. *When changing the Ringing and/or DC parameters, the order of these parameters being changed is not guaranteed. Therefore, to avoid false ring trip or other hook detect issues, the line should be set to **VP_LINE_DISCONNECT** state prior to reconfiguring the line.*
2. *Most Applications will not need to call this function since the line can be configured during initialization by passing the necessary profiles to **VpInitDevice()** or **VpInitLine()**.*

RETURNS

See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

All

6.2.4 VpInitCustomTermType()

SYNTAX

```
VpStatusType
VpInitCustomTermType(
 VpDevCtxType *pDevCtx,          /* Pointer to device context */
 VpLineCtxType *pLineCtx,        /* Pointer to line context */
 VpProfilePtrType                /* Pointer to profile that defines the
   pCustomTermProfile)           custom termination type */
```

DESCRIPTION

This function is used to complete the definition of a line termination type. It is required for CSLAC-580 designs using a combination of CODECs and SLICs that are not covered by standard Application reference designs.

RETURNS

See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED

None

DEVICES

CSLAC-580

TERMINATIONS

FXS

6.2.5 VpCal()

SYNTAX

```
VpStatusType
VpCal(
 VpLineCtxType *pLineCtx,          /* Pointer to line context */
 VpCalType calType,                /* Calibration to perform */
 void *inputArgs)                  /* Pointer to input args */
```

DESCRIPTION

This function invokes various calibration routines. It accepts a `calType` parameter to indicate the calibration to be performed and a void pointer of parameters to use during calibration. The specific values pointed to by `inputArgs` will depend on the calibration routine being performed. Descriptions as follows:

```
Enumeration Data Type: VpCalType:
    VP_CAL_BFILTER          /* Calibrate B-Filter */
    VP_CAL_APPLY_BFILTER    /* Apply specific B-Filter coefficients */
    VP_CAL_MEASURE_BFILTER /* Measure current B-Filter performance */
    VP_CAL_GET_SYSTEM_COEFF /* Retrieves device and line calibration values */
    VP_CAL_APPLY_SYSTEM_COEFF /* Programs device and line calibration values */
```

`VP_CAL_BFILTER`: B-Filter calibration is done on an FXO line to minimize the 4-wire (D-D) echo due to line impedance mismatch. It must be done on a quiet off-hook line after dial tone has been removed.

Value of `inputArgs` for `calType` `VP_CAL_BFILTER` is:

```
typedef struct {
    uint16 vRms;                /* Target rms output (error) */
    VpProfilePtrType pAcProfile; /* Pointer to AC profile with B-Filter coeff */
} VpCalBFilterType;
```

Calibration of the B-Filter runs until the reflected signal is less than the `vRms` value, or until all B-Filter coefficients in the AC Profile provided have been tried. In the latter case, the set of B-Filter coefficients providing the least `vRms` reflected level will be loaded in the device when calibration completes. The `index` of the best coefficient set in the AC profile is returned in the `parmHandle` field of the `VP_EVID_CAL_CMP` event. The `vRms` level of the best coefficient set is returned in the `eventData` parameter of the event. The average return loss with the new B-Filter coefficients can be estimated by: `-20 * log10(vRms / 14535) dB`

`VP_CAL_APPLY_BFILTER`: This routine will apply a specific B-Filter coefficient set from an AC profile to the device.

Value of `inputArgs` for `calType` `VP_CAL_APPLY_BFILTER` is:

```
typedef struct {
    uint16 index;               /* Index of a B-Filter set in the AC Profile */
    VpProfilePtrType pAcProfile; /* Pointer to AC profile with B-Filter coeff */
} VpCalApplyBFilterType;
```

This can be used to either restore the coefficient results from a previous run of `VP_CAL_BFILTER` or to step through the coefficient sets in an AC profile for manual testing. The `index` parameter corresponds to the `index` result returned from the `VP_CAL_BFILTER` routine.

`VP_CAL_MEASURE_BFILTER`: This routine measures the `vRms` reflected level performance with the current coefficients. The `vRms` result is returned in the `eventData` parameter of the `VP_EVID_CAL_CMP` event. The average return loss with the new B-Filter coefficients can be estimated as above in the description of `VP_CAL_BFILTER`.

`VP_CAL_GET_SYSTEM_COEFF`: This routine retrieves the device calibration values associated with the line context passed and the calibration values for all lines associated with the device (i.e., the line passed and all other lines on the same device). The device and all lines on the device must have been calibrated using **`VpInitDevice()`** and **`VpCalLine()`** or loaded with calibration values using option `VP_CAL_APPLY_SYSTEM_COEFF` prior to running this routine. Otherwise this function will return `VP_STATUS_LINE_NOT_CONFIG`.

When using VP_CAL_GET_SYSTEM_COEFF the parameter hasResults in the event structure filled in by **VpGetEvent()** will be set to TRUE indicating that the application must call **VpGetResults()** to retrieve the calibration values. The size of the buffer to provide to **VpGetResults()** for calibration values is the number of bytes indicated in the event structure by parameter eventData. The string of data returned by the VP-API-II may be stored and later provided back into the system using VP_CAL_APPLY_SYSTEM_COEFF.

VP_CAL_APPLY_SYSTEM_COEFF: This routine loads device calibration values associated with the line context passed and the calibration values for all lines associated with the device (i.e., the line passed and all other lines on the same device). Performing this procedure prior to calling **VpInitDevice()** causes the VP-API-II to skip the device calibration performed normally as part of **VpInitDevice()** and the application will not need to call **VpCalLine()**. Calling **VpCalLine()** after system coefficients have been provided will have no negative affect, but simply results in an immediate event being generated (the device will not be accessed).

Value of inputArgs for calType VP_CAL_APPLY_SYSTEM_COEFF is a pointer to a uint8 string formatted with calibration coefficients per VP_CAL_GET_SYSTEM_COEFF. If the uint8 pointer (inputArgs) is set = VP_NULL when calType = VP_CAL_APPLY_SYSTEM_COEFF, the calibration values will be reset requiring either a new set of calibration values to be applied or run-time calibration to be performed.

Notes:

Event **VP_EVID_CAL_CMP** *will not be generated in the case where calibration coefficients are applied prior to calling* **VpInitDevice()** *(using VP_CAL_APPLY_SYSTEM_COEFF). This is because a non-initialized device cannot generate events. See* Section 4.2.2 for details.

RETURNS See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED
VP_EVID_CAL_CMP (when calibration has completed successfully)
VP_EVID_CAL_BUSY (when calibration could not be completed successfully)

DEVICES CSLAC-880, CSLAC-890

TERMINATIONS All

6.2.6 VpCalLine()

SYNTAX

```
VpStatusType
VpCalLine(
 VpLineCtxType *pLineCtx)          /* Pointer to line context */
```

DESCRIPTION

Calibration is required on all VE880 and VE890 FXS lines in order to meet Data Sheet Specifications. This requirement can be met in one of two ways:

1. *Calling* `VpCalLine()` *for the line - This will perform run-time calibration on the line and apply the computed correction factors to the line upon completion. If successful* `VpCalLine()` *will generate the* `VP_EVID_CAL_CMP` *event. If all lines of the device are complete, the system (device and all lines of the same device) calibration values can be read using* `VpCal()`.
2. *Apply previously determined calibration factors (such as those computed using the methods described in step #1) to the system using* `VpCal()`. *Note that using this method, the calibration factors are not applied until the next call to* `VpApiTick()`.

To run `VpCalLine()` the line must be set to an "on-hook" state. Valid "on-hook" states are: `VP_LINE_STANDBY`, `VP_LINE_STANDBY_POLREV`, `VP_LINE_OHT` or `VP_LINE_OHT_POLREV`.

If calibration correction factors are applied prior to calling `VpInitDevice()` the VP-API-II will skip it's normal run-time device calibration in `VpInitDevice()` and apply the device correction factors provided.

If calibration correction factors are applied prior to calling `VpCalLine()` then `VpCalLine()` will return `VP_STATUS_SUCCESS` and immediately generate the `VP_EVID_CAL_CMP` event, but will NOT access the device. It only provides the appearance of successful calibration execution in order to present a consistent behavior to the Application that a software developer can write to.

There are three ways to reset/change the calibration factors:

1. *Calling* `VpCal()` *- When this function is called the VP-API-II will overwrite previous calibration factors with the new calibration factors provided. Note there are two modes to apply calibration factors: new values and reset (if providing NULL). For details, refer to* `VpCal()` *in* **Section 6.2.5**.
2. *Calling* `VpMakeDeviceObject()` *or* `VpMakeLineObject()` *- This causes all data in the device/line object to be reset, including most of the calibration data. The VP-API-II cannot restore the object data from silicon content alone. The Application must restart.*
3. *Hardware Reset - Most silicon content is reset, so regardless of calibration this condition cannot be restored by the VP-API-II. The Application must restart.*

For all other function calls, including device/line initialization and configuration (`VpInitDevice()`, `VpInitLine()` or `VpConfigLine()`) repeated calibration is NOT necessary.

Notes:

1. `VpCalLine()` is not performed automatically by the VP-API-II. If the Application is not using `VpCal()` to apply calibration correction factors, then the Application **must call** this function for VE880 and VE890 device families in order to avoid hook and ring trip detect issues.
2. The procedures performed in `VpCalLine()` may cause Tip/Ring and Vbat (Tracker Devices) transients. This may cause noise on other lines if proper layout procedures are not taken.
3. No other VP-API-II function for the line should be called while the line is calibrating.
4. Calibration should not be run while a phone is offhook on the line.

RETURNS

See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED

```
VP_EVID_CAL_CMP   /* Calibration was Successful */
VP_EVID_CAL_BUSY  /* Calibration failed or could not be performed */
```

DEVICES

CSLAC-880, CSLAC-890

TERMINATIONS

FXS

6.2.7 VpInitRing()

SYNTAX

```
VpStatusType
VpInitRing(
 VpLineCtxType *pLineCtx,          /* Pointer to line context */
 VpProfilePtrType
   pCadProfile,                    /* Pointer to ringing cadence profile */
 VpProfilePtrType
   pCidProfile)                    /* Pointer to Caller ID profile */
```

DESCRIPTION

This function initializes the Ringing Cadence and CID sequence applied to the line when the line is set to the Ringing state (calling **VpSetLineState()** with either VP_LINE_RINGING or VP_LINE_RINGING_POLREV). The default VP-API-II behavior is "ringing cadence = always on" and "no CID". This function is used to support general ringing cadence and systems with Type-I Caller ID.

The profiles may be specified by pointer or profile tables as described in *Profile Tables*, on page 14.

The pCadProfile argument specifies the ringing cadence. If pCadProfile is VP_PTABLE_NULL then ringing cadence is "always on" (i.e., no cadence).

The pCidProfile argument specifies the Caller ID Profile used within the ringing cadence. Note that if no ringing cadence is specified, the CID profile cannot be used. The CID profile is used ONLY if the ringing profile contains either the "Start CID" or the "Wait on CID.." element.

If Type-I Caller ID is enabled for a line, the application must first call **VpInitCid()** to copy Caller ID data into the Caller ID transmit buffer before putting the line into the Ringing state. See *VpInitCid()*, on page 65 for more information. Note that although **VpInitCid()** must be called each time prior to generating CID, **VpInitRing()** need only be called once for a given ringing and (Type-I) CID profile. Once initialized, these cadences are applied each time the line is put into the ringing state or until otherwise changed (i.e., additional calls to this function or **VpInitDevice()** / **VpInitLine()**).

This function does not start ringing on the line. The application must call **VpSetLineState()** with VP_LINE_RINGING or VP_LINE_RINGING_POLREV as the state argument in order to start ringing the line.

Notes:

1. *Type-I Caller ID referred to in this function has the CLI (Caller Line Identity) frame transmitted as part of the caller alert sequence (ringing signal plus Caller ID sequence and data). This type of CLI is only transmitted while the line is on-hook. Applications with Caller ID that is NOT associated with Ringing (e.g., Type-II Caller ID) should refer to function* *VpSendCid()*, on page 109.

RETURNS

See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

FXS

6.2.8 VpInitCid()

SYNTAX

```
VpStatusType
VpInitCid(
 VpLineCtxType *pLineCtx,        /* Pointer to line context */
 uint8 length,                   /* Length of Caller ID data */
 uint8p pCidData)                /* Pointer to the Caller ID data */
```

DESCRIPTION

This function must be called before placing a line into the Ringing state (for Type-I CID) if the associated line was set-up for Caller ID by **VpInitRing()**.

The `length` argument should specify the total length in bytes of the entire message to be transmitted if the message length is less than or equal to 32 bytes, otherwise it should be set to 32. For Caller ID messages longer than 32 bytes, the application will need to make several VP-API-II function calls to transmit a complete Caller ID message. To facilitate this, the VP-API-II generates the `VP_LINE_EVID_CID_DATA` event (with `eventData` equal to `VP_CID_DATA_NEED_MORE_DATA`) when it can accept 16 more bytes of Caller ID data. Upon receiving the `VP_LINE_EVID_CID_DATA:VP_CID_DATA_NEED_MORE_DATA` event, the application must call **VpContinueCid()** to provide the VP-API-II with additional Caller ID data.

If this function is called with `length` less than or equal to 16 bytes, then the VP-API-II immediately generates the `VP_CID_DATA_NEED_MORE_DATA` event.

The `pCidData` argument should point to a buffer containing the initial bytes to be sent as the Caller ID message. Neither the VP-API-II nor the VTD automatically generate the message type or message length. These should be included, if desired, in the buffer pointed to by `pCidData`.

When the VTD is done transmitting Caller ID it generates the `VP_LINE_EVID_CID_DATA` event with `eventData` member set to `VP_CID_DATA_TX_DONE`.

RETURNS

See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED

VP_LINE_EVID_CID_DATA

Note: The Application must refer to the `eventData` when this event is generated in order to determine the state of the Caller ID state machine.

DEVICES

All except CSLAC-580

TERMINATIONS

FXS

6.2.9 VpInitMeter()

SYNTAX

```
VpStatusType
VpInitMeter(
 VpLineCtxType *pLineCtx,            /* Pointer to line context */
 VpProfilePtrType pMeterProfile)     /* Ptr to metering profile */
```

DESCRIPTION

This function initializes the metering parameters for the specified line, using the values contained in the Metering Profile. It should be called prior to initiating one or more metering pulses using **VpStartMeter()**.

Like other profiles, the metering profiles used by this function may be pre-loaded in the profile tables or can be directly loaded from application memory. Refer to *Profile Tables*, on page 14 for more information.

If `pMeterProfile` is `VP_PTABLE_NULL`, nothing happens and this function simply returns `VP_STATUS_SUCCESS`.

RETURNS

See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED

None

DEVICES

CSLAC-880

TERMINATIONS

FXS

6.2.10 VpInitProfile()

SYNTAX

```
VpStatusType
VpInitProfile(
 VpDevCtxType *pDevCtx,              /* Pointer to device context */
 VpProfileType type,                 /* Type of profile to load */
 VpProfilePtrType pProfileIndex,     /* Profile index selector */
 VpProfilePtrType pProfile)          /* Pointer to the profile data */
```

DESCRIPTION

This function initializes an entry in the VP-API-II profile table. The VP-API-II implements *soft* profile tables for the CSLAC devices wherein this function simply saves the profile pointer (`pProfile`) for later use. When subsequent VP-API-II function calls reference this profile by its index, the VP-API-II looks-up the saved pointer and reads the profile from application memory. See *Profile Tables*, on page 14 for more information.

The default entries in the profile table may not contain valid profiles. Hence the application should initialize the profile table with valid profiles if it intends to reference them later.

The profile `type` is given by the following enumeration:

```
Enumeration Data Type: VpProfileType:
    VP_PROFILE_DEVICE              /* Device profile */
    VP_PROFILE_AC                  /* AC Profile */
    VP_PROFILE_DC                  /* DC Profile */
    VP_PROFILE_RING                /* Ringing Profile */
    VP_PROFILE_RINGCAD             /* Ringing Cadence Profile */
    VP_PROFILE_TONE                /* Tone Profile */
    VP_PROFILE_METER               /* Metering Profile */
    VP_PROFILE_CID                 /* Caller ID Profile */
    VP_PROFILE_TONECAD             /* Tone Cadence Profile */
    VP_PROFILE_FXO_CONFIG          /* FXO Configuration Profile */
```

The `pProfileIndex` parameter determines which profile in the table is updated. The argument should be of the form `VP_PTABLE_INDEXx`, where `x` is the index into the profile table. This value `x` must not be larger than number of entries in the profile table for the given profile `type`. Refer to Table 2–2 for the maximum value for `pProfileIndex` for each profile type. If `pProfileIndex` is `VP_PTABLE_NULL`, this function returns `VP_STATUS_INVALID_ARG`.

If `pProfile` is `VP_PTABLE_NULL`, this function marks the profile table entry as uninitialized. Subsequent VP-API-II function calls attempting to use this profile table entry return the `VP_STATUS_ERR_PROFILE` error code.

Notes:

1. *The application can call this function to modify a profile that is currently being used by one or more lines, but using this function in this manner is not recommended. In this case the application should immediately call functions such as* **`VpInitDevice()`**, **`VpInitLine()`**, **`VpConfigLine()`**, *or* **`VpInitRing()`** *to apply the updated profile to the relevant lines.*
2. *All VP-API-II functions that take profile pointers return `VP_STATUS_ERR_PROFILE` if called with a profile table index pointing to an uninitialized profile table entry. The application must call this function to initialize a profile table entry before attempting to use that profile table entry.*

RETURNS

See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

All

CHAPTER

7 INTERRUPT AND EVENT HANDLING

7.1 OVERVIEW

This chapter discusses how to manage device and VP-API-II level interrupts. It also describes functions required by the CSLAC VP-API-II (used to generate the VP-API-II interrupts). Note that functions described in this section are provided by Microsemi but may require interrupt management functions implemented by the customer. Interrupt management functions (when needed) to be implemented by the customer are described in Chapter 11, System Services.

The functions described in this section are summarized below:

- `VpApiTick()` – Called periodically to provide a timing basis for the VP-API-II.
- `VpVirtualISR()` – Services CSLAC device interrupts.
- `VpGetEvent()` – Returns events corresponding to a device.
- `VpFlushEvents()` – Flushes all outstanding events.
- `VpGetResults()` – Reads the data associated with an event.

The first section of this chapter will focus on configuration of the VP-API-II for one of four interrupt modes, the second section will provide a detailed description of each function.

7.2 HANDLING INTERRUPTS FROM CSLAC DEVICES

CSLAC devices have almost none of the internal timing mechanisms that are required to implement many of the VP-API-II features. Specifically, meter pulse generation, ringing cadence, ring exit debounce, caller id, etc.. all require the use of an application timer that when used with device level interrupts provide the functionality specified at the VP-API-II level. This means that in any given application there are much more device level interrupts than there are VP-API-II events. Similarly, the customer should not confuse device level level interrupts with VP-API-II events. A given VP-API-II event may be the result of one or several device level interrupts and timing.

The timer required by the VP-API-II is implemented in an application call to `VpApiTick()` at a (application) specified interval. The `VpApiTick()` function manages the lower level device interrupts and timing, and provides an indication to the application when a VP-API-II event occurs.

The next few sections discuss the functions required to manage the device interrupts and VP-API-II timing (i.e., `VpApiTick()`). Note that the functions required will depend on the interrupt mode, and some of the functions must be implemented by the user.

7.2.1 VP-API-II Time Base

Mentioned previously, the time base of the VP-API-II is a periodic call to `VpApiTick()` from this point on referred to as the "tick". The tick period is user-defined and selected in the device profile (provided to `VpInitDevice()`). The nominal interval at which `VpApiTick()` is called must match that specified in the device profile, otherwise unpredictable behavior will result.

Notes:

Due to current silicon limitations, the preferred tick interval for CSLAC devices is 5ms or 10ms. Contact Microsemi support if using a tick interval that exceeds 10ms

7.2.1.1 VP-API-II Tick derived from a Timer Interrupt - Example

Microsemi's demo applications implement the VP-API-II Tick rate from an OS (Linux) timer ISR. This timer ISR results in call to `VpApiTick()` every time the interrupt occurs. In Microsemi's VP-API-II the tick interval is generally configurable, but for Microsemi's demo applications each interrupt occurs every 10 ms.

The choice of an appropriate interrupt rate is dependent on the customer's application and the required system latency for reporting the supervisory events such as ring trip, and on-hook/off-hook/pulse dial detection. See Chapter 4, Design Considerations for details.

7.2.1.2 VP-API-II Tick Details

In addition to providing the timing base for the VP-API-II, `VpApiTick()` also services any interrupts that have occurred during the previous tick period. If running in either of the two interrupt modes or the efficient polled mode (see Section 7.2.2), then `VpApiTick()` only services SLAC device interrupts after the SLAC device interrupt line goes active. In simple polled mode, on the other hand, the tick function will perform interrupt polling during each tick.

