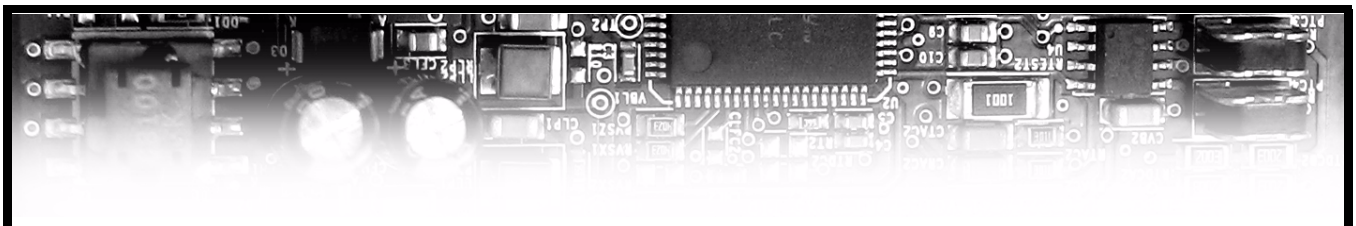


VoicePath™ API-II

CSLAC Reference Guide

Part Number:	**580, 790, 880, 890 Series**
Revision Number:	**G2**
Issue Date:	**January 2008**

ZARLINK SEMICONDUCTOR

FEATURING Legerity® VOICE SOLUTIONS

TABLE OF CONTENTS

CHAPTER

1 INTRODUCTION

1.1 ABOUT THIS USER'S GUIDE

This document describes the Zarlink Semiconductor VoicePath™ (VP) Application Program Interface (API) II that controls Zarlink Semiconductor's telephony Voice Termination Devices (VTDs). This is the second version of the VoicePath API, and throughout this document it is simply referred to as the *VoicePath API* or *VP-API*. This chapter highlights the document structure and conventions and summarizes the VP-API 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 profiles.

Chapter 3, System Configuration Functions: Describes the VP-API system configuration functions.

Chapter 4, Options: Describes the VP-API options.

Chapter 5, Events: Describes the VP-API events.

Chapter 6, Initialization Functions: Describes the VP-API initialization functions.

Chapter 7, Control Functions: Describes the VP-API control functions.

Chapter 8, Status and Query Functions: Describes the VP-API status and query functions.

Chapter 9, System Services: Describes the VP-API system support functions.

Chapter 10, Hardware Abstraction Layer: Describes the VP-API hardware abstraction layer functions.

Chapter 11, Interrupt Handling: Discusses methods for handling VP-API interrupts.

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

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

1.1.2 Frequently Used Terms

The following terms are used extensively throughout this document:

Device: The term *device* refers to a Zarlink Semiconductor VTD. Examples of a device include Zarlink Semiconductor's conventional SLAC™ devices, Voice Control Processor (VCP) devices, and Voice Packet Processor (VPP) devices. The VP-API 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 software, Zarlink Semiconductor 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 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.
Italic	Emphasizes an important term.

1.2 VOICEPATH API OVERVIEW

The VoicePath API 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 Zarlink Semiconductor Voice Termination Devices. The VP-API hides the details of controlling Zarlink Semiconductor VTDs and allows software developers to focus on the application instead of the hardware.

1.2.1 Features

Listed below are the key features of the VoicePath API. Note that some features depend on support from the underlying hardware.

- Provides an abstract, uniform software interface for any combination of Zarlink Semiconductor voice products.
- Utilizes the concept of Profiles to help organize design-specific parameters.
- Supports any combination of FXS and FXO lines configured for either loop-start signaling or ground-start signaling.
- Includes pulse-digit/flash decoder and ring/tone cadence engine.
- Proven on embedded operating systems such as Linux and VxWorks, and also compatible

with non-OS environments. Fits into common driver and static/dynamic library models.

- Supports various interrupt modes and both big-endian and little-endian microprocessors.
- Implemented in object oriented C code that is efficient, portable, and ANSI C compliant.
- Supports line testing equivalent to GR-844 and GR-909 functions.

1.2.1.1 Profiles

Zarlink Semiconductor products can be configured to meet worldwide standards, including custom requirements. To address such varying system-level specifications, Zarlink Semiconductor provides tools like WinSLAC™ and ProfileWizard to help engineers generate design data. The design data provided by these tools is organized into profiles to meet specific system requirements. The data for each profile is created with the Profile Wizard application. The VP-API defines profiles for the following design parameters:

- Device Configuration
- AC transmission
- DC feed
- FXO Line Detection and Generation Parameters
- Ringing configuration
- Call-progress tones
- Cadence patterns
- Caller ID configuration
- Custom Termination I/O Configuration
- Metering configuration

1.2.1.2 Options

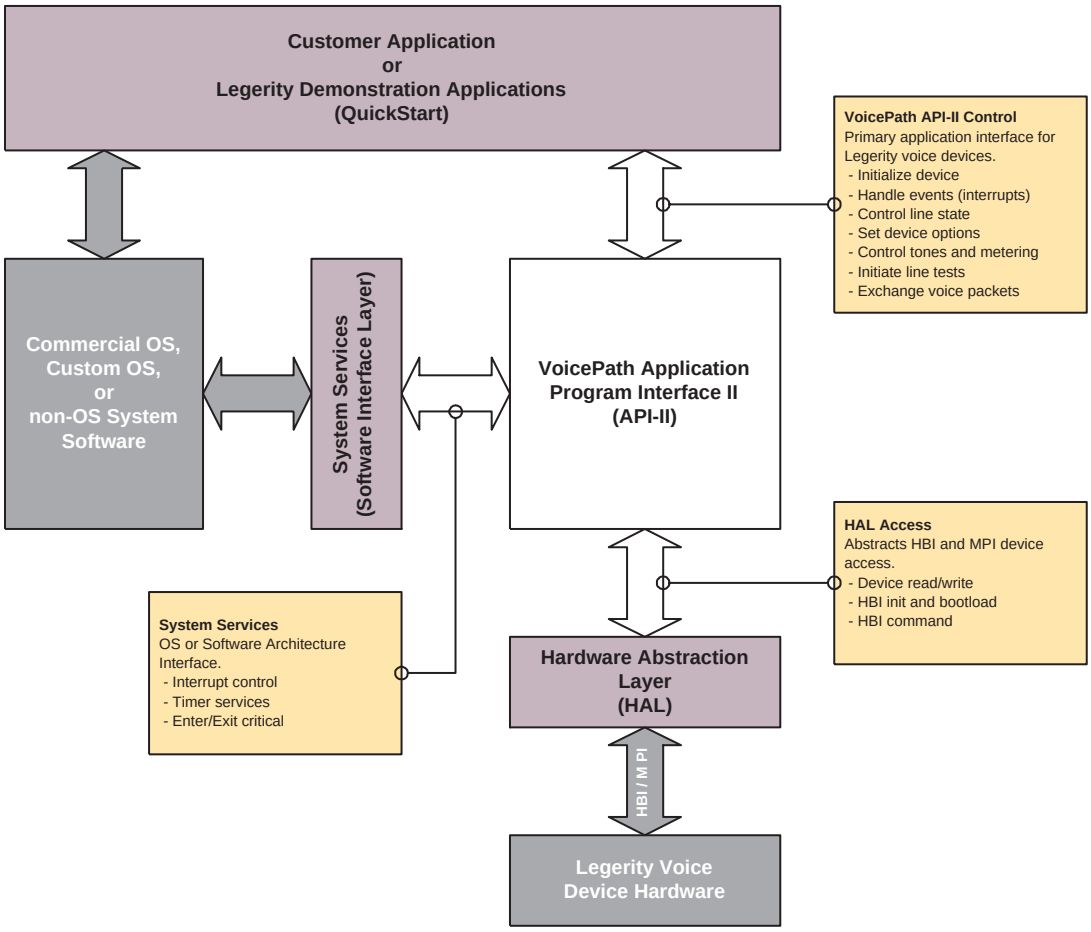
The VP-API provides several options to configure Zarlink Semiconductor products for a variety of systems and markets. These options can be viewed and modified using VP-API functions without requiring knowledge about their implementation. The following is a short list of VP-API configuration options:

- Codec selection
- PCM timeslot and highway assignment
- Pulse digit detection timing
- Automatic state transition upon fault
- Loopback

1.2.2 Architecture

Figure 1–1 on page 4 illustrates a typical software block diagram of a system incorporating the VP-API. The VP-API module provides services to the Application Layer. The VP-API requires the System Services Layer and Hardware Abstraction Layer to operate correctly. This document describes the interfaces between the VP-API and those software modules implemented by the user. The following sections describe each of the blocks shown in Figure 1–1.

Figure 1–1 Software Block Diagram



1.2.2.1 VoicePath API

The VP-API is the core component of Zarlink Semiconductor's VoicePath Software Development Kit (SDK). This software module runs on the host microprocessor that controls one or more Zarlink Semiconductor VTDs. This code is supplied by Zarlink Semiconductor and should not require modification by the application developer.

1.2.2.2 Customer Application

This block represents the user's *line management* module that performs task such as initializing the system, configuring lines, changing line states in response to line events and other inputs, switching digitized voice traffic, etc. These functions may be distributed across a large and complex system, but they are shown as one block in Figure 1–1 for convenience. Zarlink Semiconductor provides example implementations of this layer as part of the VP-SDK.

1.2.2.3 Operating System

This block represents whatever operating system (if any) that the user is running on the host microprocessor. The VP-API does not directly utilize any operating system resources (e.g. queues, semaphores, etc.). However, the application developer may wish to use operating system features such as tasks or shared memory with the VP-API. *Multi-Tasking Applications*, on page 21 covers using the VP-API in a multitasking environment in detail. Note also that the System Services Layer may utilize operating system facilities depending on the application.

1.2.2.4 Hardware Abstraction Layer

The Hardware Abstraction Layer (HAL) provides access to Zarlink Semiconductor devices through the Host Bus or Microprocessor Interface (HBI or MPI) depending on the selected device. The HAL software is platform-dependent and must be implemented by the VP-API user. Zarlink Semiconductor provides example HAL source code with the VP-SDK. Refer to *Hardware Abstraction Layer*, on page 121 for further details.

1.2.2.5 System Services Layer

The System Services Layer abstracts platform-specific functions such as interrupt control and timing services. This layer derives the functions required by the VP-API from the facilities provided by the underlying hardware or operating system. This module is also platform-dependent and must be implemented by the VP-API user. Zarlink Semiconductor provides example System Services Layer source code with the VP-SDK. Refer to *System Services*, on page 111 for further details.