7.2.2 CSLAC Device Interrupt Architecture Variations

The requirement of an application utilizing the VP-API-II is providing the VP-API-II with a SLAC device interrupt indication. Here, the system designer has a choice of four SLAC device interrupt handling modes:

- **Mode 1: Interrupt Driven, Edge Triggered –** The SLAC device generates an interrupt, which is edge-triggered, in the processor. The ISR sets an interrupt flag, which is serviced during the next VP-API-II tick period.
- **Mode 2: Interrupt Driven, Level Triggered –** The SLAC device generates an interrupt, and the ISR disables interrupts at the microprocessor and sets interrupt flag. Interrupt is serviced and interrupts are re-enabled during the next VP-API-II tick period.
- **Mode 3: Efficient Polled Mode –** The SLAC device Interrupt Line is tied to either a disabled microprocessor interrupt line or a PIO line. The VP-API-II tick function services SLAC device interrupts only when the SLAC device interrupt line is active, which is polled every VP-API-II tick.
- **Mode 4: Simple Polled Mode –** The SLAC device interrupts are disabled. The SLAC device is polled (via an MPI interrupt read command) every VP-API-II Tick for an interrupt, whether one has occurred or not.

All four methods are supported by Microsemi's API-II. The desired interrupt mode is changed by a compile time value in vp_api_cfg.h. The method chosen depends upon both the hardware architecture in the customer's system and differences in interrupt handling between various microprocessors.

In order to support any given interrupt mode, certain functions must be provided by the customer's application. The exact functions required depend on the interrupt mode chosen (see details in the following sub-sections). The VP-API-II provides declarations of the functions required in sys_service.c and sys_service.h. This document describes how to properly define these functions for each interrupt mode in Chapter 11, on page 123. The API-II includes example definitions of these functions. It is the customer's responsibility to provide these functions for their platform.

The sub-sections below describe each mode, in detail.

7.2.2.1 Mode 1: Interrupt Driven, Edge-Triggered

This mode should be used when the SLAC device's interrupt line is connected to an edge-triggered interrupt line on the micro-processors.

Interrupt servicing sequence of events:

1. The SLAC device's interrupt line goes Active (Low).
2. The customer's BSP calls the corresponding ISR.
3. This fielding action results in a call to `VpVirtualISR()`.

4. `VpVirtualISR()`, in turn, sets a flag that signals `VpApiTick()` to service the interrupt.
5. `VpApiTick()` services the interrupt on the occurrence of the next VP-API-II tick (See Figure 7–1).
6. `VpApiTick()` reads from the device until either the interrupt is cleared or until the value set for max interrupts for the device has been reached (See Figure 7–2).

Between the time of the initial interrupt and the subsequent tick function call, the SLAC device continues to provide an active interrupt indication (INT line low). The microprocessor is configured for edge-triggered interrupts, so it ignores the active interrupt. No further edges occur until `VpApiTick()` services the pending interrupts) on the next VP-API-II tick.

Notes:

Multiple interrupts can occur between tick function calls. In this scenario, the SLAC device pulses its interrupt line after `VpApiTick()` *reads the SLAC device's interrupt register. The SLAC device pulse causes the SLAC ISR to set the interrupt flag again before the tick function completes. The tick function services the interrupts up to the value set for max interrupts in the Device Profile for each VP-API-II tick. See Section 7.2.1 for further information.*

Notes:

The QSLAC/VE580 Series Data Sheets recommend using Level Triggered Interrupts with QSLAC/VE580 series devices. See Section 7.2.2.2 for further information on using Level Triggered Interrupts with the VP-API-II.

Figure 7–1 Mode 1: Interrupt Driven, Edge-Triggered (Flow)

Figure 7–2 Mode 1: Interrupt Driven, Edge-Triggered (Timing)

7.2.2.2 Mode 2: Interrupt Driven, Level-Triggered

The second method is a derivation of the first, and allows systems that use level-triggered interrupts to work. The only requirement is that the customer's hardware has the ability to disable/ignore an active SLAC device interrupt. Actions occur in the following sequence:

1. The SLAC device's interrupt line goes Active (Low).
2. The customer's BSP fields the corresponding ISR.
3. The customer's ISR calls `VpVirtualISR()`.

`VpVirtualISR()`, in turn, calls `VpSysDisableInt()` and then sets a flag that signals `VpApiTick()` to service the interrupt. `VpSysDisableInt()` disables the SLAC device interrupt, thus preventing a flurry of interrupts from occurring until the tick function can service the SLAC device interrupt. See Figure 7–3.

4. `VpApiTick()` services the interrupt indicated by the flag. See Figure 7–4.
5. `VpApiTick()` continues to read from the device until either the interrupt is no longer active, or the value set for max interrupts for the device has been reached.
6. `VpApiTick()` re-enables the SLAC device interrupts by calling `VpSysEnableInt()`.

Microsemi's System Services Layer specifies two functions to control the enabling and disabling of the SLAC device interrupt:

- `VpSysDisableInt()` is called from the `VpVirtualISR()`
- `VpSysEnableInt()` is called just after `VpApiTick()` services the SLAC device interrupt.

To use level sensitive interrupts, the customer must implement these two hardware-dependent functions to enable and disable the hardware servicing of a SLAC device interrupt.

Notes:

Multiple interrupts can occur between tick function calls. In this scenario, the SLAC device pulses its interrupt line after `VpApiTick()` *reads the SLAC device's interrupt register and re-enables SLAC device interrupts (by calling* `VpSysEnableInt()`*). The SLAC device causes the SLAC device ISR to set the interrupt flag again before the tick function completes. The tick function services the interrupts up to the value set for max interrupts in the Device Profile for each VP-API-II tick. See Section 7.2.1 for further information.*

Notes:

The QSLAC/VE580 Series Data Sheets recommend using Level Triggered Interrupts with QSLAC/VE580 series devices.

Figure 7–3 Mode 1: Interrupt Driven, Level Triggered (Flow)

Figure 7–4 Mode 1: Interrupt Driven, Level Triggered (Timing)

7.2.2.3 Mode 3: Efficient Polled Mode

Although polling traditionally implies inefficiency, the Efficient Polled Mode interrupt handling mode is actually the most efficient method for handling SLAC device interrupts. In this method, the SLAC device's interrupt pin is connected to a non-interrupting PIO pin on the microprocessor that is "polled" to test the status of the SLAC device's interrupt pin. The function `VpSysTestInt()` tests for a SLAC device interrupt by reading the SLAC device's Interrupt Pin as presented on the PIO pin. `VpApiTick()` will service the SLAC device only if `VpSysTestInt()` returns true.

Notes:

Multiple interrupts can occur between tick function calls. In this scenario, the SLAC device pulses its interrupt line after `VpApiTick()` *reads the SLAC device's interrupt register. The SLAC device pulse causes* `VpSysTestInt()` *to indicate a new interrupt before the tick function completes. The tick function services the interrupts up to the value set for max interrupts in the Device Profile for each VP-API-II tick. See* Section 7.2.1 *for further information.*

Figure 7–5 Efficient Polling

7.2.2.4 Mode 4: Simple Polled Mode

The final method works in those remaining cases where interrupts are not indicated by the SLAC device's interrupt line. In this case, the SLAC device's interrupt pin does not need to be connected to the microprocessor. Instead, the interrupt state of each SLAC device in the system is polled by a SLAC device MPI command on every VP-API-II tick. You can also use this interrupt mode for debugging target hardware and application software.

Notes:

Multiple interrupts can occur between tick function calls. In this scenario, `VpApiTick()` *continues reading the SLAC device's interrupt register until it indicates no more interrupts. The tick function services the interrupts up to the value set for max interrupts in the Device Profile for each VP-API-II tick. See* Section 7.2.1 *for further information.*

Figure 7–6 Polled Interrupt Servicing

7.3 FUNCTION DESCRIPTIONS

7.3.1 VpApiTick()

SYNTAX

```
VpStatusType
VpApiTick(
 VpDevCtxType *pDevCtx,          /* Pointer to device context */
 bool *pEventStatus)             /* Pointer to the event pending flag */
```

DESCRIPTION

This function reads and debounces all supervisory inputs for the target CSLAC device and should be called at regular time intervals. See *VP-API-II Time Base*, on page 67 for more information. Any timing needed by the VP-API-II is derived from this periodic function call. This function writes `TRUE` to the location pointed to by `pEventStatus` if a VP-API-II event is pending; otherwise `FALSE` is written to this location. The application should call **`VpGetEvent()`** if an event is pending.

This function must be called exactly once every VP-API-II tick period for every device. Thus, if there are four VTDs in the system, this function must be called four times every tick period—each time with a different device context. It can be attached to an interrupt and called directly from an interrupt service routine.

This function is time deterministic. Only a limited number of interrupts are handled per VP-API-II tick period, which is defined in the Device Profile. This value can be modified by the Profile Wizard application.

This function issues a variable number of MPI commands per call, dependent upon cadence changes and CSLAC interrupt status. The maximum execution time can be adjusted by changing number of interrupts handled per VP-API-II tick and the tick rate.

RETURNS

See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED

None

DEVICES

CSLAC

TERMINATIONS

All

7.3.2 VpVirtualISR()

SYNTAX

```
VpStatusType
VpVirtualISR(
 VpDevCtxType *pDevCtx)          /* Pointer to device context */
```

DESCRIPTION

This function is used by the VP-API-II when **`Mode 1: Interrupt Driven, Edge-Triggered`** described in Section 7.2.2.1 or **`Mode 2: Interrupt Driven, Level-Triggered`** described in Section 7.2.2.2 are used. This function must be called from the host microprocessor's CSLAC interrupt service routine and should not be called if there is no CSLAC interrupt. This routine updates internal variables indicating to the VP-API-II that an interrupt has occurred and must be serviced.

RETURNS

None

EVENTS GENERATED

None

DEVICES

CSLAC

TERMINATIONS

All

7.3.3 VpGetEvent()

SYNTAX

```
bool
VpGetEvent(
 VpDevCtxType *pDevCtx,        /* Pointer to device context */
 VpEventType *pEvent)          /* Pointer to target event data buffer */
```

DESCRIPTION

The **VpGetEvent()** function reports a single VTD event. This function must be called if the **VpApiTick()** function sets its event flag to TRUE, indicating that an event has occurred.

The pDevCtx argument must point to the context of the VTD reporting an event. The pEvent argument must point to an application buffer for the event data returned by this function. The application buffer should be of **VpEventType** type, which is defined as follows:

```
typedef struct {
    VpStatusType status;  /* Function return status */
    uint8 channelId;      /* Channel which caused the event */
    VpLineCtxType *pLineCtx;  /* Pointer to the line context corresponding to
                               * the line that caused the event */
    VpLineIdType lineId;      /* Application provided line Id to ease mapping
                               * of lines to specific line contexts. */
    VpDeviceIdType deviceId; /* Id of the device that caused the event */
    VpDevCtxType *pDevCtx;    /* Pointer to the device context corresponding to
                               * the device that caused the event */
    VpEventCategoryType eventCategory; /* Event category */
    uint16 eventId;      /* Unique event ID (within event category) */
    uint16 parmHandle;   /* Event's parameter or application handle */
    uint16 eventData;    /* Data associated with the event */
    bool hasResults;     /* Indicates whether event has extra results */
} VpEventType;
```

The status variable indicates whether an error occurred while executing this function. This function's boolean return value indicates whether an event was retrieved from the VTD. The application should interpret these two variables as follows:

- status == VP_STATUS_SUCCESS and **VpGetEvent()** returned TRUE
  An event was retrieved from the VTD, event data valid.
- status == VP_STATUS_SUCCESS and **VpGetEvent()** returned FALSE
  No event was retrieved from the VTD, ignore event data.
- status != VP_STATUS_SUCCESS
  VpGetEvent() encountered an error. Ignore event data and function return value. See *VP-API-II Function Return Type*, on page 11 for a complete list of error codes.

If the event is line-specific, the channelId and pLineCtx variables indicate which line caused the event. If the event is device-specific, channelId and pLineCtx should be ignored. The deviceId and pDevCtx variables will identify the device that caused the event.

Events are classified into event categories so that the application can easily process them. The eventCategory member of the event structure indicates which category the event belongs to; eventCategory can be any of the following values:

```
Enumeration Data Type: VpEventCategoryType:
    VP_EVCAT_FAULT                /* Fault event category */
    VP_EVCAT_SIGNALING            /* Signaling event category */
    VP_EVCAT_RESPONSE             /* Response event category */
    VP_EVCAT_TEST                 /* Test event category */
    VP_EVCAT_PROCESS              /* Call Process event category */
    VP_EVCAT_FXO                  /* FXO event category */
```

The individual event ID is passed through the eventId member of the event structure.

The `parmHandle` and `eventData` variables contain additional event-specific information. The boolean `hasResults` indicates whether additional data related to the event is available. If `hasResults` == `TRUE` the application must retrieve the additional data using **`VpGetResults()`** before calling **`VpGetEvent()`** again.

Chapter 8, on page 77 describes the `eventId` for each value of `eventCategory`, `parmHandle`, `eventData` and extended results (`hasResults` == `TRUE`) for each VP-API-II event.

Notes:

This function returns only non-masked events. Events masks are set by calling **`VpSetOption()`** *with the* `VP_OPTION_ID_EVENT_MASK` *option. The default event masks are set when* **`VpInitDevice()`** *function is called.*

RETURNS `TRUE` if an event is pending `FALSE` otherwise. See function description for details.

EVENTS GENERATED None—this function is called to retrieve an event.

DEVICES All

TERMINATIONS All

7.3.4 VpFlushEvents()

SYNTAX

```
VpStatusType
VpFlushEvents(
  VpDevCtxType *pDevCtx)            /* Pointer to device context */
```

DESCRIPTION This function empties the VP-API-II event queue, but it does not clear the VP-API-II results buffer. The application must call **`VpGetResults()`** to clear the results buffer.

RETURNS See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED None

DEVICES All

TERMINATIONS All

7.3.5 VpGetResults()

SYNTAX

```
VpStatusType
VpGetResults(
 VpEventType *pEvent,      /* Ptr to event that was filled by VpGetEvent() */
 void *pResults)           /* Pointer to buffer for the results */
```

DESCRIPTION

This function retrieves data associated with events marked by the event `hasResults` member set to `TRUE`. When `hasResults` == `TRUE` the results must be retrieved calling **`VpGetResults()`**.

To retrieve results from the VTD, the application must allocate a buffer large enough to hold the result structure then call this function with a pointer to that buffer. This function copies the result data into the buffer and frees the VP-API-II result buffer allowing other results to be retrieved. The size of the buffer will generally be defined in the application by "`sizeof(Result Type)`" specified in Table 7–1)" or "`sizeof(event.eventData)`" (for event `VP_EVID_CAL_CMP` generated by **`VpCal()`** with `calType` = `VP_CAL_GET_SYSTEM_COEFF`).

In case the application does not know (or care about) the results being generated, a buffer of size `sizeof(VpResultsType)` should be provided. The type `VpResultsType` is an internal VP-API-II type defined to encompass the largest size possible returned by **`VpGetResults()`** given the VP-API-II compile options (device and feature dependant).

Table 7–1 lists the functions that generate results, the corresponding event ID and results data type.

Table 7–1 VP-API-II Functions with Extended Results

Function	Event ID	Result Type
`VpSetRelGain()`	`VP_LINE_EVID_GAIN_CMP`	**`VpRelGainResultsType`**
`VpGetLoopCond()`	`VP_LINE_EVID_RD_LOOP`	**`VpLoopCondResultsType`**
`VpGetOption()`	`VP_LINE_EVID_RD_OPTION`	See Section 10.2.6.
`VpDeviceIoAccess()`	`VP_DEV_EVID_IO_ACCESS_CMP`	**`VpDeviceIoAccessDataType`**
`VpCal()` **(see note 2)**	`VP_EVID_CAL_CMP`	**`uint8p`**

Notes:

1. *When reading options using* **`VpGetOption()`**, *the application can use the* `eventData` *member of the event structure to determine the option type (**`VpOptionIdType`**) that was read.*

RETURNS

See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

All

CHAPTER

8 EVENTS

8.1 OVERVIEW

The VP-API-II uses an abstract event type to report VTD events to the host application. These events typically correspond to asynchronous VTD interrupts or occur as a result of some command issued by the application. VP-API-II events are organized into categories, with several events in each category. Each event may have some combination of a time stamp, a handle, event data, or event results attached to the event. This chapter covers all VP-API-II events in detail and describes the data types attached to each event. Each event is described using the following format:

DESCRIPTION This is a summary description of the event and what causes it.

T.S. OR HANDLE This parameter is identified in the event structure by `parmHandle`.

An event can have a time stamp, user defined handle, or event specific value associated with it.

- Event time stamps are reported as 16-bit integers in units of 0.5 ms.
- Event handles are 16-bit variables that the application can use to associate an event with a prior command. For some VP-API-II functions, the application can provide a handle that is returned with the event carrying the results for that function. The VP-API-II does not use the handle in any way; it simply passes the handle back to the application with an event. The application can use the handle for any purpose, or ignore it altogether.
- Some events use neither the time stamp nor the handle, in which case this field may be marked "N/A."

EVENT DATA This parameter is identified in the event structure by `eventData`.

Every event carries a 16-bit variable that may contain a small amount of data associated with the event. The meaning of this variable is described for each individual event. Some events do not use this variable, in which case this field is marked "N/A."

RESULTS This parameter is identified by the member `hasResults`.

Not all events have data associated with them, but those that do fall into two categories: data is <= 16-bit, or data is > 16-bit. Events that have results larger than 16-bit are indicated by element in the event structure `hasResults = TRUE`. Conversely, events with no results or results <= 16-bit are indicated by `hasResults = FALSE`. Note that the same event may be generated in conditions where it can have results and NOT have results. So the application must check `hasResults` flag for every event.

In cases where `hasResults = TRUE` the VP-API-II uses the concept of a *mailbox* to pass this data back to the application. The application must then call **`VpGetResults()`** to retrieve the event data from the mailbox. The type of the result data is described in this section for each event. In cases where the results are a variable length (`VP_EVID_CAL_CMP` when generated for `calType` = `VP_CAL_GET_SYSTEM_COEFF` and event `VP_LINE_EVID_LLCMD_RX_CMP`) the length of data is specified in event element `eventData`.

DEVICES This field lists the devices that can generate the event.

TERMINATIONS This field lists the termination types (FXS, FXO, All) that can generate the event. Termination type "All" means either all termination types supported by the applicable devices, or the termination type is not relevant to the event.

The `VpEventType` definition is provided in function *VpGetEvent(), on page 74*.

8.2 EVENT SUMMARY

Table 8–1 on page 78 lists all events that the VP-API-II can generate. The events are organized into categories, and these categories are defined in the software by the `VpEventCategoryType` enumeration indicated in the event structure by element `eventCategory`. Notice that some events apply only to certain device types and some events apply only to certain termination types.

Events are generally either device-specific or line-specific. In rare cases an event type may be both device and line, meaning the same type event can be generated by a line context and by a device context. For readability, the event name starts with the type it can be generated by as follows:

- Starts: `VP_DEV_EVID_` -> Device Event only. Will NOT be generated by a line.
- Starts: `VP_LINE_EVID_` -> Line Event only. Will NOT be generated by a device.
- Starts: `VP_EVID_` -> Unspecified. Will be generated by both device and line types.