1.2.3 Supported Hardware Configurations

The VP-API supports three general hardware configurations with many possible combinations of Zarlink Semiconductor devices. These three hardware configurations are identified by the type of Zarlink Semiconductor device that the VP-API host microprocessor is directly interfaced to. The following hardware configurations are supported:

- Conventional SLAC
  In this configuration the microprocessor running the VP-API is directly connected to one or more traditional Zarlink Semiconductor SLAC devices. The term *CSLAC* in this document refers to this hardware configuration.
- Voice Control Processor (includes VCP and VCP2 device types)
  In this configuration the microprocessor running the VP-API is interfaced to Zarlink Semiconductor's Voice Control Processor (VCP), which aggregates the control of several CSLAC type of devices each controlling one or more lines.
  – For remainder of this document, VCP and VCP2 will be used interchangeably. Differences are noted when relevant.
- Voice Packet Processor
  In this configuration the microprocessor running the VP-API is interfaced to Zarlink Semiconductor's Voice Packet Processor (VPP), which is essentially an enhanced SLAC device targeted at Voice Over Broadband applications. The term *VPP* in this document refers to this configuration.

There may be many combinations of features, terminations, and devices supported by any one of the above hardware configurations. Table 1–2 on page 6 lists the device configurations supported by the VP-API. The *Primary Device* column indicates which type of device the host microprocessor is directly interfaced to. The *Software Device Type* column indicates which **`VpDeviceType`** constant maps to each Primary Device. The *Part Numbers* column indicates which Zarlink Semiconductor parts or device families apply to each configuration. Finally, the *Configuration Name* column lists the terminology used throughout this document to refer to a specific device configuration. The VP-API supports any combination of the configurations shown in Table 1–2 at run-time, subject to the limitations of the target hardware.

Table 1–2 Supported Device Configurations

Primary Device	Software Device Type (VpDeviceType)	Part Numbers	Configuration Names
CSLAC	VP_DEV_790_SERIES	Le79Q224x Le7922x Le79228x	CSLAC-790 CSLAC-790 CSLAC-790
	VP_DEV_880_SERIES	Le88xxx	CSLAC-880
	VP_DEV_890_SERIES	Le89xxx	CSLAC-890
	VP_DEV_580_SERIES	Le58QLxxx Le58083 Le58DL021	CSLAC-580 CSLAC-580 CSLAC-580
VPP	VP_DEV_VPP_SERIES	Le79610	VPP
VCP	VP_DEV_VCP_SERIES	Le79112+Le79228 Le79112+Le88xxx	VCP-790 VCP-790-NT (No Test) VCP-790-BT (Basic Test) VCP-790-AT (Advanced Test) VCP-880
VCP2	VP_DEV_VCP2_SERIES	Le79114+Le79228 Le79124+Le79238	VCP2-790 VCP2-790-NT (No Test) VCP2-790-BT (Basic Test) VCP2-790-AT (Advanced Test) VCP2-790-ATP (Advanced Test+) VCP2-792 VCP2-792-NT (No Test) VCP2-792-BT (Basic Test) VCP2-792-AT (Advanced Test) VCP2-792-ATP (Advanced Test+)

Some VP-API features are only supported with certain device configurations. This document refers to the specific device Configuration Names shown in Table 1–2 when referring to such a device-specific feature. Note that the VCP device supports different SLAC device families depending on the firmware loaded into the VCP at run-time. The line testing features supported by the VCP-790 also vary according to the firmware load. 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 Termination Types

Software Termination Type (VpTermType)	Devices	Description
FXS Termination Types		
VP_TERM_FXS_GENERIC	CSLAC, VCP, VCP2, VPP	Generic FXS termination
VP_TERM_FXS_ISOLATE	CSLAC-880	FXS termination with SLIC driver isolation relay
VP_TERM_FXS_SPLITTER	CSLAC-880	FXS termination with Splitter and sense path for FEMF is outside the Splitter (connection closest to customer T/R).

Table 1–3 Supported Termination Types

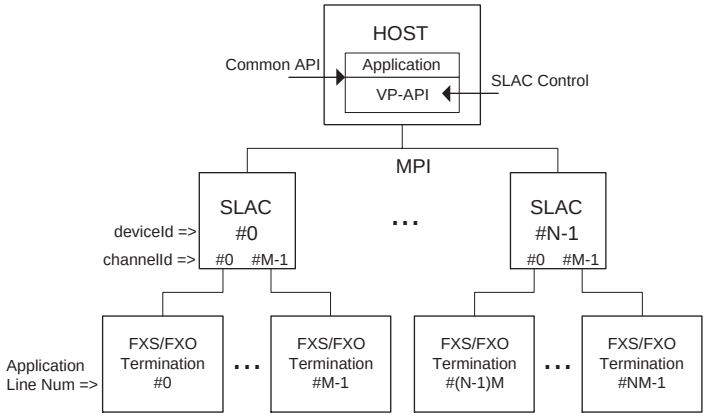
VP_TERM_FXS_TITO_TL_R	VCP-790, VCP2-790	FXS termination using Le792x2 SLIC, with shared test-in, test-out bus access relay and ringing bus access relay and a test load.
VP_TERM_FXS_75181	CSLAC-790, VCP-790, VCP2-790	SLIC Le792x2/ LCAS Le75181; External ringing bus access through the LCAS.
VP_TERM_FXS_75282	CSLAC-790, VCP-790, VCP2-790	SLIC Le792x2 / LCAS Le75282; Test-out bus access relay; Test-in, external ringing bus access through the LCAS.
VP_TERM_FXS_RR	VCP-790, VCP2-790	SLIC Le792x2 with shared ringing, reset relay and with test load
VP_TERM_FXS_TO_TL	CSLAC-790, VCP-790, VCP2-790	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.
FXO Termination Types		
VP_TERM_FXO_GENERIC	CSLAC-880, CSLAC-890	FXO termination for VE880 and VE890 series
VP_TERM_FXO_CUSTOM	CSLAC-580	Custom FXO termination. Case where the user specifies the SLAC, and I/O connections required to set Line States and Line Detector status.
VP_TERM_FXO_DISC	CSLAC-880	FXO termination using disconnect circuitry to improve Disconnect detection on an FXO line when the FXO line is providing Loop Close.

This document describes the VP-API only as it applies to CSLAC devices. Information specific to other device configurations is omitted from this document.

1.2.3.1 Conventional SLAC Device

The system architecture for CSLAC products is shown below.

Figure 1–2 VoicePath System Architecture Example Using Conventional SLAC™ Devices



In a CSLAC architecture, the VP-API 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 per SLAC device depends on the SLAC device used in the design. The VP-API permits the combination of more than one type of SLAC device in a design. Note that with the VP-API, all the SLAC device control happens in the host processor.

The following Zarlink Semiconductor SLAC devices are supported by the VP-API:

- VE790 series (excluding the Le79610 device)
- VE880 series
- VE890 series
- VE580 series

For all supported CSLAC devices, the HAL layer abstracts the basic MPI access functions since these devices all utilize the MPI interface. Refer to Chapter 9 and Chapter 10 for details about System Service and HAL functions required by the VP-API for CSLAC devices.

1.2.4 VP-API Function Summary

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

1.2.4.1 System Configuration

The VP-API 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. A device object can represent any type of VTD (CSLAC, VCP, or VPP) as long as support for that type of VTD is included in the VP-API at compile-time. *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. *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.
- **`VpMakeLineObject()`** – Initlializes a line object and associates it with a device object. Also initializes a line context that refers to the new line object.
- **`VpGetDeviceInfo()`** – Retrieves device-specific information from a device or line context.
- **`VpGetLineInfo()`** – Retrieves line-specific information from a device or line context.
- **`VpFreeLineCtx()`** – Tells the API that the application no longer needs a particular line context.
- **`VpMakeDeviceCtx()`** – Allows creating more than one device context referring to the same device object, which is useful in multitasking applications.
- **`VpMakeLineCtx()`** – Allows creating more than one line context referring to the same line object, which is useful in multitasking applications.

1.2.4.2 Initialization

These functions initialize aspects of the system or perform the configuration required 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.
- **VpSetBatteries()** – Sets the battery settings in the device, used to improve dc feed performance on devices that support this function.
- **VpCalCodec()** – Issues a calibrate analog circuit command to a SLAC device.
- **VpInitRing()** – Sets the ringing cadence, ringing source, and ringing parameters 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.
- **VpInitCustomTerm()** – Configures the SLAC to Line I/O Connections used to set line state and detect line conditions.
- **VpInitProfile()** – Initializes the device's profile tables.

1.2.4.3 Control

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

- **VpSetLineState()** – Sets a line to the requested state.
- **VpSetLineTone()** – Generates a cadenced call progress tone on a FXS line.
- **VpSetRelayState()** – Sets the line relay configuration.
- **VpSetRelGain()** – Sets the relative transmit or receive gain for a 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 for the purposes of implementing Type-II Caller ID.
- **VpStartMeter()** – Starts metering on a FXS line.
- **VpSetOption()** – Sets various device and line specific options.
- **VpDeviceIoAccess()** – Controls device input/output pins.
- **VpVirtualISR()** – Services CSLAC device interrupts.
- **VpApiTick()** – Called periodically to provide a timing basis for the VP-API.
- **VpLowLevelCmd()** – Allows the application to issue low level commands directly to the VTD. This function is an internal debugging tool that should not be used by the application.
- **VpSetBFilter()** – Enables with the coefficients provided or disables the B-Filter.

1.2.4.4 Query/Status

These functions get information and events from the VTD.

- **VpGetEvent()** – Returns events corresponding to a device.
- **VpGetLineStatus()** – Returns the state of a particular status flag for one line.
- **VpGetDeviceStatus()** – Returns the state of a particular status flag for up to 32 lines.
- **VpGetLoopCond()** – Reads a snapshot of loop conditions for an FXS line and returns parameters such as voltage, current, and resistance.
- **VpGetOption()** – Returns the current setting of an option.
- **VpGetLineState()**– Reads the current line state.

- **VpFlushEvents()** – Flushes all outstanding events.
- **VpGetResults()** – Reads the data associated with an event.
- **VpClearResults()** – Discards the data associated with an event.

1.2.4.5 System Support

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

- **VpSysEnterCritical()** – Blocks entry into a critical section of VP-API code through some user-defined method.
- **VpSysExitCritical()** – Marks the end of a VP-API critical code section.
- **VpSysWait()** – Implements a software delay that is only used during CSLAC device initialization.
- **VpSysDisableInt()** – Required by the Level-Triggered interrupt mode to disable the CSLAC device interrupt in the host microprocessor.
- **VpSysEnableInt()** – Required by the Level-Triggered interrupt mode to 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()**, **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.

1.2.4.6 Hardware Abstraction Layer

The Hardware Abstraction Layer (HAL) functions contain the lowest level of code used to directly interface with the VTDs. These functions are platform-specific and must be implemented for the target host processor.

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

1.2.5 Basic VP-API Data Types

Table 1–4 lists the basic data types used extensively throughout the VP-API. These types are defined in the `vp_api_types.h` header file. It may be necessary to change the definition of some of these types depending on the target platform or application preferences. Many other types are defined within the VP-API, but the user should never redefine any VP-API types other than those in `vp_api_types.h`.