Table 8–1 List of VP-API-II Events

Event ID (`eventId`)	Devices	Terminations	Page
Fault Events (`eventCategory = VP_EVCAT_FAULT`)			
VP_DEV_EVID_BAT_FLT	CSLAC-790	FXS	80
VP_DEV_EVID_CLK_FLT	All	All	80
VP_LINE_EVID_THERM_FLT	All	FXS	80
VP_LINE_EVID_DC_FLT	CSLAC-790, CSLAC-890	All	81
VP_LINE_EVID_AC_FLT	CSLAC-790	FXS	81
VP_LINE_EVID_RES_LEAK_FLT	CSLAC-880, CSLAC-890	FXS Low Power Types	81
Signaling Events (`eventCategory = VP_EVCAT_SIGNALING`)			
VP_LINE_EVID_HOOK_OFF	All	FXS	82
VP_LINE_EVID_HOOK_ON	All	FXS	82
VP_LINE_EVID_GKEY_DET	All	FXS	83
VP_LINE_EVID_GKEY_REL	All	FXS	83
VP_LINE_EVID_FLASH	All	FXS	83
VP_LINE_EVID_STARTPULSE	All	FXS	84
VP_LINE_EVID_DTMF_DIG	All	FXS	84
VP_LINE_EVID_PULSE_DIG	All	FXS	85
VP_DEV_EVID_TS_ROLLOVER	All	All	85
VP_LINE_EVID_EXTD_FLASH	All	FXS	84
Response Events (`eventCategory = VP_EVCAT_RESPONSE`)			
VP_LINE_EVID_RD_OPTION	All	All	86
VP_EVID_CAL_CMP	CSLAC-880, CSLAC-890, CSLAC-790	All	86
VP_EVID_CAL_BUSY	CSLAC-880, CSLAC-890, CSLAC-790	All	86
VP_LINE_EVID_RD_LOOP	CSLAC-790	FXS	87
VP_LINE_EVID_GAIN_CMP	CSLAC-880, CSLAC-890, CSLAC-580	All	88

VP_DEV_EVID_DEV_INIT_CMP	All	All	89
VP_LINE_EVID_LINE_INIT_CMP	All	All	89
VP_DEV_EVID_IO_ACCESS_CMP	All	All	89
Process Events (eventCategory = VP_EVCAT_PROCESS)			
VP_LINE_EVID_MTR_CMP	All	FXS	90
VP_LINE_EVID_MTR_ABORT	All	FXS	90
VP_LINE_EVID_CID_DATA	All	FXS	91
VP_LINE_EVID_RING_CAD	All	FXS	91
VP_LINE_EVID_SIGNAL_CMP	All	All	92
VP_LINE_EVID_TONE_CAD	All	All	92
FXO Events (eventCategory = VP_EVCAT_FXO)			
VP_LINE_EVID_RING_ON	CSLAC-880, CSLAC-890, CSLAC-580	FXO	93
VP_LINE_EVID_RING_OFF	CSLAC-880, CSLAC-890, CSLAC-580	FXO	93
VP_LINE_EVID_LIU	CSLAC-880, CSLAC-890	FXO	93
VP_LINE_EVID_LNIU	CSLAC-880, CSLAC-890	FXO	94
VP_LINE_EVID_FEED_DIS	CSLAC-880, CSLAC-890, CSLAC-580	FXO	94
VP_LINE_EVID_FEED_EN	CSLAC-880, CSLAC-890, CSLAC-580	FXO	94
VP_LINE_EVID_DISCONNECT	CSLAC-880, CSLAC-890, CSLAC-580	FXO	94
VP_LINE_EVID_RECONNECT	CSLAC-880, CSLAC-890, CSLAC-580	FXO	95
VP_LINE_EVID_POLREV	All except CSLAC-790	FXO	95
VP_LINE_EVID_POH	CSLAC-880, CSLAC-890, CSLAC-580	FXO	95
VP_LINE_EVID_PNOH	CSLAC-880, CSLAC-890, CSLAC-580	FXO	97
Test Events (eventCategory = VP_EVCAT_TEST)			
VP_LINE_EVID_TEST_CMP	CSLAC-880, CSLAC-890	FXS	97

8.3 FAULT EVENTS

The fault events report critical VTD errors. The set of valid fault events is defined in the software by the `VpFaultEventType` enumeration.

8.3.1 VP_DEV_EVID_BAT_FLT

DESCRIPTION
This event occurs when a battery fault is detected or is no longer detected. Refer to the appropriate *Chip Set User's Guide* for details on the battery fault condition.

T.S. OR HANDLE (`parmHandle`)
N/A

EVENT DATA (`eventData`)
Event data indicates the source of the battery fault and can be any of the values shown below. See *Battery Name Interpretation*, on page 88 for battery mapping.

```
Enumeration Data Type: VpBatFltEventDataType:
    VP_BAT_FLT_NONE = 0x00        /* No battery fault */
    VP_BAT_FLT_BAT1 = 0x02        /* Battery 1 fault */
    VP_BAT_FLT_BAT2 = 0x01        /* Battery 2 fault */
    VP_BAT_FLT_BAT3 = 0x04        /* Battery 3 fault */
```

RESULTS (`hasResults`)
FALSE

DEVICES
CSLAC-790

TERMINATIONS
FXS

8.3.2 VP_DEV_EVID_CLK_FLT

DESCRIPTION
This event occurs when a clock fault is detected or is no longer detected. Refer to the appropriate *Chip Set User's Guide* for details on the clock fault condition.

T.S. OR HANDLE (`parmHandle`)
N/A

EVENT DATA (`eventData`)
Event data bit 0 indicates whether the fault condition is present (1) or absent (0).

RESULTS (`hasResults`)
FALSE

DEVICES
All

TERMINATIONS
All

8.3.3 VP_LINE_EVID_THERM_FLT

DESCRIPTION
This event occurs when a thermal fault is detected or is no longer detected. The FXS line may be configured to automatically transition to the `VP_LINE_DISCONNECT` state when this event (with "present" indication) occurs. See `VP_DEVICE_OPTION_ID_CRITICAL_FLT` in Section 3.3.1 for details.

T.S. OR HANDLE (`parmHandle`)
Timestamp

EVENT DATA (`eventData`)
'1' = Thermal Fault is Present, '0' indicates Thermal Fault is Absent

RESULTS (`hasResults`)
FALSE

DEVICES
All

TERMINATIONS
FXS

8.3.4 VP_LINE_EVID_DC_FLT

DESCRIPTION	For FXS termination types, this event occurs when a DC fault occurs. For FXO termination types, this event occurs when a high DC voltage is applied (but not necessarily a fault). The FXS line may be configured to automatically transition to the `VP_LINE_DISCONNECT` state when this event (with "present" indication) occurs. See `VP_DEVICE_OPTION_ID_CRITICAL_FLT` in Section 3.3.1 for details.
T.S. OR HANDLE (`parmHandle`)	N/A
EVENT DATA (`eventData`)	Event data bit 0 indicates whether the DC condition is present (1) or absent (0).
RESULTS (`hasResults`)	FALSE
DEVICES	CSLAC-790, CSLAC-890
TERMINATIONS	All

8.3.5 VP_LINE_EVID_AC_FLT

DESCRIPTION	This event occurs when a AC fault is detected or is no longer detected. The FXS line may be configured to automatically transition to the `VP_LINE_DISCONNECT` state when this event (with "present" indication) occurs. See `VP_DEVICE_OPTION_ID_CRITICAL_FLT` in Section 3.3.1 for details.
T.S. OR HANDLE (`parmHandle`)	N/A
EVENT DATA (`eventData`)	Event data bit 0 indicates whether the fault condition is present (1) or absent (0).
RESULTS (`hasResults`)	FALSE
DEVICES	CSLAC-790
TERMINATIONS	FXS

8.3.6 VP_LINE_EVID_RES_LEAK_FLT

DESCRIPTION	This event occurs when the VTD detects a leaky resistance on the line and is generated only for termination types `VP_TERM_FXS_LOW_PWR` and `VP_TERM_FXS_SPLITTER_LP`. When this event is generated, the device is taken out of "Low Power" mode automatically. The status of a leaky resistive line can also be determined by **`VpGetDeviceStatus()`** or **`VpGetLineStatus()`**. The leaky line status will reset when the line is taken off-hook. When put back on-hook, the leaky line detection will repeat and a new event will be reported (if the line remains leaky).
T.S. OR HANDLE (`parmHandle`)	N/A
EVENT DATA (`eventData`)	N/A
RESULTS (`hasResults`)	FALSE
DEVICES	CSLAC-880, CSLAC-890
TERMINATIONS	FXS Low Power Types

8.4 SIGNALING EVENTS

The signaling events report changes on an individual line. The set of valid signaling events is defined in the software by the `VpSignalingEventType` enumeration.

8.4.1 VP_LINE_EVID_HOOK_OFF

DESCRIPTION	The behavior of this event depends on whether pulse-digit decoding is enabled or disabled. See **VP_DEVICE_OPTION_ID_PULSE** and **VP_DEVICE_OPTION_ID_PULSE2** for details. If pulse-digit decoding is enabled, this event occurs when the VTD/VP-API-II determines that the line is off-hook beyond the pulse-digit make period. Therefore, the application would not observe this event during pulse digit dialing with digit decoding enabled. If pulse-digit decoding is disabled, this event occurs every time the VTD/VP-API-II detects the off-hook condition. Therefore the application will observe this event during pulse digit dialing with digit decoding disabled.
T.S. OR HANDLE (`parmHandle`)	Timestamp
EVENT DATA (`eventData`)	N/A
RESULTS (`hasResults`)	FALSE
DEVICES	All
TERMINATIONS	FXS

8.4.2 VP_LINE_EVID_HOOK_ON

DESCRIPTION	The behavior of this event depends on whether pulse-digit decoding is enabled or disabled. See **VP_DEVICE_OPTION_ID_PULSE** and **VP_DEVICE_OPTION_ID_PULSE2** for details. If pulse-digit decoding is enabled, this event occurs when the VTD/VP-API-II determines that the line is on-hook beyond the pulse-digit break period and hook flash period. In other words, this event does not occur during pulse dialing or a hook-switch flash. The exception is when an invalid pulse train is detected and an on-hook occurs while monitoring the pulse train. In that case, only an on-hook event is generated rather than an invalid digit. If pulse-digit decoding is disabled, this event occurs every time the VTD/VP-API-II detects the on-hook condition. This event is reported during pulse dialing and a hook-switch flash.
T.S. OR HANDLE (`parmHandle`)	Timestamp
EVENT DATA (`eventData`)	N/A
RESULTS (`hasResults`)	FALSE
DEVICES	All
TERMINATIONS	FXS

8.4.3 VP_LINE_EVID_GKEY_DET

DESCRIPTION	This event occurs when the ground-key condition is detected, which is used in ground-start signaling. If the system does not support ground-start signaling, then this condition could indicate a DC fault on the line.
T.S. OR HANDLE (parmHandle)	Timestamp
EVENT DATA (eventData)	N/A
RESULTS (hasResults)	FALSE
DEVICES	All
TERMINATIONS	FXS

8.4.4 VP_LINE_EVID_GKEY_REL

DESCRIPTION	This event occurs when the ground-key condition is no longer detected (ground-key release).
T.S. OR HANDLE (parmHandle)	Timestamp
EVENT DATA (eventData)	N/A
RESULTS (hasResults)	FALSE
DEVICES	All
TERMINATIONS	FXS

8.4.5 VP_LINE_EVID_FLASH

DESCRIPTION	This event indicates that a hook-switch flash was detected. This event only occurs if pulse-digit decoding is enabled via `VpSetOption()`. See `VP_DEVICE_OPTION_ID_PULSE` and `VP_DEVICE_OPTION_ID_PULSE2` for details.
T.S. OR HANDLE (parmHandle)	Timestamp
EVENT DATA (eventData)	N/A
RESULTS (hasResults)	FALSE
DEVICES	All
TERMINATIONS	FXS

8.4.6 VP_LINE_EVID_STARTPULSE

DESCRIPTION	This event occurs when the start of a pulse digit or flash has been detected. This is useful for determining when to turn-off dial tone at the start of dialing. This event only occurs if pulse-digit decoding is enabled via `VpSetOption()`. See `VP_DEVICE_OPTION_ID_PULSE` and `VP_DEVICE_OPTION_ID_PULSE2` for details.
T.S. OR HANDLE (`parmHandle`)	Timestamp
EVENT DATA (`eventData`)	N/A
RESULTS (`hasResults`)	FALSE
DEVICES	All
TERMINATIONS	FXS

8.4.7 VP_LINE_EVID_EXTD_FLASH

DESCRIPTION	This event can only be generated if `EXTENDED_FLASH_HOOK` is set to #define in `vp_api_cfg.h`. This event occurs when an extended flash hook has been detected. This is useful for determining detection of "new call" request (flash duration longer than "hook flash" but less than "on hook"). This event only occurs if pulse-digit decoding is enabled via `VpSetOption()`. See `VP_DEVICE_OPTION_ID_PULSE` and `VP_DEVICE_OPTION_ID_PULSE2` for details.
T.S. OR HANDLE (`parmHandle`)	Timestamp
EVENT DATA (`eventData`)	N/A
RESULTS (`hasResults`)	FALSE
DEVICES	All
TERMINATIONS	FXS

8.4.8 VP_LINE_EVID_DTMF_DIG

DESCRIPTION	This event occurs at the beginning and end of DTMF digit detection. For CSLAC devices the application must implement some DTMF decoding resource and call `VpDtmfDigitDetected()` in order for this event to occur. Refer to *VpDtmfDigitDetected(), on page 111* for more information.
T.S. OR HANDLE (`parmHandle`)	Timestamp
EVENT DATA (`eventData`)	Event data bits 3 to 0 contain the received digit information, which can be decoded by comparing with the `VpDigitType` (see *VpSetLineTone(), on page 105*) enumeration constants. Bit 4 indicates whether this event corresponds to the start (1) or the end (0) of the DTMF digit.
RESULTS (`hasResults`)	FALSE
DEVICES	All
TERMINATIONS	FXS

8.4.9 VP_LINE_EVID_PULSE_DIG

DESCRIPTION	This event occurs when a pulse digit is detected. This event only occurs if pulse-digit decoding is enabled via `VpSetOption()`. See `VP_DEVICE_OPTION_ID_PULSE` and `VP_DEVICE_OPTION_ID_PULSE2` for details.
T.S. OR HANDLE (`parmHandle`)	'0' if the digit detected meets the parameters specified by `VP_DEVICE_OPTION_ID_PULSE`, '1' if the digit detected meets the parameters specified by `VP_DEVICE_OPTION_ID_PULSE2`.
EVENT DATA (`eventData`)	Event data bits 3 to 0 contain the received digit information, which can be decoded by comparing with the `VpDigitType` enumeration constants. Event data will be VP_DIG_NONE if while monitoring the pulse train any digit fails to meet the `breakMin`, `breakMax`, `makeMin`, `makeMax`, or if an on-hook occurs within the `interDigitMin` time specified by `VP_DEVICE_OPTION_ID_PULSE` and `VP_DEVICE_OPTION_ID_PULSE2`
RESULTS (`hasResults`)	FALSE
DEVICES	All
TERMINATIONS	FXS

8.4.10 VP_DEV_EVID_TS_ROLLOVER

DESCRIPTION	This event occurs when the VP-API-II time stamp counter rolls-over from 65535 to 0. Since this counter is effectively incremented every 0.5 ms, it rolls-over every 32.768 seconds. For CSLAC devices the time stamp counter may not actually increment at a 0.5 ms rate, but roll-over period is still 32.768 seconds.
T.S. OR HANDLE (`parmHandle`)	Timestamp
EVENT DATA (`eventData`)	N/A
RESULTS (`hasResults`)	FALSE
DEVICES	All
TERMINATIONS	All

8.5 RESPONSE EVENTS

The response events occur as a result of some action initiated by the application. Several of these events have extended results data associated with them. The set of valid response events is defined in the software by the `VpResponseEventType` enumeration.

8.5.1 VP_LINE_EVID_RD_OPTION

DESCRIPTION	This event occurs as a result of calling **`VpGetOption()`** and indicates that the VP-API-II has retrieved the requested option setting from the VTD. See *VpGetOption(), on page 120* for information on that function. This event is non-maskable.
T.S. OR HANDLE (`parmHandle`)	Handle
EVENT DATA (`eventData`)	Event data is of `VpOptionIdType` type and indicates which option was read from the VTD, allowing the application to correctly interpret the associated results data. See *Option Summary, on page 17* for the complete list of VP-API-II options.
RESULTS (`hasResults`)	The data type (value of `hasResults`) of the result associated with this event depends on exactly which option was read. The application should determine the option data type by inspecting the event data field, allocate a buffer of the appropriate type, and call **`VpGetResults()`** with a pointer to that buffer. Chapter 3 describes the result type for each VP-API-II option.
DEVICES	All
TERMINATIONS	All

8.5.2 VP_EVID_CAL_CMP

DESCRIPTION	This event occurs as a result of calling **`VpCalLine()`** or **`VpCal()`** and indicates that the requested calibration procedure completed successfully. Note that disabling (masking) this event blocks this event for all of the **`VpCalLine()`** and **`VpCal()`** functions.
T.S. OR HANDLE (`parmHandle`)	When **`VpCal()`** is called for the `VP_CAL_BFILTER` routine, this field contains the index of the B-Filter coefficient set that was selected and applied to the device. N/A otherwise.
EVENT DATA (`eventData`)	When **`VpCal()`** is called for the `VP_CAL_BFILTER` or `VP_CAL_MEASURE_BFILTER` routines, this field contains the measured `vRms` result. See *VpCal(), on page 61* for more information. N/A otherwise.
RESULTS (`hasResults`)	The value of `(hasResults)` depends on the cause of this event.
DEVICES	CSLAC-880, CSLAC-890, CSLAC-790
TERMINATIONS	All

8.5.3 VP_EVID_CAL_BUSY

DESCRIPTION	This event occurs as a result of calling **`VpCalLine()`** or **`VpCal()`** `and` indicates that the requested operation cannot be performed at this time. Note that disabling (masking) this event blocks this event for both the **`VpCal()`** and **`VpCalLine()`** functions.
T.S. OR HANDLE (`parmHandle`)	N/A
EVENT DATA (`eventData`)	Enumerated values indicated the Calibration Status, with ’0’ = VP_RSRC_CONFLICT.
RESULTS (`hasResults`)	FALSE
DEVICES	CSLAC-880, CSLAC-890, CSLAC-790
TERMINATIONS	All

8.5.4 VP_LINE_EVID_RD_LOOP

DESCRIPTION

The event occurs as a result of calling **`VpGetLoopCond()`** and indicates that the loop condition results are available. See *VpGetLoopCond(), on page 115* for information on that function.

This event is non-maskable.

T.S. OR HANDLE (`parmHandle`)

Handle

EVENT DATA (`eventData`)

N/A

RESULTS (`hasResults`)

The results associated with this event are passed through the `VpLoopCondResultsType` structure shown below.

```
typedef struct {
    int16 rloop;                  /* Measured loop resistance */
    int16 ilg;                    /* Sensed longitudinal (common mode) current */
    int16 imt;                    /* Sensed metallic (differential) current */
    int16 vsab;                   /* Sensed voltage on AB (tip/ring) leads */
    int16 vbat1;                  /* Battery 1 measured voltage */
    int16 vbat2;                  /* Battery 2 measured voltage */
    int16 vbat3;                  /* Battery 3 measured voltage */
    int16 mspl;                   /* Measured metering signal peak level */
    VpBatteryType selectedBat;    /* Battery currently used for DC feed */
    VpDcFeedRegionType dcFeedReg;/* DC feed region presently selected */
} VpLoopCondResultsType;

Enumeration Data Type: VpBatteryType:
    VP_BATTERY_UNDEFINED          /* Not known or feature not supported */
    VP_BATTERY_1                  /* Battery 1 */
    VP_BATTERY_2                  /* Battery 2 */
    VP_BATTERY_3                  /* Battery 3 */

Enumeration Data Type: VpDcFeedRegionType:
    VP_DF_UNDEFINED               /* Not known or feature not supported */
    VP_DF_ANTI_SAT_REG            /* DC feed is in anti saturation region */
    VP_DF_CNST_CUR_REG            /* DC feed is in constant current region */
    VP_DF_RES_FEED_REG            /* DC feed is in resistive feed region */
```

The application can convert the integer results in this structure to real-world values using the conversion equations found in the following tables. Note that these equations assume that the application is using the recommended external components. Refer to the appropriate Microsemi documentation (*Chip Set User's Guide* or *Data Sheet*) for recommended application circuits.

Table 8–2 Loop Condition Results Conversion 790

Loop Condition	CSLAC-790
Loop Resistance	(`rloop` / 32768) x 11.67 kΩ
Longitudinal Current	(`ilg` / 32768) x 101.32 mA
Metallic Current	(`imt` / 32768) x 101.66 mA
Metallic Voltage	(`vsab` / 32768) x 153 V
Battery *N* Voltage	(`vbat`*N* / 32768) x 99.2 V
Metering Signal Peak Voltage	(`mspl` / 32768) x 10.2 V

These results are based on a single instantaneous reading; no filtering is performed. The `ilg` value will fluctuate under the presence of AC induction on the line.

The loop resistance (`rloop`) result is *not valid* in the following states: `VP_LINE_STANDBY`, `VP_LINE_TIP_OPEN`, `VP_LINE_DISCONNECT`, `VP_LINE_RINGING`, and `VP_LINE_RINGING_POLREV`.

The longitudinal (`ilg`) and metallic (`imt`) current results are *not valid* in the following states: `VP_LINE_DISCONNECT`, `VP_LINE_RINGING`, and `VP_LINE_RINGING_POLREV`.

The metallic voltage (vsab) result is *not valid* in the following states: VP_LINE_STANDBY, VP_LINE_TIP_OPEN, and VP_LINE_DISCONNECT.

If the loop conditions measurement is taken during a metering pulse, the mspl field reports the current metering signal peak voltage. Otherwise, the mspl field equals zero.

VpLoopCondResultsType returns the measured battery voltages using the generic battery names vbat1, vbat2, and vbat3. Table 8–3 decodes these generic battery names to device-specific battery names (VBH, VBL, etc.). Note that the interpretation of vBat1 and vBat2 depends on how the device's sense pins are connected to the battery supplies as shown in Table 8–3.

Table 8–3 Battery Name Interpretation

Device Type	Normal Config.		Crossed Config.		
	vBat1	vBat2	vBat1	vBat2	vBat3
CSLAC-790	VBL	VBH	VBH	VBL	VBP

The selectedBat and dcFeedReg results are just enumeration types and require no conversion.