Table 1–4 Basic VP-API 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).
uchar	Abbreviated unsigned char.
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.
int8p	Pointer to 8-bit signed integer.
int16p	Pointer to 16-bit signed integer.
int32p	Pointer to 32-bit signed integer.
VpProfilePtrType	Pointer to profile data.
VpImagePtrType	Pointer to VCP/VPP bootable image.
VpVectorPtrType	Pointer to VCP algorithm test vector.
VpPktDataType	VPP packet buffer type.
VpPktDataPtrType	Pointer to VPP packet buffer.

1.2.6 VP-API Function Return Type

The vast majority of VP-API 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 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_MAILBOX_BUSY	Function failed because VCP or VPP device's downstream mailbox is busy. The application should try the same call again later. The VP-API can be configured to repeatedly try the mailbox, which should hide most of these errors. See vp_api_cfg.h. Not applicable to CSLAC devices.
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 requested in call to **VpSetOption()** or **VpGetOption()**.
VP_STATUS_ERR_VERIFY	Returned by **VpBootLoad()** if an error is detected in the VCP or VPP boot-load process. Not applicable to CSLAC devices.
VP_STATUS_DEVICE_BUSY	Resources required to perform the requested function are not available.
VP_STATUS_MAILBOX_EMPTY	Returned by **VpGetResults()** if there is no data in the upstream mailbox.
VP_STATUS_ERR_MAILBOX_DATA	Returned by **VpGetResults()** if the data in the upstream mailbox does not match the expected data type.
VP_STATUS_ERR_HBI	HBI communication with the VCP or VPP failed.
VP_STATUS_ERR_IMAGE	**VpBootLoad()** detected an error in the VCP or VPP boot image. Not applicable to CSLAC devices.
VP_STATUS_IN_CRTCL_SECTN	Another thread is executing a critical section of code, and this thread can not call the requested function simultaneously.
VP_STATUS_DEV_NOT_INITIALIZED	The specified device object is not yet initialized via **VpInitDevice()**.
VP_STATUS_ERR_PROFILE	VP-API 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_INVALID_VOICE_STREAM	This error is returned by voice packet handling functions when an invalid stream identifier is specified.

1.3 API-II SOURCE VERSION NUMBER

Due to feature (device, termination, or function) additions and bug fixes, an application may at runtime want to determine the current version of the API-II release. This can be found in the header file "vp_api.h" from the macro:

#define VP_API_VERSION_TAG (0x020900)

The example shown is for Major release 02, Minor release 09, Revision 00. The Major number indicates a complete interface change such that the application is not likely to compile, and will not work. An application detecting a Major release number change should immediately stop. Major

number = 02 will cover the entire API-II series (with no plans for 03). The Minor release indicates a functional change or addition. Adding a new device, termination type, function, event, or option would justify a new Minor number. This will occur at times, but should not break a well written application (one that uses proper `default` handling and unmasks only those events the application is designed to handle). A Revision change is used for bug fixes only.

1.4 TECHNICAL SUPPORT

For technical support email techsupport@zarlink.com.

CHAPTER

2 PROFILES

2.1 OVERVIEW

Profiles are structures that contain design data to meet specific system requirements. Many VP-API 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. 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 Profile Types

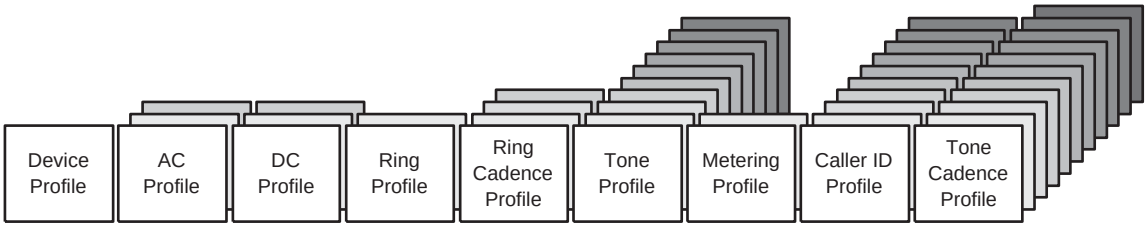
Profile Type	Description
Device Profile	Contains the default start-up values for device-specific configuration options that are normally set at initialization and never changed.
AC Profile	Used for programming the transmission characteristics of the system, the AC Profile holds the VTD programmable gain 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	Contains the commands and data to set up the ringing signal generator of the VTD. Different profiles can be used to vary the ringing characteristics of a line. Options available in the Ringing Profile include ringing waveform, frequency, amplitude, and DC offset.
Metering Profile	Contains the commands and data to set up the metering pulse signal generator of the VTD. The parameters configured include pulse current limits, voltage limits, and frequency.
Tone Profile	Defines the various call progress tones that might be used in a system. Examples tones include: dial tone, busy, ring-back, and reorder.
Ringing Cadence Profile	Defines the various cadences that might be used when ringing a phone.
Tone Cadence Profile	Defines the various cadences that might be used when generating call progress tones.
Caller ID Profile	Defines the off-hook and on-hook signaling protocol for services such as Caller ID and *message waiting* indication.
FXO Configuration Profile	Contains the FXO configuration, including ringing detection frequency window, ringing detection amplitude, and loop open disconnect voltage.
Custom Termination Profile	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 VpInitCustomTerm() prior to initialization of the line..

All profiles should be generated by the Zarlink Semiconductor Profile Wizard, but in cases where run-time modification is desirable, all modifications must be such that the entire profile remains compliant with the Profile Wizard Specification. Note that all Profile Data from Profile Wizard is declared as `const`. So an application that intends to modify Profile Data during run-time should copy the data into an array that can be modified to prevent compiler warnings/errors.

2.3 PROFILE TABLES

The VP-API provides *profile tables* that allow a one or more instances of each type of profile to be pre-loaded into the device. Profiles tables are implemented in the VCP and VPP devices themselves, but for CSLAC devices the profile tables are simulated in software. 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. In this example the number and type of profiles shown applies to the VCP device. The application refers to an individual profile within a profile table by passing a *profile table index* into VP-API 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. *However, overwriting a profile table entry while that profile is in use could result in unusual behavior.* Profiles are typically loaded by the application during VTD initialization. When the host application requires the services of the profile, it simply refers to the profile by its index in the profile table. For example, the application can call **`VpInitProfile()`** to load a ringing cadence profile into the profile table. In subsequent calls to **`VpInitRing()`**, the application can apply this ringing cadence to a line by specifying the profile's index in the profile table in the call to **`VpInitRing()`**. If the application modifies this ringing cadence profile entry by calling **`VpInitProfile()`** again, it should force the VTD to apply the new profile to the lines using the modified profile entry by calling **`VpInitRing()`** again for each line.

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 function profile argument that is not a valid profile table index is automatically interpreted as a pointer to a profile in memory.

As previously mentioned, the number and type of profiles supported varies between the different types of devices. Table 2–2 lists the number and type of profiles supported for the CSLAC device family.

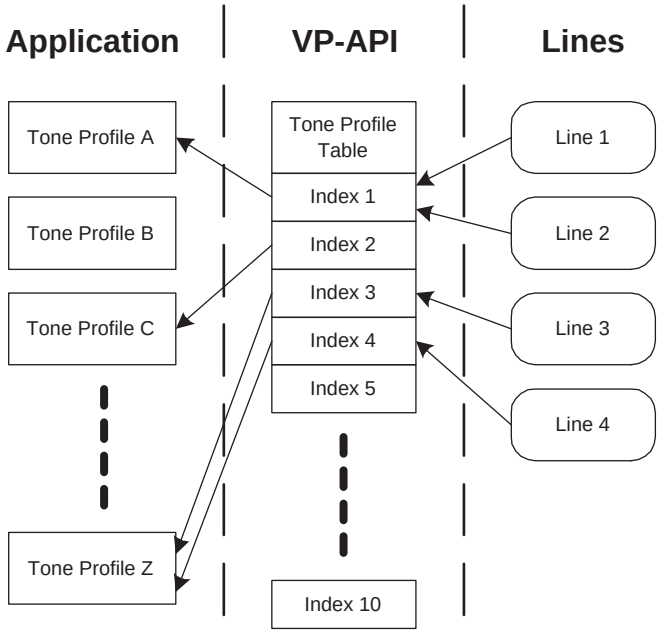
Table 2–2 Profile Table Capacity

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

2.3.1 Soft Profile Tables

CSLAC devices cannot store any profiles internally. Therefore, the VP-API simulates these profile tables in software. These *soft profiles tables* are stored in memory allocated by the application software and referenced by the VP-API as necessary. The VP-API *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 may be using it. This restriction applies to both soft profile tables and any soft profile that is passed to the VP-API by reference (using a profile pointer instead of a profile table index). Figure 2–2 on page 17 illustrates the concept of soft profiles.

Figure 2–2 Soft Profile Table Example



This example shows a hypothetical tone profile table setup with the VP-API 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 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.

2.4 PROFILE FUNCTIONS

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

- *VpInitProfile()*, on page 74
- *VpInitDevice()*, on page 66
- *VpInitLine()*, on page 68
- *VpConfigLine()*, on page 69
- *VpInitRing()*, on page 71
- *VpInitMeter()*, on page 73
- *VpSetLineTone()*, on page 80
- *VpSendCid()*, on page 86

CHAPTER

3 SYSTEM CONFIGURATION FUNCTIONS

3.1 OVERVIEW

The VoicePath API supports the following key features:

- A single host microprocessor can control multiple device types (CSLAC, VCP, VCP2, VPP) and multiple line termination types through a common API.
- The VP-API is compatible with both multi-tasking and single-threaded operating systems.

VoicePath API II introduces the concept of *device objects*, *line objects*, *device contexts*, and *line contexts* to realize these important features. The System Configuration functions described in this chapter manage these objects and contexts, and are summarized below.

- **`VpMakeDeviceObject()`** – Initializes a device object to control a physical device. Also initializes a device context that refers to the new device object.
- **`VpMakeLineObject()`** – Initlializes a line object and associates it with a device object. Also initializes a line context that refers to the new line object.
- **`VpGetDeviceInfo()`** – Retrieves device-specific information from a device or line context.
- **`VpGetLineInfo()`** – Retrieves line-specific information from a device or line context.
- **`VpFreeLineCtx()`** – Tells the API that the application no longer needs a particular line context.
- **`VpMakeDeviceCtx()`** – Allows creating more than one device context referring to the same device object, which is useful in multitasking applications.
- **`VpMakeLineCtx()`** – Allows creating more than one line context referring to the same line object, which is useful in multitasking applications.