Notes:

An equivalent of **VpGetLoopCond()** *is available for 880 and 890 devices using Microsemi's VeriVoice Line Test Suite.*

DEVICES CSLAC-790

TERMINATIONS FXS

8.5.5 VP_LINE_EVID_GAIN_CMP

DESCRIPTION This event occurs as a result of calling **VpSetRelGain()** and indicates that the transmit and/or receive gain is adjusted. See *VpSetRelGain(), on page 107* for information on that function.

This event is non-maskable.

T.S. OR HANDLE (parmHandle) Handle

EVENT DATA (eventData) N/A

RESULTS (hasResults) Gain adjustment results are returned through the VpRelGainResultsType structure defined below.

```
typedef struct {
    VpGainResultType gResult; /* Success / Failure status return */
    uint16 gxValue;           /* new GX register value */
    uint16 grValue;           /* new GR register value /*
VpRelGainResultsType;
```

The gResult variable indicates whether overflow occurred in the calculation of the transmit or receive gain. If an error does occur, the corresponding gain is automatically restored to the default value from the AC Profile. gResult can have any of the following values:

```
Enumeration Data Type: VpGainResultType:
    VP_GAIN_SUCCESS            /* Gain setting adjusted successfully */
    VP_GAIN_GR_OOR             /* Receive gain overflow, reset to default */
    VP_GAIN_GX_OOR             /* Transmit gain overflow, reset to default */
    VP_GAIN_BOTH_OOR           /* Tx and Rx gain overflow, reset to default */
```

The gxValue and grValue variables return the contents of the VTD gain registers.

DEVICES CSLAC-880, CSLAC-890, CSLAC-580

TERMINATIONS All

8.5.6 VP_DEV_EVID_DEV_INIT_CMP

DESCRIPTION

This event occurs as a result of calling **`VpInitDevice()`** and indicates that VTD initialization is done. See *VpInitDevice(), on page 56* for information on that function.

This event is non-maskable.

T.S. OR HANDLE (`parmHandle`)

N/A

EVENT DATA (`eventData`)

The event data member is set to 1 if the device is successfully initialized.

RESULTS (`hasResults`)

FALSE

DEVICES

All

TERMINATIONS

All

8.5.7 VP_LINE_EVID_LINE_INIT_CMP

DESCRIPTION

This event occurs as a result of calling **`VpInitLine()`** and indicates that the requested line is initialized. See *VpInitLine(), on page 58* for information on that function.

This event is non-maskable.

T.S. OR HANDLE (`parmHandle`)

N/A

EVENT DATA (`eventData`)

N/A

RESULTS (`hasResults`)

FALSE

DEVICES

All

TERMINATIONS

All

8.5.8 VP_DEV_EVID_IO_ACCESS_CMP

DESCRIPTION

This event occurs as a result of calling **`VpDeviceIoAccess()`** and indicates that the requested I/O access is done. See *VpDeviceIoAccess(), on page 112* for information on this function.

This event is non-maskable.

T.S. OR HANDLE (`parmHandle`)

N/A

EVENT DATA (`eventData`)

The event data variable indicates whether the operation was a read (`VP_DEVICE_IO_READ`) or write (`VP_DEVICE_IO_WRITE`).

RESULTS (`hasResults`)

If this event corresponds to an I/O read operation, the application should call **`VpGetResults()`** with a pointer to an appropriate struct, depending on which function call resulted in the event.

For **`VpDeviceIoAccess()`**, a `VpDeviceIoAccessDataType` struct should be passed. **`VpGetResults()`** copies the I/O read results into the `deviceIOData_63_32` and `deviceIOData_31_0` variables of `VpDeviceIoAccessDataType`. *VpDeviceIoAccess(), on page 112* describes how to map the I/O read results in these variables to physical I/O pin logic states.

DEVICES

All

TERMINATIONS

All

8.6 PROCESS EVENTS

The process events report the progress of some process initiated by the application. The set of valid process events is defined in the software by the `VpProcessEventType` enumeration.

8.6.1 VP_LINE_EVID_MTR_CMP

DESCRIPTION	This event occurs as a result of calling **`VpStartMeter()`** and indicates that the metering signal was generated as requested.
T.S. OR HANDLE (`parmHandle`)	N/A
EVENT DATA (`eventData`)	N/A
RESULTS (`hasResults`)	FALSE
DEVICES	All
TERMINATIONS	FXS

8.6.2 VP_LINE_EVID_MTR_ABORT

DESCRIPTION	This event occurs as a result of calling **`VpStartMeter()`** and indicates that the metering signal was aborted before completion.
T.S. OR HANDLE (`parmHandle`)	N/A
EVENT DATA (`eventData`)	Event data returns the number of complete metering pulses transmitted. If a pulse was interrupted it is not included in this count.
RESULTS (`hasResults`)	FALSE
DEVICES	All
TERMINATIONS	FXS

8.6.3 VP_LINE_EVID_CID_DATA

DESCRIPTION

This event indicates that the Caller ID data buffer is either half-empty or empty, depending on the state of the flag attached to the event. This event occurs as a result of calling **VpInitCid()**, **VpSendCid()** or **VpContinueCid()**. See *VpInitCid(), on page 65*, *VpSendCid(), on page 109*, or *VpContinueCid(), on page 110* for information on these functions.

T.S. OR HANDLE (parmHandle)

N/A

EVENT DATA (eventData)

The event data variable indicates whether the VTD can take more Caller ID data or Caller ID transmission is done. This information is passed through the **VpCidDataEventDataType** structure shown below.

```
Enumeration Data Type: VpCidDataEventDataType:
    VP_CID_DATA_NEED_MORE_DATA    /* Caller ID is expecting more data */
    VP_CID_DATA_TX_DONE           /* Caller ID transmission is complete */
```

The VP_CID_DATA_NEED_MORE_DATA event occurs when the VP-API-II can accept 16 bytes of CID message data.

The VP_CID_DATA_TX_DONE event occurs when the CID sequence is complete. It occurs for both normal and abnormal (e.g., off-hook, no ack from CPE for type II CID, etc..) termination of CID.

RESULTS (hasResults)

FALSE

DEVICES

All

TERMINATIONS

FXS

8.6.4 VP_LINE_EVID_RING_CAD

DESCRIPTION

This event occurs when the VP-API-II/VTD performs automatic ringing cadencing. It notifies the application of ringing-on and ringing-off state transitions as the VP-API-II/VTD executes the specified cadence. This event does not occur for any line that does not have a ringing cadence applied to it. See *VpInitRing(), on page 64* for information on applying a ringing cadence to a line. This event also does not occur when the ringing cadence is halted because of a state change or ring-trip.

T.S. OR HANDLE (parmHandle)

Timestamp

EVENT DATA (eventData)

Event data contains a variable of **VpRingCadEventDataType** type, which indicates the type of ringing cadence state change that occurred.

```
Enumeration Data Type: VpRingCadEventDataType:
    VP_RING_CAD_BREAK,            /* Begin OFF period of ringing cadence */
    VP_RING_CAD_MAKE,             /* Begin ON period of ringing cadence */
    VP_RING_CAD_DONE,             /* End of ringing cadence */
```

RESULTS (hasResults)

FALSE

DEVICES

All

TERMINATIONS

FXS

8.6.5 VP_LINE_EVID_SIGNAL_CMP

DESCRIPTION	This event occurs as a result of calling `VpSendSignal()` and indicates that the requested signal was sent. See *VpSendSignal(), on page 108* for information on that function.
T.S. OR HANDLE (parmHandle)	Timestamp, except for `VP_SENDSIG_MOMENTARY_LOOP_OPEN` which indicates '1' if a parallel off-hook is detected, '0' otherwise.
EVENT DATA (eventData)	Event data contains a variable of `VpSendSignalType` type, which indicates the type of signal that was sent. This enumeration type is described with *VpSendSignal(), on page 108*.
RESULTS (hasResults)	N/A
DEVICES	All
TERMINATIONS	All

8.6.6 VP_LINE_EVID_TONE_CAD

DESCRIPTION	This event occurs when the VP-API-II/VTD performs tone cadencing. It notifies the application of completion of the cadence. Completion of the cadence occurs when a cadence reaches an "always on", "always off", or end of cadence.
T.S. OR HANDLE (parmHandle)	Timestamp
EVENT DATA (eventData)	N/A
RESULTS (hasResults)	FALSE
DEVICES	All
TERMINATIONS	All

8.7 FXO EVENTS

The FXO event category includes only those events related to a FXO line. The set of valid FXO events is defined in the software by the `VpFxoEventType` enumeration.

8.7.1 VP_LINE_EVID_RING_ON

DESCRIPTION	This event occurs when ringing is detected on a FXO line.
T.S. OR HANDLE (`parmHandle`)	Timestamp
EVENT DATA (`eventData`)	N/A
RESULTS (`hasResults`)	FALSE
DEVICES	CSLAC-880, CSLAC-890, CSLAC-580
TERMINATIONS	FXO

8.7.2 VP_LINE_EVID_RING_OFF

DESCRIPTION	This event occurs when ringing is no longer detected on a FXO line.
T.S. OR HANDLE (`parmHandle`)	Timestamp
EVENT DATA (`eventData`)	N/A
RESULTS (`hasResults`)	FALSE
DEVICES	CSLAC-880, CSLAC-890, CSLAC-580
TERMINATIONS	FXO

8.7.3 VP_LINE_EVID_LIU

DESCRIPTION	This event indicates that another FXO termination type is using the line (e.g. a parallel telephone went off-hook).
T.S. OR HANDLE (`parmHandle`)	Timestamp
EVENT DATA (`eventData`)	N/A
RESULTS (`hasResults`)	FALSE
DEVICES	CSLAC-880, CSLAC-890
TERMINATIONS	FXO

8.7.4 VP_LINE_EVID_LNIU

DESCRIPTION	This event indicates that another FXO termination type is using the line (e.g. a parallel telephone went off-hook).
T.S. OR HANDLE (`parmHandle`)	Timestamp
EVENT DATA (`eventData`)	N/A
RESULTS (`hasResults`)	FALSE
DEVICES	CSLAC-880, CSLAC-890
TERMINATIONS	FXO

8.7.5 VP_LINE_EVID_FEED_DIS

DESCRIPTION	This event occurs when the FXO no longer detects voltage across the loop. This is similar to the `VP_LINE_EVID_DISCONNECT` event, except that the FXO has opened the loop in this case.
T.S. OR HANDLE (`parmHandle`)	Timestamp
EVENT DATA (`eventData`)	N/A
RESULTS (`hasResults`)	FALSE
DEVICES	CSLAC-880, CSLAC-890, CSLAC-580
TERMINATIONS	FXO

8.7.6 VP_LINE_EVID_FEED_EN

DESCRIPTION	This event occurs when the FXO detects that the FXS has applied a voltage across the loop. This is similar to the `VP_LINE_EVID_RECONNECT` event, except that the FXO has opened the loop in this case.
T.S. OR HANDLE (`parmHandle`)	Timestamp
EVENT DATA (`eventData`)	N/A
RESULTS (`hasResults`)	FALSE
DEVICES	CSLAC-880, CSLAC-890, CSLAC-580
TERMINATIONS	FXO

8.7.7 VP_LINE_EVID_DISCONNECT

DESCRIPTION	This event occurs when the FXO no longer detects loop current from the FXS. This is similar to the `VP_LINE_EVID_FEED_DIS` event, except that the FXO has closed the loop in this case.
T.S. OR HANDLE (`parmHandle`)	Timestamp
EVENT DATA (`eventData`)	N/A
RESULTS (`hasResults`)	FALSE
DEVICES	CSLAC-880, CSLAC-890, CSLAC-580
TERMINATIONS	FXO

8.7.8 VP_LINE_EVID_RECONNECT

DESCRIPTION	This event occurs when the FXO detects that the FXS has started driving loop current. This is similar to the `VP_LINE_EVID_FEED_EN` event, except that the FXO has closed the loop in this case.
T.S. OR HANDLE (`parmHandle`)	Timestamp
EVENT DATA (`eventData`)	N/A
RESULTS (`hasResults`)	FALSE
DEVICES	CSLAC-880, CSLAC-890, CSLAC-580
TERMINATIONS	FXO

8.7.9 VP_LINE_EVID_POLREV

DESCRIPTION	This event occurs when the FXO port detects loop voltage polarity reversal.
T.S. OR HANDLE (`parmHandle`)	Timestamp
EVENT DATA (`eventData`)	’0’ if the polarity transition was from reverse to normal, ’1’ if the polarity transition was from normal to reverse.
RESULTS (`hasResults`)	FALSE
DEVICES	All except CSLAC-790
TERMINATIONS	FXO

8.7.10 VP_LINE_EVID_POH

DESCRIPTION	This event occurs when the FXO port detects a parallel phone going off-hook while providing a loop close state. This event is not generated while the FXO line is providing a loop open. Generation of this event forces a change of the FXO line status `VP_INPUT_LIU` = `TRUE`. See Section 10.2.2 for details.
T.S. OR HANDLE (`parmHandle`)	Timestamp
EVENT DATA (`eventData`)	N/A
RESULTS (`hasResults`)	FALSE
DEVICES	CSLAC-880, CSLAC-890, CSLAC-580
TERMINATIONS	FXO

8.7.11 VP_LINE_EVID_PNOH

DESCRIPTION	This event occurs when the FXO port detects a parallel phone going on-hook while providing a loop close state. This event is not generated while the FXO line is providing a loop open. Generation of this event forces a change of the FXO line status `VP_INPUT_LIU` = `FALSE`. See Section 10.2.2 for details.
T.S. OR HANDLE (`parmHandle`)	Timestamp
EVENT DATA (`eventData`)	N/A
RESULTS (`hasResults`)	FALSE
DEVICES	CSLAC-880, CSLAC-890, CSLAC-580
TERMINATIONS	FXO

8.8 TEST EVENTS

8.8.1 VP_LINE_EVID_TEST_CMP

DESCRIPTION	The test events occur as a result of the Application calling a Line Test API function. The Application only needs to pass this event to the Line Test API. This event is non-maskable.
T.S. OR HANDLE (`parmHandle`)	N/A
EVENT DATA (`eventData`)	N/A
RESULTS (`hasResults`)	Managed by the Line Test Software Packages. The Application only needs to handle results generated by the Line Test API.
DEVICES	CSLAC-880, CSLAC-890
TERMINATIONS	FXS

CHAPTER

9 CONTROL FUNCTIONS

9.1 OVERVIEW

This chapter covers VP-API-II functions that primarily control the VTD. The following control functions are described in this chapter:

- `VpFreeRun()` – Puts the device/lines into and takes it out of PLL Free Run mode.
- `VpSetLineState()` – Sets a line to the requested state.
- `VpSetRelayState()` – Sets the line relay configuration.
- `VpSetOption()` – Sets various device and line specific options.
- `VpSetLineTone()` – Generates a cadenced call progress tone on a FXS line.
- `VpSetRelGain()` – Sets the relative transmit or receive gain for a line.
- `VpStartMeter()` – Starts metering on a FXS line.
- `VpSendSignal()` – Generates message waiting pulse on FXS lines, or pulse and DTMF digits on FXO lines.
- `VpSendCid()` – Starts a Caller ID sequence on a FXS line without waiting for a ring state change.
- `VpContinueCid()` – Refreshes the Caller ID buffer for a FXS line during message transmission.
- `VpDtmfDigitDetected()` – Reports a DTMF digit detected outside the scope of the VP-API. Used for implementing Type-II Caller ID.
- `VpDeviceIoAccess()` – Controls device input/output pins.

9.2 FUNCTION DESCRIPTIONS

9.2.1 VpFreeRun()

SYNTAX

```
VpStatusType
VpFreeRun(
 VpDevCtxType *pDevCtx,            /* Pointer to device context */
 VpFreeRunModeType freeRunMode)    /* Free run start/stop flag */
```

DESCRIPTION

This function puts the device (and it's associated lines) into PLL Free Run mode and takes it out of PLL Free Run mode depending on the **`VpFreeRunModeType`** value passed:

```
Enumeration Data Type: VpFreeRunModeType:
 VP_FREE_RUN_START/* Puts the device/lines into PLL Free Run Mode */
 VP_FREE_RUN_STOP /* Takes the device/lines out of PLL Free Run Mode */
```

PLL Free Run occurs when PCLK is removed (i.e., Clock Fault is detected). It is defined as a mode where the PLL is running without an accurate clock, causing the PLL to run at a reduced rate -- up to 5x slower than normal. This rate reduction has a direct affect on switcher and line performance that should be avoided if possible.

If a PCLK fault can be anticipated and to avoid the affects of the PLL rate reduction, the application should call this function with `freeRunMode = VP_FREE_RUN_START` prior to PCLK removal. Note however that once called with `freeRunMode = VP_FREE_RUN_START`, the device will remain in Free Run mode until this function is called with `freeRunMode = VP_FREE_RUN_STOP` or until **`VpInitDevice()`** is called.

RETURNS

See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED

None

DEVICES

CSLAC-880, CSLAC-890

TERMINATIONS

All

9.2.2 VpSetLineState()

SYNTAX

```
VpStatusType
VpSetLineState(
 VpLineCtxType *pLineCtx,            /* Pointer to line context */
 VpLineStateType state)              /* Selects the desired line state */
```

DESCRIPTION

This function sets the line associated with `pLineCtx` to the requested line state. All valid line state arguments of the VP-API-II are listed below, but note that not all devices or termination types will be able to support all VP-API-II line states. If a line state cannot be supported the VP-API-II will return `VP_STATUS_INVALID_ARG`.

Enumeration Data Type: VpLineStateType

FXS Line States	
VP_LINE_STANDBY	/* Normal Polarity Feed from High Battery. A/D and D/A Converters * Disabled. PCM Path Disabled. Used for on-hook conditions only. */
VP_LINE_STANDBY_POLREV	/* Reverse Polarity Feed from High Battery. A/D and D/A Converters * Disabled. PCM Path Disabled. Used for on-hook conditions only. */
VP_LINE_TIP_OPEN	/* Battery Feed on Ring Lead, Tip lead in High Impedance. A/D and * D/A Converters Disabled. PCM Path Disabled. */
VP_LINE_RING_OPEN	/* Battery Feed on Ring Lead, Tip lead in High Impedance. A/D and * D/A Converters Disabled. PCM Path Disabled. */
VP_LINE_ACTIVE	/* Normal Polarity Feed. Battery used depends on DC loop resistance. * A/D and D/A Converters Enabled. PCM Path Disabled. */
VP_LINE_ACTIVE_POLREV	/* Reverse Polarity Feed. Battery used depends on DC loop resistance. * A/D and D/A Converters Enabled. PCM Path Disabled. */
VP_LINE_TALK	/* Normal Polarity Feed. Battery used depends on DC loop resistance. * A/D and D/A Converters Enabled. PCM Path Enabled. */
VP_LINE_TALK_POLREV	/* Reverse Polarity Feed. Battery used depends on DC loop resistance. * A/D and D/A Converters Enabled. PCM Path Enabled. */
VP_LINE_OHT	/* Normal Polarity Feed from High Battery. A/D and D/A Converters * Enabled. PCM Path Enabled. Used for on-hook conditions only. */
VP_LINE_OHT_POLREV	/* Reverse Polarity Feed from High Battery. A/D and D/A Converters * Enabled. PCM Path Enabled. Used for on-hook conditions only. */
VP_LINE_DISCONNECT	/* Tip and Ring in High Impedance conditions. A/D and D/A Converters * Disabled. PCM Path Disabled. Service denial. */
VP_LINE_RINGING	/* Feed Conditions per last provided Ringing and Ringing Cadence * Profile. DC Bias is defined as (Tip-Ring). PCM Path is disabled * while Ringing Signal is being driven on the customer line. */
VP_LINE_RINGING_POLREV	/* Feed Conditions per last provided Ringing and Ringing Cadence * Profile. DC Bias is defined as (Ring - Tip). PCM Path is disabled * while Ringing Signal is being driven on the customer line. */
VP_LINE_HOWLER	/* Normal Polarity Feed. Battery used depends on DC loop resistance. * A/D and D/A Converters Enabled. D-A Path Gain increase by 11.5dB * compared to a 0dBr D-A setting. PCM Path Disabled. */
VP_LINE_HOWLER_POLREV	/* Reverse Polarity Feed. Battery used depends on DC loop resistance. * A/D and D/A Converters Enabled. D-A Path Gain increase by 11.5dB. * compared to a 0dBr D-A setting. PCM Path Disabled. */
FXO Line States	
VP_LINE_FXO_OHT	/* FXO providing Loop Open to the line. A/D and D/A Converters * Enabled. PCM Path Enabled */
VP_LINE_FXO_LOOP_OPEN	/* FXO providing Loop Open to the line. A/D and D/A Converters * Disabled. PCM Path Disabled. */
VP_LINE_FXO_LOOP_CLOSE	/* FXO providing Loop Close to the line. A/D and D/A Converters * Enabled. PCM Path Enabled */
VP_LINE_FXO_TALK	/* FXO providing Loop Close to the line. A/D and D/A Converters * Disabled. PCM Path Disabled. */

The behavior of the line in states `VP_LINE_RINGING` and `VP_LINE_RINGING_POLREV` is defined by the last Ringing Cadence Profile passed to **`VpInitRing()`**. If no Ringing Cadence Profile has been provided or the last profile was `VP_NULL`, the Ringing Cadence behavior is "Always On".

In addition to the Ringing Profile and Ringing Cadence Profile, parameters affecting ring entry and exit can be specified by calling **`VpSetOption()`** with *VP_OPTION_ID_RING_CNTRL, on page 26*.

When using VP_LINE_HOWLER or VP_LINE_HOWLER_POLREV, the previous line state must be one of VP_LINE_ACTIVE, VP_LINE_TALK, VP_LINE_ACTIVE_POLREV, or VP_LINE_TALK_POLREV. When exiting VP_LINE_HOWLER or VP_LINE_HOWLER_POLREV, the line must be set to the state it was in prior to entering the Howler state. The following legal and illegal sequences are used to clarify this requirement:

```
VP_LINE_ACTIVE->VP_LINE_HOWLER->VP_LINE_ACTIVE   /* ok */
VP_LINE_ACTIVE->VP_LINE_HOWLER->VP_LINE_TALK     /* illegal */
VP_LINE_STANDBY->VP_LINE_HOWLER                  /* illegal */
```

When an illegal sequence is encountered, this function will return VP_STATUS_INVALID_ARG. Applications that use VP_LINE_HOWLER or VP_LINE_HOWLER_POLREV should be written to check the return value from this function to avoid unexpected behavior.

Notes:

1. *VpInitDevice() and VpInitLine() places all FXS lines in the VP_LINE_DISCONNECT state. The application must call this function after initialization to enable service to the customer.*
2. *All VE880-FXS and VE890-FXS lines **must be calibrated** using either VpCalLine()* or VpCal() prior to being used for normal operation. Calibration is required to meet Data Sheet Specifications. A line that is not calibrated may experience hook detect and ring trip issues, and will in general consume more power while both on and off-hook.
3. *A new line state request by the application is assumed to be a higher priority than any other currently set state or conflicting cadence. Any conflicting cadence running at the time VpSetLineState() is called will be terminated. This includes, but not limited to, Caller ID, Ringing Cadence, Metering, and signals generated by VpSendSignal(). Tone cadences will continue provided that the state being entered is one with the D/A converter enabled.*
4. *If metering is active the current meter pulse will continue through to completion. After the current pulse is complete, metering will then terminate without generating any of the remaining meter pulses. Metering pulses defined as "always on" will be terminated immediately.*

RETURNS See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED None.

DEVICES All

TERMINATIONS All

9.2.3 VpSetRelayState()

SYNTAX

```
VpStatusType
VpSetRelayState(
 VpLineCtxType *pLineCtx,            /* Pointer to line context */
 VpRelayControlType rState)          /* Relay state */
```

DESCRIPTION

This function configures the VTD-controlled relays. Depending on the line termination type, the VTD may control some combination of LCAS, electro-mechanical relay, and test load. The line circuit configuration determines the allowable parameters for `rState`. This function returns an error if the `rState` argument tries to control a relay that is not included in the reference design circuit (as specified in the call to *VpMakeLineObject()*, on page 52 through the line termination type). The relay states are listed below.

```
Enumeration Data Type: VpRelayControlType:
    VP_RELAY_NORMAL
    VP_RELAY_RESET
    VP_RELAY_TESTOUT
    VP_RELAY_TALK
    VP_RELAY_RINGING
    VP_RELAY_TEST
    VP_RELAY_BRIDGED_TEST
    VP_RELAY_SPLIT_TEST
    VP_RELAY_DISCONNECT
    VP_RELAY_RINGING_NOLOAD
    VP_RELAY_RINGING_TEST
```

The `VP_RELAY_NORMAL` state allows for normal VTD control of the LCAS or relays), according to the current line state selected by the **`VpSetLineState()`** function and any fault condition detected. Selection of any other relay state overrides the automatic VTD relay control. The LCAS, EMR, and test load are forced into the desired state without consideration for the current SLIC/SLAC state.

The appendix *Relay Configurations*, on page 143 describes simple connection diagrams for applicable relay states for various line termination types.

RETURNS

See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

All

9.2.4 VpSetOption()

SYNTAX

```
VpStatusType
VpSetOption(
 VpLineCtxType *pLineCtx,          /* Pointer to line context */
 VpDevCtxType *pDevCtx,            /* Pointer to the Device Object */
 VpOptionIdType option,            /* Selects the option to modify */
 void *pValue)                     /* Pointer to the option's new value */
```

DESCRIPTION

This function sets an option to the specified value for one or more lines. The `option` argument determines which option is modified. Options may be line-specific or device-specific. Refer to Chapter 3, on page 17 for a complete list and definition of all VP-API-II options.

This function acts at only the device level, only on the particular line, or on all lines across the device depending on the values of the device context, line context, and option arguments. The table below summarizes this behavior. The "Option" column indicates whether the target option is device-specific or line-specific. The "Device Ctx" and "Line Ctx" columns indicate whether a valid pointer or `VP_NULL` is passed for the `pDevCtx` and `pLineCtx` parameters, respectively

Table 9–1 `VpSetOption()` Behavior

Option	Device Ctx	Line Ctx	Result
device	VP_NULL	VP_NULL	returns VP_STATUS_INVALID_ARG
device	VP_NULL	valid	returns VP_STATUS_INVALID_ARG
device	valid	VP_NULL	sets option for the specified device
device	valid	valid	returns VP_STATUS_INVALID_ARG
line	VP_NULL	VP_NULL	returns VP_STATUS_INVALID_ARG
line	VP_NULL	valid	sets option for the specified line
line	valid	VP_NULL	sets option for all lines of the specified device
line	valid	valid	returns VP_STATUS_INVALID_ARG

Notice that device-specific options apply to all lines controlled by the device, regardless of whether a line context or a device context argument is given in the call to **`VpSetOption()`**.

The arguments required for each option are different. Therefore, a `void` pointer (`pValue`) is provided as the option input parameter to the function. This argument must point to an initialized instance of the input structure related to the target option. For example, if the device critical fault options are being set, then `pValue` must point to an initialized instance of `VpOptionCriticalFltType`. Chapter 3, on page 17 describes the input parameter type for each option.

All options are set to their default values after the device is initialized as a result of calling the **`VpInitDevice()`** function (see *VpInitDevice(), on page 56*). Therefore, options should only be changed after device initialization is complete, as indicated by the `VP_DEV_EVID_DEV_INIT_CMP` event. Line-specific options are also reset as a result of calling the **`VpInitLine()`** function. The application should set these line-specific options to their desired values after line initialization is complete, as indicated by the `VP_LINE_EVID_LINE_INIT_CMP` event.

RETURNS

See *VP-API-II Function Return Type, on page 11*

EVENTS GENERATED

None.

DEVICES

All

TERMINATIONS

All

9.2.5 VpSetLineTone()

SYNTAX

```
VpStatusType
VpSetLineTone(
 VpLineCtxType *pLineCtx,          /* Pointer to line context */
 VpProfilePtrType
   pToneProfile,                   /* Pointer to Tone Profile */
 VpProfilePtrType
   pCadProfile,                    /* Pointer to Tone Cadence Profile */
 VpDtmfToneGenType
   *pDtmfControl)                  /* Pointer to DTMF control structure */
```

DESCRIPTION

This function starts a call progress tone with an optional cadence on the line associated with `pLineCtx`. This function can also generate DTMF tones.

The generated tone is defined by the Tone Profile at `pToneProfile`. If `pToneProfile` is `VP_PTABLE_NULL`, then any currently active tone is stopped.

The cadence (on/off sequence) applied to the tone is defined by the Cadence Profile at `pCadProfile`. If `pCadProfile` is `VP_PTABLE_NULL`, then the specified tone is played continuously until turned-off with a subsequent call to **`VpSetLineTone()`**.

The argument `pDtmfControl` controls DTMF tone generation. If this argument is `VP_NULL`, no action is performed for DTMF tone generation. DTMF tone generation occurs only if no Tone Profile is specified (`pToneProfile` is `VP_PTABLE_NULL`) and `pDtmfControl` argument is not `VP_NULL`. The DTMF control structure is defined below.

```
Enumeration Data Type: VpDigitType:
    1 to 9                        /* Digits 1 to 9; No constants defined for this */
    VP_DIG_ZERO                   /* Digit 0 */
    VP_DIG_ASTER                  /* "*" key on the telephone keypad */
    VP_DIG_POUND                  /* "#" key on the telephone keypad */
    VP_DIG_A                      /* "A" key on the telephone keypad */
    VP_DIG_B                      /* "B" key on the telephone keypad */
    VP_DIG_C                      /* "C" key on the telephone keypad */
    VP_DIG_D                      /* "D" key on the telephone keypad */
    VP_DIG_NONE                   /* Stop digit generation */

Enumeration Data Type: VpDirectionType:
    VP_DIRECTION_DS               /* Tone generation in D-A direction */
    VP_DIRECTION_US               /* Tone generation in A-D direction */

typedef struct {
    VpDigitType toneId,           /* The requested DTMF tone */
    VpDirectionType dir,          /* DTMF tone generation direction */
} VpDtmfToneGenType;
```

Generating Special Howler Tones:

- Special Howler Tones are those country specific howler tones that require frequency and/or amplitude changes during tone generation. The special howler tones supported in the VP-API-II are:

1. *UK: BTNR 1080 Version 15 and Draft 960-G. The BTNR 1080 version used by the VP-API-II is selectable at compile time in file vp_api_cfg.h by setting the value* `VP_UK_HOWLER_IN_USE` *to either* `VP_UK_HOWLER_BTNR_VER15` *or* `VP_UK_HOWLER_BTNR_DRAFT_G` *as follows:*

   For BTNR 1080 Version 15 compliance set as (default):

   ```
   #define VP_UK_HOWLER_IN_USE    VP_UK_HOWLER_BTNR_VER15
   ```

   For BTNR 1080 Draft 960-G compliance set as:

   ```
   #define VP_UK_HOWLER_IN_USE    VP_UK_HOWLER_BTNR_DRAFT_G
   ```

2. *AUS: Australia ACIF S002:2001*
3. *NTT: NTT Edition 5*

- The special howler tones (UK, AUS, NTT) are selected from the Tone Cadence profile menu in Profile Wizard. When generating a special howler tone from the tone cadence profile `pToneProfile` must be set to a value other than `VP_PTABLE_NULL`. The VP-API-II treats the argument conditions: [`pToneProfile == VP_PTABLE_NULL, pDtmfControl == VP_NULL`, `pCadProfile == don't care`] to mean disable all tones. Setting `pToneProfile` to a valid tone profile pointer will allow the VP-API-II to proceed past this check and generate the special howler tone pointed to by `pCadProfile`.

Notes:

1. *Call progress tone and DTMF tone generation cannot be performed simultaneously.*
2. *DTMF tone generation is supported in the D-A direction only.*
3. *Special Howler Tones are supported on VE880 and VE890 only.*
4. *When generating the NTT Howler Tone, the total level increase will be over the duration of [14 to 15] seconds. Note that the NTT Edition 5 Specification allows tone level increase duration of [3 to 15] seconds.*
5. *When used with line states `VP_LINE_HOWLER` or `VP_LINE_HOWLER_POLREV` the maximum output level of the special howler tones is: UK (14.5dBm), NTT (14.5dBm), AUS (10.0dBm) when measured into a 600ohm load. When used with talk line states, the maximum output level is +3dBm using a standard AC Profile and setting the maximum receive (D-A) gain of 0dBr.*

RETURNS See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED *VP_LINE_EVID_TONE_CAD*, on page 92

DEVICES All

TERMINATIONS All

9.2.6 VpSetRelGain()

SYNTAX

```
VpStatusType
VpSetRelGain(
 VpLineCtxType *pLineCtx,        /* Pointer to line context */
 uint16 txLevel,                 /* Relative adjustment to Tx level */
 uint16 rxLevel,                 /* Relative adjustment to Rx level */
 uint16 handle)                  /* Handle returned with event */
```

DESCRIPTION

This function adjusts the transmit and receive gain for the specified line. The gain adjustment is defined as a voltage gain (multiplier) made relative to the gain levels set in the AC Profile applied to the line. Setting the `txLevel` or `rxLevel` to 1.0 sets the respective path to the default gain from the AC Profile.

The transmit and receive relative gain settings are specified as 2.14 fixed-point unsigned numbers with a range of 0 to 4.0. Keeping in mind these are voltage gains, converting from a target dB change to `txLevel` and `rxLevel` values can be done as follows:

Generic Equation:

```
dB = 20log(level)
```

converting for `level` results in the equation:

```
level = 10^(dB/20)
```

From the equation: `level = 10^(dB/20)` scaling to 2.14 format requires multiplying `level` by 16,384 (0x4000). The final result can then be applied to either `txLevel` or `rxLevel`.

Walk-through examples (`txLevel` can be replaced with `rxLevel` in the following cases):

- Target = -3dB, then `txLevel` = 10^(-3/20) = ~0.70795. Scale to 2.14 format requires multiplication by 16,384 resulting in (`txLevel` = 0.70795 * 16,384 = 11,599).
- Target = -2dB, then `txLevel` = 10^(-2/20) = ~0.79433. Scale to 2.14 format requires multiplication by 16,384 resulting in (`txLevel` = 0.79433 * 16,384 = 13,014).
- Target = +2dB, then `txLevel` = 10^(2/20) = ~1.25893. Scale to 2.14 format requires multiplication by 16,384 resulting in (`txLevel` = 1.25893 * 16,384 = 20,626).

Note that a negative gain is a relative loss (e.g., -3dB, -2dB).

The amount of adjustment possible depends on the zero transmission level point (0 TLP) set in the AC Profile. The application can be coded to know that the 0 TLP allows for a guaranteed adjustment range, or it can get the default gain level by setting the `txLevel` and `rxLevel` inputs to 1.0 and compute the available adjustment range using the data returned with the `VP_LINE_EVID_GAIN_CMP` event.

The `VP_LINE_EVID_GAIN_CMP` event occurs once the VTD has applied the new gain settings. The results of this function are described in Section 8.5.5 with the definition of this event.

RETURNS

See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED

VP_LINE_EVID_GAIN_CMP, on page 88

DEVICES

CSLAC-880, CSLAC-890, CSLAC-580

TERMINATIONS

All

9.2.7 VpSendSignal()

SYNTAX

```
VpStatusType
VpSendSignal(
 VpLineCtxType *pLineCtx,            /* Pointer to line context */
 VpSendSignalType signalType,        /* Specifies the type of signal */
 void *pSignalData)                  /* Specifies signal parameters */
```

DESCRIPTION

This function generates a signal on the specified line. The following types of signals are defined:

```
Enumeration Data Type: VpSendSignalType:
    VP_SENDSIG_MSG_WAIT_PULSE   /* Send message waiting signal (FXS only)*/
    VP_SENDSIG_DTMF_DIGIT       /* Generate DTMF digit (FXO only) */
    VP_SENDSIG_PULSE_DIGIT      /* Generate pulse digit (FXO only) */
    VP_SENDSIG_HOOK_FLASH       /* Generate hook flash (FXO only) */
    VP_SENDSIG_FWD_DISCONNECT   /* Generate a Forward Disconnect (FXS only) */
    VP_SENDSIG_POLREV_PULSE     /* Generate a Polarity Reversal (FXS only) */
    VP_SENDSIG_MOMENTARY_LOOP_OPEN/* Execute Momentary Extension Check on FXO */
    VP_SENDSIG_TIP_OPEN_PULSE   /* Generate a Tip Open Pulse (FXS only) */
```

For each of the above signal types, the `pSignalData` argument points to an initialized instance of a structure or a variable that defines the signal.

When sending a message waiting signal (`VP_SENDSIG_MSG_WAIT_PULSE`), `pSignalData` must point to a **`VpSendMsgWaitType`** instance. The **`VpSendMsgWaitType`** structure is defined as follows:

```
typedef struct {
    int8 voltage,               /* Voltage (Volts) applied to the line. A
                                 * negative value means Tip is more negative than
                                 * Ring, a positive value means Ring is more
                                 * negative than Tip. */
    uint16 onTime,              /* Duration of pulse on-time in mS. If the
                                 * on-time is 0 it stops an ongoing message
                                 * waiting signal generation. */
    uint16 offTime,             /* Duration of pulse off-time in mS. If the
                                 * off-time is set to 0, the voltage is applied
                                 * to the line continuously. */
    uint8 cycles,               /* Number of pulses to send on the line. If set
                                 * to 0, will repeat forever. */
} VpSendMsgWaitType;
```

When sending a DTMF or pulse digit (`VP_SENDSIG_DTMF_DIGIT` or `VP_SENDSIG_PULSE_DIGIT`), `pSignalData` must point to a **`VpDigitType`** instance. See *VpSetLineTone(), on page 105* for the definition of **`VpDigitType`**.

The parameter `pSignalData` is ignored in case of `VP_SENDSIG_HOOK_FLASH`.

When sending a Forward Disconnect, `pSignalData` must point to a uint16 instance specifying the time in the disconnect state in milli-seconds. No hook events will occur when entering disconnect, while in disconnect, and for 100ms after recovery from the disconnect state.

When sending a Polarity Reversal, `pSignalData` must point to a `uint16` instance specifying the time in the polarity reversal state in milli-seconds. No hook events will occur for 100ms after changing the polarity state. The specific state used for Polarity Reversal is the reverse polarity of the current state (i.e., if currently in `VP_LINE_TALK_POLREV`, then Polarity Reversal will be `VP_LINE_TALK`).

When sending a Tip Open Pulse, `pSignalData` must point to a uint16 instance specifying the time in the tip open state in milli-seconds. Ground key and/or hook events may occur while performing Tip Open Pulse Send Signal and after the 100ms recovery time.

The parameter `pSignalData` is ignored in case of `VP_SENDSIG_MOMENTARY_LOOP_OPEN`. The FXO applies a loop open for >=10ms and detects a T/R voltage of <=16V. If the voltage is <=16V, a parallel off-hook is reported in the event `VP_LINE_EVID_SIGNAL_CMP` by setting the value of `parmHandle` to '1', otherwise `parmHandle` is set to '0'.

If the `pSignalData` argument is `VP_NULL` then a Message Waiting Pulse, DTMF Digit, Pulse Digit, or Hook Flash type `signalType` is immediately stopped.

The `VP_LINE_EVID_SIGNAL_CMP` event occurs when signal generation is done.

Notes:

1. *The* `VP_SENDSIG_POLREV_PULSE`, *and* `VP_SENDSIG_MOMENTARY_LOOP_OPEN` *signal types are supported for the CSLAC-880 and CSLAC-890 devices only.*
2. *The* `VP_SENDSIG_FWD_DISC` *signal types are supported for the CSLAC-880 and CSLAC-890 devices only.*
3. *The* `VP_SENDSIG_TIP_OPEN` *signal type is supported for the CSLAC-880 devices only.*

RETURNS See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED *VP_LINE_EVID_SIGNAL_CMP*, on page 92

DEVICES All

TERMINATIONS All

9.2.8 VpSendCid()

SYNTAX

```
VpStatusType
VpSendCid(
 VpLineCtxType *pLineCtx,          /* Pointer to line context */
 uint8 length,                     /* Length of the CID data to send */
 VpProfilePtrType
  pCidProfile,                     /* Pointer to Caller ID Profile */
 uint8p pCidData)                  /* Pointer to Caller ID data */
```

DESCRIPTION `VpSendCid()` transmits Caller ID data on-demand. This function differs from `VpInitCid()` in that `VpInitCid()` sends Caller ID data automatically during the ringing cadence. This function enables off-hook Caller ID, also known as *Type-II* or *Call Waiting* Caller ID. `VpSendCid()` is a more flexible method of sending Caller ID than `VpInitCid()`.

The `length` argument should specify the total length in bytes of the entire message to be transmitted if the message length is less than or equal to 32 bytes, otherwise it should be set to 32. Since Caller ID messages can be longer than 32 bytes, the application may need to make several VP-API-II function calls to transmit a complete Caller ID message. To facilitate this, the VP-API-II generates the `VP_LINE_EVID_CID_DATA` event (with `eventData` equal to `VP_CID_DATA_NEED_MORE_DATA`) when the VP-API-II can accept 16 bytes of Caller ID data. Upon receiving this event, the application must call `VpContinueCid()` to buffer any remaining Caller ID data. If this function is called with `length` less than or equal to 16 bytes, then the VP-API-II assumes that the Caller ID message is not longer than 16 bytes and therefore does not generate the `VP_CID_DATA_NEED_MORE_DATA` event.

The `pCidProfile` argument selects the desired Caller ID Profile. The timing information present in the Caller ID Profile, such as the relationship between the start of Caller ID transmission and the ringing cadence, is not applicable to this function and is ignored.

The `pCidData` argument should point to a buffer containing the Caller ID message. Refer to *VpInitCid()*, on page 65 for move information on handling the Caller ID data.