3.2 OBJECTS AND CONTEXTS

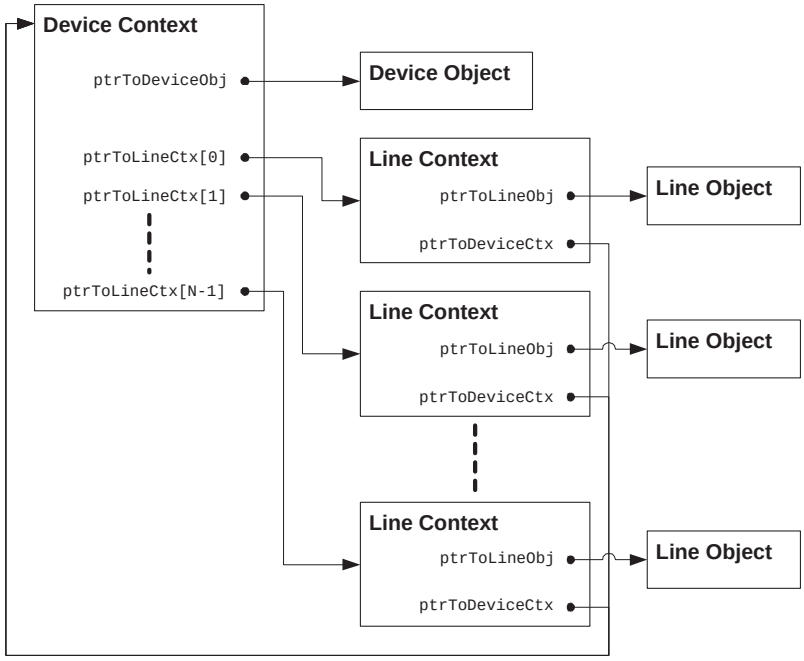
The VP-API itself does not contain any static data related to any line or device. All of the necessary information is stored in device and line object instances, which are indirectly passed to VP-API functions via device and line contexts. There is exactly one device object for every VTD in the system, and there is exactly one line object for every line controlled by each VTD. The actual content of the device and line objects may vary depending on the type of VTD(s) and line termination(s) used in the system.

Device and line contexts are essentially generic "handles" to device and line objects, respectively. Only one type of device context is defined by the VP-API, and an instance of the device context can refer to *any* type of valid device object. Similarly, only one type of line context is defined by the VP-API, and an instance of the line context can refer to *any* type of valid line object. The purpose of the device/line contexts is to hide the specific type of the underlying device/line objects from the top-level VP-API functions. This allows the VP-API functions to take pointers to generic device/line contexts as arguments instead of taking pointers to specific device/line object types. Note that more than one device context can refer to a single device object. Similarly, more than one line context can refer to a single line object.

Figure 3–1 illustrates the interconnections between these objects and contexts in the simplest case, where the VP-API is controlling one primary device that supports *N* lines. In this example there is only one context associated with each object. The generic device context contains a link (pointer) to the device-specific device 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. Note that Figure 3–1 does not precisely depict the actual C implementation of the device context, device object, line context, and line object structures.

Figure 3–1 Device and Line Objects and Contexts



As stated above, the VP-API does not allocate any storage for any type of object or context. Therefore, the application must allocate storage for these structures, then execute VP-API functions to initialize the device and line objects and their respective contexts. It is critical that the application avoid freeing or overwriting the memory space allocated for any object or context until the services of the associated device or line are no longer needed. The contents of the device/line objects and contexts are managed by the VP-API and should never be modified by the application. The application should avoid directly accessing the device/line objects and contexts. The VP-API provides functions such as **`VpGetDeviceInfo()`** and **`VpGetLineInfo()`** that allow the application to get information for the device and line, respectively.

The following table shows device and line object types that are associated with each device family supported by the VP-API. Note that appropriate compile-time switches must be set to include the source code defining the desired device and line object types. These conditional flags are defined in the `vp_api_cfg.h` file.

Table 3–1 Device and Line Objects

VpDeviceType	Compile-Time Switch	Device Object	Line Object
VP_DEV_790_SERIES	VP_CC_790_SERIES	**Vp790DeviceObjectType**	**Vp790LineObjectType**
VP_DEV_VCP_SERIES	VP_CC_VCP_SERIES	**VpVcpDeviceObjectType**	**VpVcpLineObjectType**
VP_DEV_880_SERIES	VP_CC_880_SERIES	**Vp880DeviceObjectType**	**Vp880LineObjectType**
VP_DEV_VPP_SERIES	VP_CC_VPP_SERIES	**VpVppDeviceObjectType**	**VpVppLineObjectType**
VP_DEV_580_SERIES	VP_CC_580_SERIES	**Vp580DeviceObjectType**	**Vp580LineObjectType**
VP_DEV_VCP2_SERIES	VP_CC_VCP2_SERIES	**VpVcp2DeviceObjectType**	**VpVcp2LineObjectType**
VP_DEV_890_SERIES	VP_CC_890_SERIES	**Vp890DeviceObjectType**	**Vp890LineObjectType**