RETURNS See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED *VP_LINE_EVID_CID_DATA*, on page 91

DEVICES All except CSLAC-580

TERMINATIONS FXS

9.2.9 VpContinueCid()

SYNTAX

```
VpStatusType
VpContinueCid(
 VpLineCtxType *pLineCtx,          /* Pointer to line context */
 uint8 length                      /* Length of Caller ID data */
 uint8p pCidData)                  /* Pointer to the Caller ID data */
```

DESCRIPTION

This function is necessary for both Type-I and Type-II Caller ID implementations.

This function loads the VP-API-II CID message data buffer with `length` number of bytes, but cannot exceed 16 in a single call to **`VpContinueCid()`** (the value of `length` should always be <= 16).

This function is called by the application when there is message data to send to the line (remaining data after calling **`VpInitCid()`** and **`VpSendCid()`** as appropriate with the first 32 bytes), AND in response to event `VP_LINE_EVID_CID_DATA:VP_CID_DATA_NEED_MORE_DATA`.

The event `VP_LINE_EVID_CID_DATA:VP_CID_DATA_NEED_MORE_DATA` is generated each time the VP-API-II can accept 16 more bytes of message data and will continue to be generated each time **`VpContinueCid()`** is called with more message data. The current CID message ends when the application DOES NOT call **`VpContinueCid()`** in response to event `VP_LINE_EVID_CID_DATA:VP_CID_DATA_NEED_MORE_DATA`. In this scenario, the previously remaining message data followed by the checksum (if indicated in the caller id profile) is sent, and the event `VP_LINE_EVID_CID_DATA:VP_CID_DATA_TX_DONE` is generated (indicating that CID has ended). Note that `VP_LINE_EVID_CID_DATA:VP_CID_DATA_TX_DONE` is also generated when CID aborts.

RETURNS

See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED

VP_LINE_EVID_CID_DATA, on page 91

DEVICES

All except CSLAC-580

TERMINATIONS

FXS

9.2.10 VpStartMeter()

SYNTAX

```
VpStatusType
VpStartMeter(
 VpLineCtxType *pLineCtx,          /* Pointer to line context */
 uint16 onTime,                    /* Pulse on time in 10ms increments */
 uint16 offTime,                   /* Pulse off time in 10ms increments */
 uint16 numMeters)                 /* Number of meter cycles to perform */
```

DESCRIPTION

This function starts metering pulses on the line associated with the line context argument `pLineCtx`. The metering behavior is defined by `onTime`, `offTime`, and `numMeters`, which defines the on/off timing of each meter pulse. The `numMeters` argument determines the number of pulses generated. This function assumes that the user has specified the Metering Pulse Profile using the **`VpInitMeter()`** function. See *VpInitMeter()*, on page 65. If **`VpInitMeter()`** is not called sometime prior to this function, then the VTD default meter parameters are used. If `numMeters` is zero, then any active metering sequence is terminated. If `onTime` is zero and `numMeters` is non-zero, then an infinite metering signal is played until this function is called with `numMeters` equal to zero. This allows the host to control the on/off cadence if desired.

RETURNS

See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED

VP_LINE_EVID_MTR_CMP, on page 90
VP_LINE_EVID_MTR_ABORT, on page 90

DEVICES

All

TERMINATIONS

FXS

9.2.11 VpDtmfDigitDetected()

SYNTAX

```
VpStatusType
VpDtmfDigitDetected(
 VpLineCtxType *pLineCtx,          /* Pointer to line context */
 VpDigitType digit,                /* The DTMF digit that was detected */
 VpDigitSenseType sense)           /* Indicates start/end of digit */
```

DESCRIPTION

This function is used in Type-II Caller ID when the Caller ID sequence reaches a *Detect Interval* command. The "Detect Interval" command is a special command in the Caller ID Profile that indicates to the VP-API-II to detect an acknowledgement from the CPE in the form of a DTMF digit before starting Type-II Caller ID transmission.

For CSLAC devices, the VP-API-II does not implement DTMF detection; this function along with **VpSysDtmfDetEnable()** and **VpSysDtmfDetDisable()** system services functions provide a mechanism to use a DTMF decoder that might be available outside the scope of the VP-API-II. When the VP-API-II is transmitting a Type-II CID and it reaches the Detect Interval command, the **VpSysDtmfDetEnable()** function is called indicating that DTMF digit decoding should be enabled. When the Detect Interval command completes, **VpSysDtmfDetDisable()** is called indicating that the VP-API-II does not need the services of the DTMF decoder anymore. See *VpSysDebugPrintf()*, on page 126 for further information about these functions.

During the Detect Interval **VpDtmfDigitDetected()** should be called to notify the VP-API-II that a DTMF digit was detected on the specified line. This function should only be called when the leading and the trailing edge of the DTMF digit are detected. The DTMF digits received during this interval are analyzed by the VP-API-II to detect Type-II Caller ID acknowledgement from the CPE. The `digit` argument indicates which DTMF digit was detected and is of the **VpDigitType** type described in *VpSetLineTone()*, on page 105.

The `sense` argument specifies whether the start (leading edge) of the DTMF digit was detected or the end (trailing edge) of the DTMF digit was detected. This enumeration type is defined below.

```
Enumeration Data Type: VpDigitSenseType:
    VP_DIG_SENSE_BREAK          /* Trailing edge of the DTMF digit */
    VP_DIG_SENSE_MAKE           /* Leading edge of the DTMF digit */
```

This function generates the `VP_LINE_EVID_DTMF_DIG` event to notify the application software that a digit was detected. This allows the application layer to receive DTMF digit events from the VP-API-II even though the VP-API-II does not actually perform the DTMF digit decoding.

Note that this function can be called any time that DTMF digits are detected. The DTMF digits passed into this function are reported to the application through the event mechanism described above. If such an operation is desired, the **VpSysDtmfDetEnable()** and **VpSysDtmfDetDisable()** implementations should be modified such that DTMF decoding is always enabled.

RETURNS

See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED

VP_LINE_EVID_DTMF_DIG, on page 84

DEVICES

CSLAC

TERMINATIONS

FXS

9.2.12 VpDeviceIoAccess()

SYNTAX

```
VpStatusType
VpDeviceIoAccess(
 VpDevCtxType *pDevCtx,        /* Pointer to device context */
 VpDeviceIoAccessDataType
  *pDeviceIoData)              /* Pointer to I/O access control struct */
```

DESCRIPTION

This function accesses the device input/output (I/O) pins of the VTD. Refer to **VP_DEVICE_OPTION_ID_DEVICE_IO** for more information on I/O pin configuration and restrictions. This function takes a pointer to the following structure type:

```
typedef struct {
    VpDeviceIoAccessType accessType;/* Device I/O access type */
    uint32 accessMask_31_0;         /* I/O access mask (Pins 0 - 31) */
    uint32 accessMask_63_32;        /* I/O access mask (Pins 32 - 63) */
    uint32 deviceIOData_31_0;       /* Output pin data (Pins 0 - 31) */
    uint32 deviceIOData_63_32;      /* Output pin data (Pins 32 - 63) */
} VpDeviceIoAccessDataType;
```

The `accessType` parameter determines whether a read or write operation is performed on the I/O pins, and it can take one of the following values:

```
Enumeration Data Type: VpDeviceIoAccessType:
    VP_DEVICE_IO_WRITE              /* Perform device I/O write access */
    VP_DEVICE_IO_READ               /* Perform device I/O read access */
```

The `accessMask_63_32` and `accessMask_31_0` variables are combined to make a single 64-bit `accessMask` field, where each bit determines whether an individual pin is accessed (1) or ignored (0). During an I/O write operation, the state of any pin with its `accessMask` bit set to 0 remains unchanged. For a configuration that supports only 1 user I/O pin per channel, `accessMask[N]` sets the direction for the I/O pin belonging to channel `N`. For a configuration that supports 2 user I/O pins per channel, `accessMask[2N]` sets the direction for I/O Pin 0 belonging to channel `N`, and `accessMask[2N+1]` sets the direction for I/O Pin 1 belonging to channel `N`. Unused `accessMask` bits that do not map to a channel/pin are ignored.

The `deviceIOData_63_32` and `deviceIOData_31_0` variables are combined to make a single 64-bit `deviceIOData` field, where each bit determines the state of an individual output pin when a write operation is performed. The `deviceIOData` field is mapped to I/O pins in the same manner as the `accessMask` field described above. Note that `deviceIOData` is ignored if `accessType` is `VP_DEVICE_IO_READ`.

Example 1: 2-channel device each with 2 I/O pins, setting channel 1 I/O1 =1, channel 1 I/O2 = 0, channel 2 I/O1 = ignored, channel 2 I/O2 = 1

```
accessMask_31_0 = 0xB;  1 0 1 1
                        │ │ │ └─channel 1, I/O1 Access
                        │ │ └───channel 1, I/O2 Access
                        │ └─────channel 2, I/O1 Ignored
                        └───────channel 2, I/O2 Access

deviceIOData_31_0 = 9 (or 0xD);  1 x 0 1
                                 │ │ │ └─channel 1, I/O1 = 1
                                 │ │ └───channel 1, I/O2 = 0
                                 │ └─────channel 2, I/O1 Ignored
                                 └───────channel 2, I/O2 = 1
```

Example 2: 2-channel device each with 1 I/O pins, setting channel 1 I/O1 =1, channel 2 I/O1 = ignored

```
accessMask_31_0 = 1;  0 1
                      │ └─channel 1, I/O1 Access
                      └───channel 2, I/O1 Ignored

deviceIOData_31_0 = 1 (or 3);  x 1
                               │ └─channel 1, I/O1 = 1
                               └───channel 2, I/O1 Ignored
```

This function generates the `VP_DEV_EVID_IO_ACCESS_CMP` event indicating that the requested I/O access is done. The results of an I/O read operation are returned when this event occurs. Refer to Section 8.5.8 for details.

RETURNS

See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED	*VP_DEV_EVID_IO_ACCESS_CMP*, on page 89
DEVICES	All
TERMINATIONS	All

CHAPTER

10 STATUS AND QUERY FUNCTIONS

10.1 OVERVIEW

This chapter describes VP-API-II functions that get information and events from the VTD, including the following:

- `VpGetLoopCond()` – Reads loop conditions for an FXS line and returns parameters such as voltage, current, and resistance.
- `VpGetLineStatus()` – Returns the state of a particular status flag for one line.
- `VpGetDeviceStatus()` – Returns the state of a particular status flag for the first 32 lines of the device.
- `VpGetLineInfo()` – Retrieves line-specific information from a device or line context. This function is optional.
- `VpGetDeviceInfo()` – Retrieves device-specific information from a device or line context. This function is optional.
- `VpGetOption()` – Returns the current setting of an option.`VpGetLineState()` – Reads the current line state.
- `VpGetEvent()` (see **Section 7.3.3** for details) – Returns events corresponding to a device.
- `VpGetResults()` (see **Section 7.3.5** for details) – Reads the data associated with an event.

10.2 FUNCTION DESCRIPTIONS

10.2.1 VpGetLoopCond()

SYNTAX

```
VpStatusType
VpGetLoopCond(
 VpLineCtxType *pLineCtx,          /* Pointer to line context */
 uint16 handle)                    /* Handle value returned with event */
```

DESCRIPTION

This function starts a process that eventually returns the current loop and battery conditions for the specified line. Several measurements are taken from the device when this function is called. However, these measurements may not be taken simultaneously. The `VP_LINE_EVID_RD_LOOP` event occurs when the results are available for the application to read. The application can execute this function while the target line is in any state. However, some measurement results may not be valid in certain line states. See *VP_LINE_EVID_RD_LOOP*, on page 87 for details.

Notes:

1. *This function can consume a significant amount of MPI bandwidth. Frequently calling this function could degrade system performance. Use caution if polling this function.*
2. *Even though this function is not supported in the VP-API-II for VE880 and VE890, similar functionality is provided by Microsemi Professional Line Test Software for VE880 and VE890 (OPN LE880SLVVP for VE880, and LE890SLVVP for VE890).*

RETURNS

See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED

VP_LINE_EVID_RD_LOOP, on page 87

DEVICES

CSLAC-790

TERMINATIONS

FXS

10.2.2 VpGetLineStatus()

SYNTAX

```
VpStatusType
VpGetLineStatus(
 VpLineCtxType *pLineCtx,     /* IN: Pointer to line context */
 VpInputType input            /* IN: Status type being requested */
 bool *pStatus)               /* OUT: Pointer to status results */
```

DESCRIPTION

This function obtains the status of the specified `input` for the line associated with `pLineCtx`. The status result is written to the location pointed to by the argument `pStatus`. The following line inputs can be checked with this function:

```
Enumeration Data Type: VpInputType:
 FXS Status types            Description
 VP_INPUT_HOOK               /* Hook Status (ignoring pulse & flash) */
 VP_INPUT_RAW_HOOK           /* Hook Status (include pulse & flash) */
 VP_INPUT_GKEY               /* Ring-to-Ground current status */
 VP_INPUT_THERM_FLT          /* Thermal Fault Status */
 VP_INPUT_CLK_FLT            /* Clock Fault Status */
 VP_INPUT_AC_FLT             /* AC Fault Status */
 VP_INPUT_BAT1_FLT           /* Battery 1 Fault Status */
 VP_INPUT_BAT2_FLT           /* Battery 2 Fault Status */
 VP_INPUT_BAT3_FLT           /* Battery 3 Fault Status */
 VP_INPUT_RES_LEAK           /* Resistive Leak */

 FXO Status types            Description
 VP_INPUT_RINGING            /* Ringing Status */
 VP_INPUT_LIU                /* Line In Use Status */
 VP_INPUT_FEED_DIS           /* Feed Disable Status */
 VP_INPUT_FEED_EN            /* Feed Enable Status */
 VP_INPUT_DISCONNECT         /* Feed Disconnect Status */
 VP_INPUT_CONNECT            /* Feed Connect Status */
 VP_INPUT_POLREV             /* Polarity Reversal Status */

 FXS and FXO Status types    Description
 VP_INPUT_DC_FLT

/*
 * For FXS lines VP_INPUT_DC_FLT is the status of current from Ring-to-Battery.
 * See "VP_INPUT_GKEY" input for status of current from Ring-to-Ground.
 * For FXO lines VP_INPUT_DC_FLT is the status of voltage on the line when the FXO
 * line is providing loop open (states VP_LINE_FXO_OHT or VP_LINE_FXO_LOOP_OPEN)
 */
```

Considerations for reported line status:

`VP_INPUT_HOOK, VP_INPUT_RAW_HOOK:`

- Indicate `TRUE` for loop close detection and `FALSE` for loop open detection.
- When VP-API-II pulse detection is disabled (i.e., `VP_OPTION_ID_PULSE_MODE` set to `VP_OPTION_PULSE_DECODE_OFF`) `VP_INPUT_HOOK` and `VP_INPUT_RAW_HOOK` are identical.
- When VP-API-II pulse detection is enabled (i.e., `VP_OPTION_ID_PULSE_MODE` set to `VP_OPTION_PULSE_DECODE_ON`) `VP_INPUT_RAW_HOOK` reflects the instantaneous state of the line's hook detector and will toggle at the dial pulse/flash-hook rate. `VP_INPUT_HOOK` on the other hand reflects the overall state of hook detector and will remain steady during dial pulse/flash-hook activity. `VP_INPUT_HOOK` will change from `FALSE` (on-hook) to `TRUE` (off-hook) at the start of the initial off-hook detection and will continue to report `TRUE` until the line has detected on-hook for the time specified by `flashHookMax` if `EXTENDED_FLASH_HOOK` is set to `#undef`, or `onHookMin` if `EXTENDED_FLASH_HOOK` is set to `#define`.

VP_INPUT_LIU:

- While providing a loop close the FXO line status VP_INPUT_LIU changes with VP_LINE_EVID_POH (VP_INPUT_LIU = TRUE) and VP_LINE_EVID_PNOH (VP_INPUT_LIU = FALSE). Therefore VP_INPUT_LIU has no meaning while the FXO line is providing a loop close for devices that do not support these events. See Section 8.7.10 and Section 8.8.1 for supported devices.

This function returns the VP_STATUS_INVALID_ARG if the application requests status information for the wrong type of line termination (e.g. requesting FXS status for an FXO termination).

Notes:

1. *Applications that do not support Ring-to-Ground currents as a normal part of the call processing (e.g., Applications that do not support Ground Start Signaling) may want to monitor VP_INPUT_GKEY as a line fault condition.*
2. *See Table 8–3 to convert generic battery names to device-specific battery names.*

RETURNS See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED None

DEVICES All

TERMINATIONS All

10.2.3 VpGetDeviceStatus()

SYNTAX

```
VpStatusType
VpGetDeviceStatus(
  VpDevCtxType *pDevCtx,          /* Pointer to device context */
  VpInputType input,              /* Test the status of this input type */
  uint32 *pDeviceStatus)          /* Pointer to status results */
```

DESCRIPTION This function returns the status of the requested input for all lines the device.

Each bit in the result represents the status of input for one line. The status result is written to the location pointed to by pDeviceStatus. The least significant bit represents line 1. Each successive bit represents the next line, up to the most significant bit which represents line 32. If a device only supports N lines, then only the least-significant N bits of the result are meaningful. Refer to *VpGetLineStatus()*, on page 116 for the type definition of the input argument and for information on decoding the status flags.

RETURNS See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED None

DEVICES All

TERMINATIONS All

10.2.4 VpGetLineInfo()

SYNTAX

```
VpStatusType
VpGetLineInfo(
  VpLineInfoType *pLineInfo)        /* Pointer to line info */
```

DESCRIPTION

This function returns information about a line. The following structure is defined for use with this function:

```
typedef struct  {
    VpDevCtxType *pDevCtx;      /* Pointer to device Context */
    uint8 channelId;            /* Channel identity */
    VpLineCtxType *pLineCtx;    /* Pointer to Line Context */
    VpTermType termType;        /* Termination Type */
    VpLineIdType lineId;        /* Application system wide line identifier */
} VpLineInfoType;
```

This function can be used in the following two ways:

1. If the pointer to the device context (`pLineInfo->pDevCtx`) is not `VP_NULL`, then this function returns information for the line associated with the specified device context and `channelId`. It fills all other elements of the **`VpLineInfoType`** struct (`pLineCtx, lineId` and `termType`) with the requested data. If no line context is associated with the specified `channelId`, then this function writes `VP_NULL` to the line context pointer.
2. If `pDevCtx` is `VP_NULL`, and `pLineInfo->pLineCtx` (the pointer to the line context) is not `VP_NULL`, then this function returns information for the line associated with the specified line context. It fills all other elements of the **`VpLineInfoType`** struct (`pDevCtx`, `channelId`, and `termType`) with the requested data.

If `pLineInfo` is `VP_NULL` then this function returns an error. If both the pointer to the line context and the pointer to the device context are `VP_NULL` then this function also returns an error.

RETURNS

See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

All

10.2.5 VpGetDeviceInfo()

SYNTAX

```
VpStatusType
VpGetDeviceInfo(
  VpDeviceInfoType
*pDeviceInfo)                          /* Pointer to device info */
```

DESCRIPTION

This function returns information about a device. The following structure is defined for use with this function:

```
typedef struct  {
    VpLineCtxType *pLineCtx;                    /* Pointer to Line Context */
    VpDeviceIdType deviceId;                    /* Device identity */
    VpDevCtxType *pDevCtx;                      /* Pointer to device Context */
    VpDeviceType deviceType;                    /* Device Type */
    VpFeatureListType featureList;              /* Silicon features available */
    uint8 numLines;                             /* Number of lines */
    uint8 revCode;                              /* Silicon revision of the device */
    uint16 productCode;                         /* Silicon Product Code Number */
} VpDeviceInfoType;

typedef struct  {
    VpFeatureType testLoadSwitch;               /* External Test Load */
    VpFeatureType internalTestTermination;      /* Internal Test Load */
} VpFeatureListType;

Enumeration Data Type: VpFeatureType:
  VP_AVAILABLE
  VP_NOT_AVAILABLE
```

Devices have either the (external) `testLoadSwitch`, the (internal) `internalTestTermination` or neither. There are no devices that have both. Devices that have either the `testLoadSwitch` or the `internalTestTermination` support relay state `VP_RELAY_BRIDGED_TEST` (argument passed to **`VpSetRelayState()`**). See Appendix C for details.

This function can be used in the following two ways:

1. If the pointer to the line context (`pDeviceInfo->pLineCtx`) is not `VP_NULL`, then this function returns information about the device associated with the given line context. It fills all other elements in the `VpDeviceInfoType` struct. The identity of the device is stored in the `deviceId` field, a pointer to device context is stored in the `pDevCtx` field, the type of device is stored in the `deviceType` field, and the number of lines supported by the device is stored in the `numLines` field.
2. If the pointer to the line context is `VP_NULL` and the pointer to the device context (`pDeviceInfo->pDevCtx`) is not `VP_NULL`, then this function returns other details like device identity, device type, and the number of lines supported by the device. It stores this information in their respective fields. No information is written to the line context. Note that the application can use this function in this mode to learn the number of lines supported by the device before creating line objects.

This function returns an error if either `pDeviceInfo` is `VP_NULL` or if both the line context pointer and the device context pointer are `VP_NULL`.

RETURNS

See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

All

10.2.6 VpGetOption()

SYNTAX

```
VpStatusType
VpGetOption(
 VpLineCtxType *pLineCtx,          /* Pointer to line context */
 VpDevCtxType *pDevCtx,            /* Pointer to the device context */
 VpOptionIdType option,            /* Selects the option to get */
 uint16 handle)                    /* Handle value returned with event */
```

DESCRIPTION

This function retrieves the current setting of an option applied to the specified device or line. The `option` argument determines which option is read. For a list and description of all VP-API-II options see Chapter 3.

`VpGetOption()` actually starts a process in the VP-API-II/VTD that retrieves the requested data from a device. This function does not wait for the data to become available. Instead, this function returns immediately, and an event occurs at some later time indicating that the requested data is available.

This exact option setting that is retrieved by this function depends on the values of the device context, line context, and option arguments. The table below summarizes this behavior. The "Option" column indicates whether the target option is device-specific or line-specific. The "Device Ctx" and "Line Ctx" columns indicate whether a valid pointer or `VP_NULL` is passed for the `pDevCtx` and `pLineCtx` parameters, respectively.

Table 10–1 `VpGetOption()` Behavior

Option	Device Ctx	Line Ctx	Result
device	VP_NULL	VP_NULL	returns VP_STATUS_INVALID_ARG
device	VP_NULL	valid	gets option for device that controls the specified line
device	valid	VP_NULL	gets option for the specified device
device	valid	valid	returns VP_STATUS_INVALID_ARG
line	VP_NULL	VP_NULL	returns VP_STATUS_INVALID_ARG
line	VP_NULL	valid	gets option for the specified line
line	valid	VP_NULL	returns VP_STATUS_INVALID_ARG
line	valid	valid	returns VP_STATUS_INVALID_ARG

The `VP_LINE_EVID_RD_OPTION` event is generated as a result of this function call, indicating that the requested option data is available. The `handle` argument to **`VpGetOption()`** specifies the event handle that is attached to this event. Upon receiving this event, the application must call **`VpGetResults()`** with a pointer to the appropriate data structure to retrieve the option settings. The Refer to *VP_LINE_EVID_RD_OPTION*, on page 86 for more information on retrieving the option data associated with this event.

RETURNS

See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED

VP_LINE_EVID_RD_LOOP, on page 87

DEVICES

All

TERMINATIONS

All

10.2.7 VpGetLineState()

SYNTAX

```
VpStatusType
VpGetLineState(
 VpLineCtxType *pLineCtx,                /* Pointer to line context */
 VpLineStateType *pCurrentState)         /* Ptr to store line state */
```

DESCRIPTION

This function retrieves the current state of the specified line. The line state is written to the location pointed to by `pCurrentState`. All of the valid line states are described in context of the function *VpSetLineState()*, on page 101.

Notes:
When the line state is in either `VP_LINE_RINGING` or `VP_LINE_RINGING_POLREV`, the VP-API-II returns the user programmed state (i.e., `VP_LINE_RINGING/RINGING_POLREV`) during ringing cadence. That is, regardless of ringing cadence operations or Type I CID, the line state is considered to be in the VP-API-II state `VP_LINE_RINGING/RINGING_POLREV`.

RETURNS

See *VP-API-II Function Return Type*, on page 11

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

All

CHAPTER

11 SYSTEM SERVICES

11.1 OVERVIEW

The System Services layer provides critical section, timing and interrupt control functions. These functions are system-dependent and must be implemented specifically for each platform on which the VP-API-II is used.

Note that some of the functions described in this section are in support of interrupt management as described in Chapter 7, Interrupt and Event Handling. They are described here because these functions are to be implemented (when needed) by the user.

The following functions are included in the System Services layer.

- **VpSysWait()** – Implements a software delay
- **VpSysEnterCritical() and VpSysExitCritical()** – Blocks/Opens entry into a critical section of VP-API code through some user-defined method.
- **VpSysDisableInt() and VpSysEnableInt()** – Required by the Level-Triggered interrupt mode to disable/re-enable the CSLAC device interrupt in the host microprocessor.
- **VpSysTestInt()** – Required for all interrupt modes except for Simple Polled mode to test the CSLAC device interrupt line for active interrupts.
- **VpSysDtmfDetEnable() and VpSysDtmfDetDisable()** – These functions are used by the VP-API to control a DTMF digit decoding resource that is outside the scope of the VP-API. This is used for Type-II Caller ID implementation.
- **VpSysDebugPrintf()** – Print mechanism used by VP-API-II Debug features.

11.2 FUNCTION DESCRIPTIONS

11.2.1 VpSysWait()

SYNTAX

```
void
VpSysWait(
 uint8 time)                    /* Number of frame syncs to wait */
```

DESCRIPTION

The VP-API-II calls this function when it needs to wait a certain amount of time. The `time` argument determines the wait time in terms of 125us frame sync periods.

RETURNS

None

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

All

11.2.2 VpSysEnterCritical() and VpSysExitCritical()

SYNTAX

```
uint8
VpSysEnterCritical(or
VpSysExitCritical(
 VpDeviceIdType deviceId,                /* Selects the target device */
 VpCriticalSecType criticalSecType)      /* Indicates critical section type
                                         */
```

DESCRIPTION

These functions are only required in multi-threaded applications (including the process that calls `VpApiTick()`).

These functions protect critical sections of VP-API-II code from reentrant execution. The application developer must implement these functions such that for a given device, no VP-API-II functions can be called from another thread of execution while any thread is within a critical section. This is typically done by disabling appropriate interrupts or "taking" a task-blocking semaphore.

The `deviceId` argument indicates which device resource or object is being accessed during this critical section. In systems with more than one VTD attached to the host microprocessor, the application can use this information to limit the number of interrupts disabled or tasks blocked by semaphores to only those interrupts or tasks that are related to a specific device. In most implementations the `deviceId` argument can simply be ignored.

The `criticalSecType` argument specifies the type of critical section being entered, and may take one of the following values:

```
Enumeration Data Type: VpCriticalSecType:
    VP_MPI_CRITICAL_SEC
    VP_CODE_CRITICAL_SEC
```

MPI critical sections occur around MPI bus transactions. MPI transactions are timing-sensitive and must not be interrupted. HBI critical sections do not apply to CSLAC devices. Code critical sections are used to protect global data (device and line objects) from simultaneous access by more than one thread of execution.

In the simplest case, the same protection mechanism can be used for all critical section types. Alternatively, the application may choose to protect different types of critical sections using different mechanisms. For example, MPI critical sections could be protected by disabling all relevant interrupts, while code critical sections could be protected by semaphores. The decision is left to the application developer.

`VpSysEnterCritical() and VpSysExitCritical()` should be implemented such that critical sections could be nested within the VP-API-II. For example, a HBI/MPI critical section, or even a code critical section, could occur within a code critical section. Note that no other critical sections will ever occur within a HBI/MPI critical section. **`VpSysEnterCritical()`** should return the current critical section nesting level.

Notes:

If the application is designed such that all VP-API-II calls are made from only one thread of execution then these functions can be empty.

RETURNS The nesting level after the call is completed.

EVENTS GENERATED None

DEVICES All

TERMINATIONS All

11.2.3 VpSysDisableInt() and VpSysEnableInt()

SYNTAX

```
void
VpSysDisableInt(or
VpSysEnableInt(
  VpDeviceIdType deviceId)          /* Selects the target device */
```

DESCRIPTION

This System Services function must be implemented by the customer if the design is based on the CSLAC family of devices and uses the interrupt architecture as defined in *Mode 2: Interrupt Driven, Level-Triggered*, on page 70.

The function **VpSysDisableInt()** is called by the VP-API-II during the CSLAC Interrupt Service Routine (ISR) to disable/ignore the pending interrupt from the CSLAC without actually servicing the interrupt in the CSLAC. The CSLAC device for which interrupt needs to be disabled is indicated by `deviceId`.

The function **VpSysEnableInt()** is called by the VP-API-II just before returning from **VpApiTick()**.

These functions can both be empty in systems that use edge-triggered interrupts or poll for CSLAC interrupts.

RETURNS None

EVENTS GENERATED None

DEVICES CSLAC

TERMINATIONS All

11.2.4 VpSysTestInt()

SYNTAX

```
bool
VpSysTestInt(
  VpDeviceIdType deviceId)          /* Selects the target device. */
```

DESCRIPTION

This is a system services function and needs to be implemented by the customer only if VP-API-II is being used for CSLAC family of devices with an implementation architecture described in *Mode 3: Efficient Polled Mode*, on page 71. This function tests the logic level of the CSLAC interrupt present on the PIO pin connected to a non-interrupting input.

RETURNS Returns `TRUE` if CSLAC Interrupt pin indicates an active interrupt is pending, `FALSE` otherwise.

EVENTS GENERATED None

DEVICES CSLAC

TERMINATIONS All

11.2.5 VpSysDtmfDetEnable() and VpSysDtmfDetDisable()

SYNTAX

```
void
VpSysDtmfDetEnable(or
VpSysDtmfDetDisable(
 VpDeviceIdType deviceId,          /* Selects the target device */
 uint8 channelId)                  /* Selects the target channel */
```

DESCRIPTION

These functions are used in Type-II Caller ID when the Caller ID sequence reaches a "Detect Interval" command. This command is used to detect an acknowledgement from the CPE in the form of a DTMF digit.

For CSLAC devices, the VP-API/VTD does not implement DTMF detection; this is provided as a system service controlled by the functions **VpSysDtmfDetEnable()** and **VpSysDtmfDetDisable()**. When the Caller ID sequence reaches the "Detect Interval" command, **VpSysDtmfDetEnable()** is called. When the "Detect Interval" command completes, **VpSysDtmfDetDisable()** is called.

VpSysDtmfDetEnable() should do whatever initialization is required to enable DTMF detection on the specified device and channel. **VpSysDtmfDetDisable()** may be used to free any resources allocated by **VpSysDtmfDetEnable()**. These functions are platform-specific and must be implemented only if the functionality is needed. Otherwise, the implementation of these functions should be empty.

During DTMF detection, when a digit is received, the VoicePath API should be notified with a call to **VpDtmfDigitDetected()**. See *VpDtmfDigitDetected(), on page 111* for further information on this function.

RETURNS None

EVENTS GENERATED None

DEVICES CSLAC

TERMINATIONS FXS

11.2.6 VpSysDebugPrintf()

SYNTAX

```
void
VpSysDebugPrintf(
 Variable)
```

DESCRIPTION

This function is used by the Debug Functions described in Chapter 13, on page 131. It is a print mechanism used for the purposes of debug output and can be defined simply as "printf" or "printk".

RETURNS None

EVENTS GENERATED None

DEVICES All

TERMINATIONS All

CHAPTER

12 HARDWARE ABSTRACTION LAYER

12.1 OVERVIEW

The Hardware Abstraction Layer (HAL) defines functions for communicating with a target VTD. These functions hide the details of the platform hardware design from the VP-API-II. The customer must implement these functions as appropriate for their specific platform. Microsemi provides example implementations of these functions. There is only one HAL function required for CSLAC Applications:

- **`VpMpiCmd()`** – Implements an MPI/ZSI transaction.

This HAL function takes a `deviceId` argument that identifies the target VTD. The `deviceId` argument is of type **`VpDeviceIdType`** defined by the customer which can be any valid 'C' type (see "`vp_api_types.h`"). The `deviceId` is associated with the device object during configuration (`VpMakeDeviceObject()`) and passed through the VP-API-II to the application in some of the VP-API-II functions.

12.2 FUNCTION DESCRIPTIONS

12.2.1 VpMpiCmd()

SYNTAX

```
void
VpMpiCmd(
 VpDeviceIdType deviceId,        /* Selects the target CSLAC device */
 uint8 ecVal,                    /* CSLAC Enable Channel reg. value */
 uint8 cmd,                      /* CSLAC command byte */
 uint8 cmdLen)                   /* length of data string */
 uint8 *dataPtr)                 /* Pointer to data string */
```

DESCRIPTION

This function implements block level MPI read/write transactions with the CSLAC device. The target device is indicated by the `deviceId` argument. The `ecVal` argument contains the value written to the CSLAC device's Enable Channel register (register `0x4A`).

The `cmd` argument contains the first MPI command in the block (note: the total data passed may be a series of MPI cmd+data, not just a single MPI cmd+data set). Bit zero of the `cmd` byte specifies whether the entire MPI transaction is a read (1) or a write (0) operation. If the MPI transaction is a write operation, this function writes `cmdLen` number of bytes following the command byte from the location pointed by the `dataPtr` argument to the CSLAC device. If a read operation, this function reads and stores `cmdLen` number of bytes from the CSLAC device to the location pointed by the `dataPtr` argument.

REQUIREMENTS

Value	Interface	Requirement	Comments
CS Off-Time	MPI	>= 2500ns	
	ZSI	Function of SPI CLK and PCLK	See Data Sheet for details.
NO_OP	MPI	Not required in MPI transaction	
	ZSI	1 - NO_OP (0x06) Required after the "Read" Command for every Read transaction.	See the code examples below.
Blocking	MPI and ZSI	Required ONLY IF multiple access to the same device is possible by more than one thread at the same time.	
		IF Required: All MPI/ZSI commands to the device passed to VpMpiCmd() must be sent before ANY other access ot the device can be allowed.	The Application should use functions VpSysEnterCritical() and VpSysExitCritical() or similar methods to meet this requirement.
Latency	MPI and ZSI	Once started, a single MPI/ZSI transaction (full series of [MPI/ZSI Command + Data] in a single dataPtr buffer) should be completed in the shortest amount of time possible. Unexpected behavior may occur if commands are delayed.	

MPI EXAMPLE

```
#define EC_WRT_CMD (0x4A)

void
VpMpiCmd(
    VpDeviceIdType deviceId,
    uint8 ecVal, uint8 cmd, uint8 cmdLen, uint8 dataPtr)
{
    uint8 cmdIndex = 0;

    /* Write the EC Value to the EC Register */
    spi_byte_write(deviceId, EC_WRT_CMD);
    spi_byte_write(deviceId, ecVal);

    /* Write the Command Byte */
    spi_byte_write(deviceId, cmd);

    if (cmd & 1) {  /* If cmd LSB = '1', Perform Read Operation */
        for (cmdIndex = 0; cmdIndex < cmdLen; cmdIndex++) {
            dataPtr[cmdIndex] = spi_byte_read(deviceId);
        }
    } else {        /* If cmd LSB = '0', Perform Write Operation */
        for (cmdIndex = 0; cmdIndex < cmdLen; cmdIndex++) {
            spi_byte_write(deviceId, dataPtr[cmdIndex]);
        }
    }
}
```

ZSI EXAMPLE

This is easily modified for ZSI devices as follows:

```
    if (cmd & 1) {  /* If cmd LSB = '1', Perform Read Operation */
        /* Previous Command was Read. Need to send the NO_OP first
         * for ZSI interface */
        spi_byte_write(deviceId, 0x06);

        /* Now read in the data */
        for (cmdIndex = 0; cmdIndex < cmdLen; cmdIndex++) {
            dataPtr[cmdIndex] = spi_byte_read(deviceId);
        }
    } else {        /* If cmd LSB = '0', Perform Write Operation */
        for (cmdIndex = 0; cmdIndex < cmdLen; cmdIndex++) {
            spi_byte_write(deviceId, dataPtr[cmdIndex]);
        }
    }
}
```

APPLICATION NOTE

In the case where the same **`VpMpiCmd()`** supports both MPI devices and ZSI devices, the `deviceId` may easily be used to determine which type of interface exists. Remember that the `deviceId` is of user defined type `VpDeviceIdType`, so it is possible to do something simple to identify MPI vs. ZSI. For example: using the MSB to identify ZSI ('1' = ZSI, '0' = MPI) then the `deviceId` selection could be as follows:

Given: `typedef uint8 VpDeviceIdType;`

deviceId	Interface	CS
0x00	MPI	0
0x80	ZSI	0
0x01	MPI	1
0x81	ZSI	1

and so on...

The VP-API-II makes no assumption about the type of `VpDeviceIdType`. However, if making `VpDeviceIdType` non-compatible with an integer type (e.g., if defined as a struct), then the user may have to modify the macro:

```
#define VP_PRINT_DEVICE_ID(deviceId) VpSysDebugPrintf(" (dev 0x%2.2X)",
(int)deviceId)
```

Found in `vp_api_types.h`.

CHAPTER

13 DEBUG FUNCTIONS

13.1 OVERVIEW

The VoicePath API-II includes many different types of debug used to help isolate problems in the application and/or system. Each type of output can be included/excluded at compile time. Included types can be enabled/disabled at run time using `VP_OPTION_ID_DEBUG_SELECT` option.

13.2 TYPES OF DEBUG OUTPUT

Table 13–1 Debug Flags Used by CSLAC VP-API-II

Macro	Description	Purpose
`VP_DBG_ERROR`	Error messages	Internal or System error. May coincide with VP-API-II return value other than `VP_STATUS_SUCCESS`.
`VP_DBG_API_FUNC`	VP-API-II function entry/ exit messages	Output when VP-API-II functions are entered and exited.
`VP_DBG_API_FUNC_INT`	Internal function entry/exit messages	Output when internal functions are entered and exited.
`VP_DBG_HOOK`	Hook and Ground Key activity	Device detection and VP-API-II reporting information associated with hook and ground key state changes.
`VP_DBG_LINE_STATE`	Device and VP-API-II line states.	Output when either device or VP-API-II states are changed.
`VP_DBG_CALIBRATION`	Calibration related messages.	Calibration related information. Occurs when accessing calibration values and during execution of calibration functions.
`VP_DBG_TEST_FUNC`	VP-API-II Test (primitive level) entry/exit messages.	Output when Line Test primitive functions are entered and exited.
`VP_DBG_TEST`	Generic VP-API-II Test messages.	Test information. Occurs during execution of the Line Test primitives.
`VP_DBG_TEST_PCM`	VP-API-II Test messages associated with PCM Collection.	Output during execution of Line Test primitives. Generated only if using "EZ Mode" (See appropriate LT-API User's Guide for details).
`VP_DBG_FXO_FUNC`	Messages related to FXO device or line behavior.	Output during FXO specific functions.
`VP_DBG_SEQUENCER`	Messages from the internal VP-API-II sequencer.	Output when running a ringing/tone cadence or other operation that uses the VP-API-II sequence engine (e.g., **`VpSendSignal()`**, **`VpStartMeter()`**) excluding CID operations.
`VP_DBG_CID`	Caller ID related messages.	Output for all Caller ID related functions (Note: Does not ouput Ringing Control information used in Type I CID generation).
`VP_DBG_HAL`	Hardware Abstraction Layer messages	Device accesses performed by the VP-API-II
`VP_DBG_SSL`	System Services Layer messages	Displays function calls and result codes of customer-defined SSL functions.
`VP_DBG_EVENT`	Event messages	Event related information. Occurs in functions **`VpGetEvent()`** and **`VpGetResults()`**.
`VP_DBG_GAIN`	Digital Gain	Output for access to the Digital Gain blocks of the SLAC. Occurs in **`VpSetRelGain()`** and **`VpSetOption()`** for `optionId = VP_OPTION_ID_ABS_GAIN`

The above macros are defined as bitmasks which can be ORed together to select any combination of debug output types. The macro `VP_DBG_ALL` is provided for selecting all debug output types.

13.3 DEBUG OUTPUT SELECTION AT COMPILE TIME

Debug output strings and the code necessary for displaying them can occupy a significant amount of memory in some applications. Therefore, the VP-API-II provides the ability to exclude unwanted types of debug output at compile time.

Customers may want to compile in all types of debug output during initial development, but include less debug output in the final application. Or, if separate "production" and "debug" builds are maintained, a different selection of debug output may be specified in each.

The selection of debug output types at compile time is specified in the header files in the api_lib/includes directory:

The `VP_DEBUG` option (in vp_api_cfg.h) applies to all types of debug output.

- If `VP_DEBUG` is undefined, then all debug output is compiled out.
- If `VP_DEBUG` is defined, then `VP_CC_DEBUG_SELECT` option selects which types of debug output will be included.

The `VP_CC_DEBUG_SELECT` option (in vp_debug.h) can be defined as `VP_DBG_ALL` to include all types of debug output, or it can be defined as the bitwise OR of one or more of the debug selector values (Table 13–1). Any debug output types not included in the definition will be compiled out.

Debug output excluded at compile time cannot be enabled at runtime.

13.4 DEBUG OUTPUT SPECIFICITY

For each type of debug output, the VP-API-II source code may generate line-specific, device-specific, or "non-specific" (not particular to any device or line) messages.

Line-specific messages apply to a single line in the system. For example, passing an invalid Profile argument to **`VpConfigLine()`** will generate a line-specific debug message.

If the line cannot be identified, or if the error message is logically not specific to a particular line, then a device-specific debug message will be generated.

If the device cannot be identified, a non-specific debug message will be generated. In some instances, a message may be clearly line-specific or device-specific, but the particular function in which it is generated does not have enough information to determine the line or device association. In such cases, the debug output is non-specific.

The run-time selection of line-specific debug output messages is stored in the Line Object. The device-specific selections are stored in the Device Object. The non-specific selections are stored in a global variable (`vpDebugSelectMask`) defined in vp_debug.c.

13.5 DEBUG OUTPUT SELECTION AT RUN TIME

At run time, the `VP_OPTION_ID_DEBUG_SELECT` option can be used to enable/disable debug output. Using this option, each type of debug output can be enabled/disabled per line, per device, or for non-specific output.

The `pValue` argument of **`VpSetOption()`** should be a pointer to a `uint32` number. The value should be the bitwise OR of one or more debug output selectors (Table 13–1). Other debug output types will be disabled.

The other arguments to **`VpSetOption()`** determine the specificity of the debug output that is being enabled/disabled:

Table 13–2 Valid Parameter Combinations for VP_OPTION_ID_DEBUG_SELECT

pLineCtx	pDevCtx	Result
valid	VP_NULL	Enables/disables line-specific debug output messages for the given line
VP_NULL	valid	Enables/disables device-specific debug output messages for the given device
VP_NULL	VP_NULL	Enables/disables non-specific debug output messages

It is important to note that the above three cases do not overlap in effect. I.e., enabling/disabling the device-specific debug output for a given device will have no effect on the line-specific debug output for lines of that device. Also, enabling/disabling non-specific debug output has no effect on device-specific or line-specific output.

13.6 PORTABILITY CONCERNS

13.6.1 Static Variable Coherency

It has always been a major design goal of the VP-API-II to give customers complete freedom over memory allocation. To that end, all state variables are maintained in a Line Object or Device Object which can be allocated statically, dynamically ("malloc"), or on the stack.

To allow enabling/disabling debug output at run-time for "non-specific" events, the VP-API-II uses a statically-allocated global variable, of type `uint32`, called `vpDebugSelectMask`, located in vp_debug.c. This variable is modified by **`VpSetOption()`** using `optionId` = `VP_OPTION_ID_DEBUG_SELECT` when both the `pLineCtx` and `pDevCtx` arguments are `VP_NULL`. It is accessed every time a non-specific debug output message is generated to determine if the message will be displayed.

The `vpDebugSelectMask` variable is internal to the API, and it is not normally necessary to access it externally. However, problems can arise in multiprocessing environments where each process may have its own copy of all static variables. In such cases, it is important to restore this variable each time a new process is started, either by modifying it directly, or by calling **`VpSetOption()`**.

13.6.2 VpSysDebugPrintf()

The VP-API-II routes all debug output through a customer-defined function: **`VpSysDebugPrintf()`**. This function should be declared in sys_service.h (discussed in Chapter 11). In the simplest case, the following definition is sufficient:

```
#define VpSysDebugPrintf printf
```

Please note that this function may be called multiple times in displaying a single line of debug output. I.e., the argument will not always end in `"\n"`. This may be important for systems in which prefixing, suffixing, or other handling of the output stream is done line-by-line.

13.6.3 ANSI X3.64 Colors

The VP-API-II uses ANSI X3.64 color codes to display each type of debug message in a different color. These codes work with all popular modern terminal programs. However, there are many cases in which it is desirable to exclude these extra characters from the debug output. To do so, undefine the `VP_DEBUG_COLOR` option in vp_debug.h.

13.6.4 Displaying VpDeviceIdType and VpLineIdType

Line-specific and device-specific debug output messages are prefixed by displaying the corresponding `VpLineIdType` or `VpDeviceIdType` struct. Since these are customer-defined types, it may not be appropriate to display them using a `printf("%d")` format specifier. If not, the display method can be changed by modifying the definitions of the `VP_PRINT_DEVICE_ID` and `VP_PRINT_LINE_ID` macros (located in in vp_debug.h).

CHAPTER

A GLOSSARY

Channel	See Section 1.1.2
Codec	Coder/Decoder
CSLAC	Conventional Microsemi SLAC™ device. For a complete list of supported CSLAC products, please see *Supported Hardware Configurations*, on page 5.
Device	See Section 1.1.2
DTMF	Dual-Tone Multi Frequency
FXO	Foreign eXchange Office interface. This is the plug on the phone that receives a Plain Old Telephone Service (POTS) signal, typically from a Central Office (CO) of the Public Switched Telephone Network (PSTN). An FXO interface points to the Telco office.
FXS	Foreign eXchange Subscriber interface. This is the plug on the wall that delivers a POTS signal from the local phone company's CO and must be connected to subscriber equipment such as telephones, modems, or fax machines. An FXS interface points to the subscriber.
GPI	General Purpose Parallel Interface. This is the VCP generic parallel port interface for the host processor. It is one of two physical interfaces currently available for the Host Bus Interface (HBI).
GR-909	Telcordia specification for Fiber in the Local Loop. GR-909 specifications for metallic loop testing have become the testing guidelines for many short-loop applications.
ISR	Interrupt Service Routine
LCAS	Line Circuit Access Switch. LCAS devices are essentially solid-state relays designed for telephony applications.
Line	See Section 1.1.2
MPI	Micro-Processor Interface. The MPI is Microsemi's serial control interface for CSLAC devices.
Profile	Profiles encapsulate application specific data including cadencing, tones, Caller ID parameters, etc.
ProfileWizard	A Microsoft Windows application that creates and organizes VoicePath profiles, included in the VoicePath SDK.
PSTN	Public Switched Telephone Network
SLAC™	Subscribe Line Access Circuit, a Microsemi trademark.
SLIC	Subscriber Line Interface Circuit

SPI	Serial Peripheral Interface. This is a four wire serial control interface between the VCP and the host processor that electrically conforms to the Motorola SPI slave interface standard. It is one of two physical interfaces currently available for the HBI.
Subscriber Line	The analog telephone line connecting the subscriber to the PSTN. Subscriber line is synonymous with loop or local loop.
VoicePath™ API II (VP-API-II)	An Application Program Interface that provides access to Microsemi's VTDs via the HBI or MPI. It is the primary component of the VoicePath Software Development Kit (VP SDK).
VCP	Voice Control Processor. For a complete list of VCP products please see *Supported Hardware Configurations*, on page 5.
VTD	Voice Termination Device.

CHAPTER

B FUNCTION INDEX

Table B–1 lists all VP-API-II functions, along with their input types, return type, applicable devices, and applicable termination types.

Note:

1. *Termination type "All" means either all termination types supported by the applicable devices, or the termination type is not relevant to the function.*

The page number of the complete function description is included for each function in Table B–1. If the page number for any function is empty, this means that this document was created for a device that does not support that function.

Table B–1 VoicePath™ API II Functions Summary

Function Name	Arguments	Return Type	Devices	Terminations	Page
System Configuration Functions					
VpMakeDeviceObject	VpDeviceType deviceType, VpDeviceIdType deviceId, VpDevCtxType *pDevCtx, void *pDevObj	VpStatusType	All	All	51
VpMakeLineObject	VpTermType termType, uint8 channelId, VpLineCtxType *pLineCtx, void *pLineObj, VpDevCtxType *pDevCtx	VpStatusType	All	All	52
VpMakeDeviceCtx	VpDeviceType deviceType, VpDevCtxType *pDevCtx, void *pDevObj	VpStatusType	All	All	53
VpMakeLineCtx	VpLineCtxType *pLineCtx, void *pLineObj, VpDevCtxType *pDevCtx	VpStatusType	All	All	53
VpFreeLineCtx	VpLineCtxType *pLineCtx	VpStatusType	All	All	54
VpMapLineId	VpLineCtxType *pLineCtx, VpLineIdType lineId	VpStatusType	All	All	54
Initialization Functions					
VpInitCustomTermType	VpDevCtxType *pDevCtx, VpLineCtxType *pLineCtx, VpProfilePtrType pCustomTermProfile	VpStatusType	CSLAC-580	FXS	60
VpInitDevice	VpDevCtxType *pDevCtx, VpProfilePtrType pDevProfile, VpProfilePtrType pAcProfile, VpProfilePtrType pDcProfile, VpProfilePtrType pRingProfile, VpProfilePtrType pFxoAcProfile, VpProfilePtrType pFxoCfgProfile	VpStatusType	All	All	56
VpInitLine	VpLineCtxType *pLineCtx, VpProfilePtrType pAcProfile, VpProfilePtrType pDcFeedOrFxoCfgProfile, VpProfilePtrType pRingProfile	VpStatusType	All	All	58
VpConfigLine	VpLineCtxType *pLineCtx, VpProfilePtrType pAcProfile, VpProfilePtrType pDcFeedOrFxoCfgProfile, VpProfilePtrType pRingProfile	VpStatusType	All	All	59

Table B–1 VoicePath™ API II Functions Summary(Continued)

Function Name	Arguments	Return Type	Devices	Terminations	Page
VpCalLine	VpLineCtxType *pLineCtx	VpStatusType	CSLAC-880, CSLAC-890	FXS	63
VpCal	VpLineCtxType *pLineCtx, VpCalType calType, void *inputArgs	VpStatusType	CSLAC-880, CSLAC-890	All	61
VpInitRing	VpLineCtxType *pLineCtx, VpProfilePtrType pCadProfile, VpProfilePtrType pCidProfile	VpStatusType	All	FXS	64
VpInitCid	VpLineCtxType *pLineCtx, uint8 length, uint8p pCidData	VpStatusType	All except CSLAC-580	FXS	65
VpInitMeter	VpLineCtxType *pLineCtx, VpProfilePtrType pMeterProfile	VpStatusType	CSLAC-880	FXS	65
VpInitProfile	VpDevCtxType *pDevCtx, VpProfileType type, VpProfilePtrType pProfileIndex, VpProfilePtrType pProfile	VpStatusType	All	All	66
Interrupt Functions					
VpApiTick	VpDevCtxType *pDevCtx, bool *pEventStatus	VpStatusType	CSLAC	All	73
VpVirtualISR	VpDevCtxType *pDevCtx	VpStatusType	CSLAC	All	73
VpGetEvent	VpDevCtxType *pDevCtx, VpEventType *pEvent	bool	All	All	74
VpGetResults	VpEventType *pEvent, void *pResults	VpStatusType	All	All	76
VpFlushEvents	VpDevCtxType *pDevCtx	VpStatusType	All	All	75
Control Functions					
VpFreeRun	VpDevCtxType *pDevCtx, VpFreeRunModeType freeRunMode)	VpStatusType	CSLAC-880, CSLAC-890	All	100
VpSetLineState	VpLineCtxType *pLineCtx, VpLineStateType state	VpStatusType	All	All	101
VpSetLineTone	VpLineCtxType *pLineCtx, VpProfilePtrType pToneProfile, VpProfilePtrType pCadProfile, VpDtmfToneGenType *pDtmfControl	VpStatusType	All	All	105

Table B–1 VoicePath™ API II Functions Summary(Continued)

Function Name	Arguments	Return Type	Devices	Terminations	Page
VpSetRelayState	VpLineCtxType *pLineCtx, VpRelayControlType rState	VpStatusType	All	All	103
VpSetRelGain	VpLineCtxType *pLineCtx, uint16 txLevel, uint16 rxLevel, uint16 handle	VpStatusType	CSLAC-880, CSLAC-890, CSLAC-580	All	107
VpSendSignal	VpLineCtxType *pLineCtx, VpSendSignalType signalType, void *pSignalData	VpStatusType	All	All	108
VpSendCid	VpLineCtxType *pLineCtx, uint8 length, VpProfilePtrType pCidProfile, uint8p pCidData	VpStatusType	All except CSLAC-580	FXS	109
VpContinueCid	VpLineCtxType *pLineCtx, uint8 length, uint8p pCidData	VpStatusType	All except CSLAC-580	FXS	110
VpDtmfDigitDetected	VpLineCtxType *pLineCtx, VpDigitType digit, VpDigitSenseType sense	VpStatusType	CSLAC	FXS	111
VpStartMeter	VpLineCtxType *pLineCtx, uint16 onTime, uint16 offTime, uint16 numMeters	VpStatusType	All	FXS	108
VpSetOption	VpLineCtxType *pLineCtx, VpDevCtxType *pDevCtx, VpOptionIdType option, void *pValue	VpStatusType	All	All	104
VpDeviceIoAccess	VpDevCtxType *pDevCtx, VpDeviceIoAccessDataType *pDeviceIoData	VpStatusType	All	All	112
Status and Query Functions					
VpGetLineStatus	VpLineCtxType *pLineCtx, VpInputType input, bool *pStatus	VpStatusType	All	All	116
VpGetDeviceInfo	VpDeviceInfoType *pDeviceInfo	VpStatusType	All	All	119
VpGetLineInfo	VpLineInfoType *pLineInfo	VpStatusType	All	All	118
VpGetDeviceStatus	VpDevCtxType *pDevCtx, VpInputType input, uint32 *pDeviceStatus	VpStatusType	All	All	117
VpGetLoopCond	VpLineCtxType *pLineCtx, uint16 handle	VpStatusType	CSLAC-790	FXS	115

Table B–1 VoicePath™ API II Functions Summary(Continued)

Function Name	Arguments	Return Type	Devices	Terminations	Page
VpGetOption	VpLineCtxType *pLineCtx, VpDevCtxType *pDevCtx, VpOptionIdType option, uint16 handle	VpStatusType	All	All	120
VpGetLineState	VpLineCtxType *pLineCtx, VpLineStateType *pCurrentState	VpStatusType	All	All	121
System Services Layer Functions					
VpSysEnterCritical	VpDeviceIdType deviceId, VpCriticalSecType criticalSecTyp	uint8	All	All	124
VpSysExitCritical	VpDeviceIdType deviceId, VpCriticalSecType criticalSecTyp	uint8	All	All	124
VpSysWait	uint8 time	void	All	All	123
VpSysDisableInt	VpDeviceIdType deviceId	void	CSLAC	All	125
VpSysEnableInt	VpDeviceIdType deviceId	void	CSLAC	All	125
VpSysTestInt	VpDeviceIdType deviceId	bool	CSLAC	All	125
VpSysDtmfDetEnable	VpDeviceIdType deviceId	void	CSLAC	FXS	126
VpSysDtmfDetDisable	VpDeviceIdType deviceId	void	CSLAC	FXS	126
VpSysDebugPrintf	Variable	void	All	All	126
Hardware Abstraction Layer (HAL) Functions					
VpMpiCmd	VpDeviceIdType deviceId, uint8 ecVal, uint8 cmd, uint8 cmdLen, uint8 *dataPtr	void	CSLAC	All	128

CHAPTER

C RELAY CONFIGURATIONS

This appendix describes relay configurations for various line termination types. The relay states described in this section could be exercised using the function *VpSetRelayState()*, on page 103. Only those relay states that are described in this section are valid relay states for a given line termination types.

Relay state `VP_RELAY_BRIDGED_TEST` is not supported on all 880 silicon (see Table C–1 for details). In cases when it is supported, the implementation may be either by an external resistor (typically 1Kohm) or internal test termination providing ~0ohm load. To determine which load method is used the application may call **`VpGetDeviceInfo()`** and check the value of `featurelist.internalTestTermination` (`VP_AVAILABLE` means internal test termination method is used, `VP_NOT_AVAILABLE` means external load is used).

Note:

1. The 880 and 890 devices do not support VP_RELAY_TALK.

Table C–1 VE880 BRIDGED_TEST Support

Silicon Part Number	BRIDGED_TEST Support?
Le88111, Le88116, Le88211, Le88216, Le88221, Le88226, Le88311	No
Le88131, Le88136, Le88231, Le88236, Le88241, Le88246, Le88266, Le88276, Le88286, Le88331, Le88506, Le88264, Le88536	Yes

Figure C–1 VP_TERM_FXS_GENERIC and VP_TERM_FXS_LOW_PWR Terminations

Figure C–2 VP_TERM_FXS_ISOLATE, VP_TERM_FXS_ISOLATE_LP, VP_TERM_FXS_SPLITTER and VP_TERM_FXS_SPLITTER_LP Terminations

Figure C–3 VP_TERM_FXS_TITO_TL_R Termination

Figure C–4 VP_TERM_FXS_75181 Termination

Figure C–5 VP_TERM_FXS_75282 Termination

Figure C–6 VP_TERM_FXS_RR Termination

Figure C–7 VP_TERM_FXS_TO_TL Termination

CHAPTER

D REVISION HISTORY

REV 14 - 2/09/2011

- Added termination type VP_TERM_FXS_ISOLATE to list of termination types with default Relay State in Section 6.2.1.
- Clarified 2.14 format input to **VpSetRelGain()** function in Section 9.2.6, behavior of VP_OPTION_ID_EVENT_MASK in Section 3.3.14, and device I/O configuration and access in Section 3.3.15 and Section 9.2.12.
- Added support for Le88264 silicon (see Design Considerations and VE880 BRIDGED_TEST Support).
- Provided the supported gain range and resolution in VP_OPTION_ID_ABS_GAIN.

REV 15 - 7/15/2011

- Replaced incorrect instances of **VpSendSignal()** with **VpStartMeter()**.
- Corrected Table 1–3 for existing supported FXO Termination types.
- Improved usage notes and provided example for **VpMpiCmd()**.

REV 16 - 11/15/2011

- Added new line states VP_LINE_HOWLER and VP_LINE_HOWLER_POLREV to function **VpSetLineState()**.
- Added support for UK Howler Tones: BTNR Version 15 and BTNR Draft 960-G. See **VpSetLineTone()** for details.
- Added details for Special Howler Tones generated by the VP-API-II (UK, AUS, and NTT). See **VpSetLineTone()** for details.
- Added VE880/890 Wideband Application Notes and Timeslot details to **VP_OPTION_ID_CODEC**.

REV 17 - 11/13/2012

- Added new CODEC modes VP_OPTION_ALAW_WIDEBAND and VP_OPTION_MLAW_WIDEBAND
- Updated **VP_OPTION_ID_ABS_GAIN** description of "Quiet", "Restore", "No Change" and "Gain Unknown". 'C' examples provided.

REV 18 - 03/20/13

- Added the timeStamp output to many events where previously the event member parmHandle was not being used. See Chapter **Events** for details.
- Added documentation for event **VP_LINE_EVID_TEST_CMP**.

REV 19 - 08/23/13

- Added return value **VP_STATUS_ERR_SPI** to **VP-API-II Function Return Type** (Table 1.2.6) and details in **VpInitDevice()**.
- Added ZSI discussion and improved requirements explanation for **VpMpiCmd()**.
- Improved requirements explanation for **VpInitDevice()**, **VpInitLine()** and **VpConfigLine()**.

REV 20 - 06/05/2014

- Added `VP_OPTION_ID_DCFEED_PARAMS` and `VP_OPTION_ID_RINGING_PARAMS` for VE890.
- Added a short note to say that `VpCalLine()` should not be run with a phone off-hook.
- Changed `VP_DEV_EVID_BAT_FLT` to only supported for VE790.
- Updated `VP_OPTION_ID_DEBUG_SELECT` default value.

Power Matters.

http://www.microsemi.com/cmpg

Information relating to products and services furnished herein by Microsemi Corporation or its subsidiaries (collectively "Microsemi") is believed to be reliable.. However, Microsemi assumes no liability for errors that may appear in this publication, or for liability otherwise arising from the application or use of any such information, product or service or for any infringement of patents or other intellectual property rights owned by third parties which may result from such application or use. Neither the supply of such information or purchase of product or service conveys any license, either express or implied, under patents or other intellectual property rights owned by Microsemi or licensed from third parties by Microsemi, whatsoever. Purchasers of products are also hereby notified that the use of product in certain ways or in combination with Microsemi, or non-Microsemi furnished goods or services may infringe patents or other intellectual property rights owned by Microsemi.

This publication is issued to provide information only and (unless agreed by Microsemi in writing) may not be used, applied or reproduced for any purpose nor form part of any order or contract nor to be regarded as a representation relating to the products or services concerned. The products, their specifications, services and other information appearing in this publication are subject to change by Microsemi without notice. No warranty or guarantee express or implied is made regarding the capability, performance or suitability of any product or service. Information concerning possible methods of use is provided as a guide only and does not constitute any guarantee that such methods of use will be satisfactory in a specific piece of equipment. It is the user's responsibility to fully determine the performance and suitability of any equipment using such information and to ensure that any publication or data used is up to date and has not been superseded. Manufacturing does not necessarily include testing of all functions or parameters. These products are not suitable for use in any medical products whose failure to perform may result in significant injury or death to the user. All products and materials are sold and services provided subject to Microsemi's conditions of sale which are available on request.

Microsemi, the Microsemi logo, Zarlink, and combinations thereof, Power Matters, SLAC, VeriVoice, VoiceEdge, VoicePort, and VoicePath are trademarks or registered trademarks of Microsemi Corporation. All other trademarks mentioned in this document are the property of their respective owners.

© 2014 Microsemi Corporation. All Rights Reserved

TECHNICAL DOCUMENTATION - NOT FOR RESALE