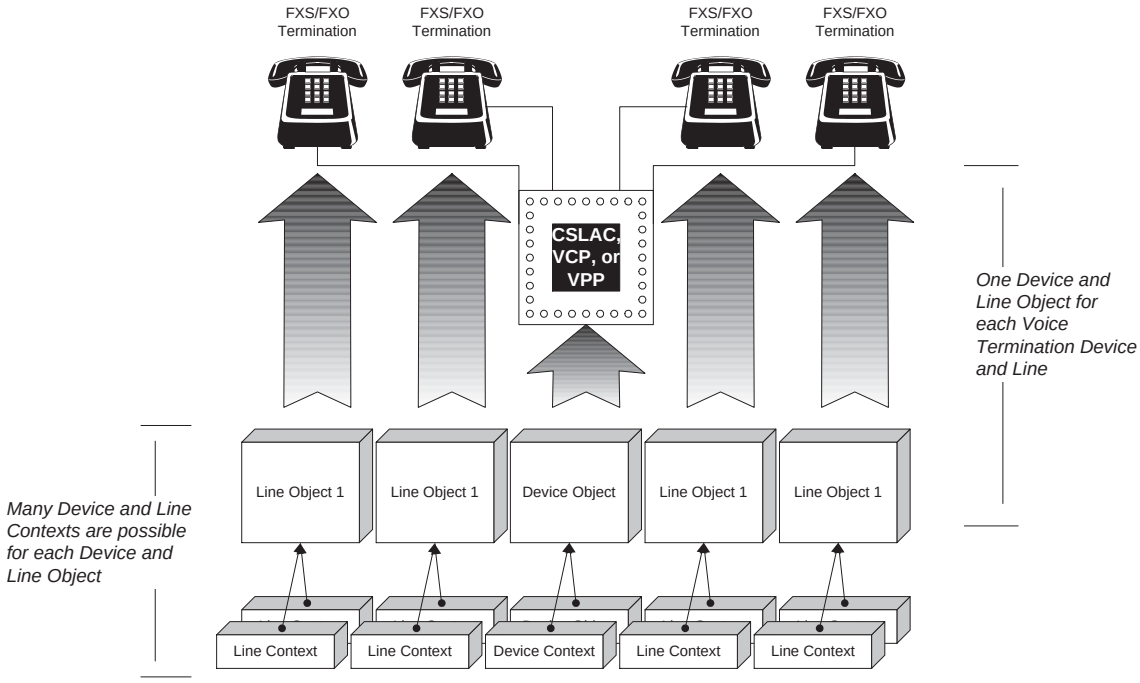
3.3 MULTI-TASKING APPLICATIONS

It may be desirable to have multiple tasks controlling various aspects of the voice path. For example, one process may handle call control, while another management process needs to query line status or perform line testing. In this example, both tasks must share access to the VoicePath API.

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

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 3–2 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 is employed by multiple tasks, one task is responsible for creating the necessary objects. All tasks that use the VP-API 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.

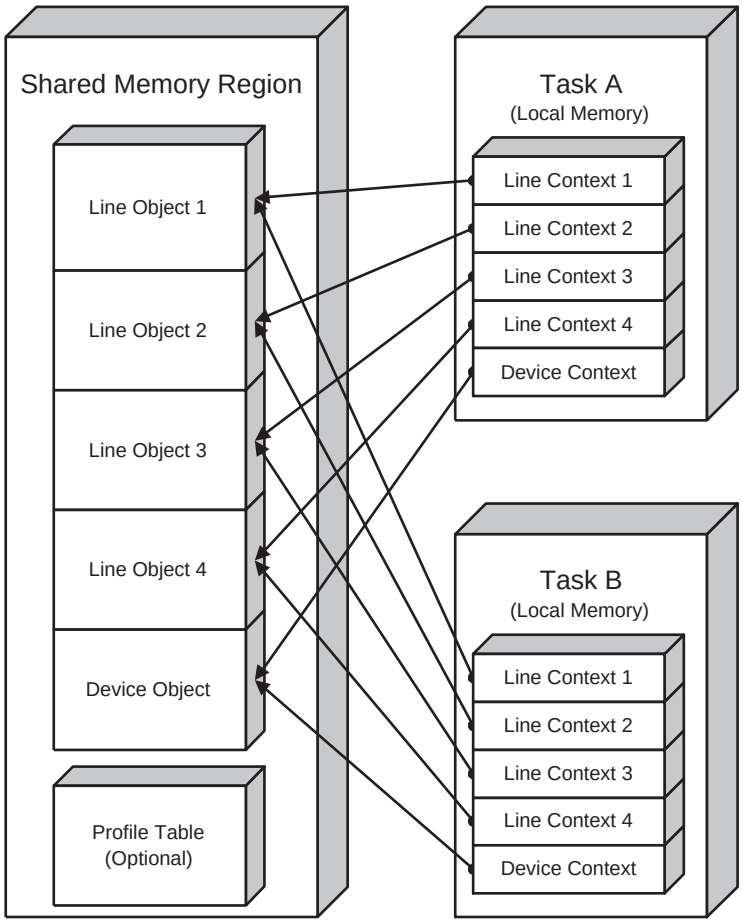
Note that only one task should call **`VpGetEvent()`** and **`VpGetResults()`**. If multiple tasks need to receive VP-API events, a centralized *event dispatcher* task should be implemented to call **`VpGetEvent()`** and **`VpGetResults()`** and forward the events to the desired tasks. Also, for CSLAC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

3.3.1 Multi-Tasking with Protected Memory

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 3–3, on page 22, a shared memory region is shown with two tasks (A and B) both needing access to the VoicePath API. 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 VoicePath API function calls.

For systems using "soft" profiles (see *Soft Profile Tables*, on page 17), 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 3–3 Protected Memory Example



3.4 FUNCTION DESCRIPTIONS

3.4.1 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 creates a device object and device context in memory allocated by the application. It uses the argument `deviceId` to identify the chip select when communicating with the device. The `deviceType` argument selects the type of device and may be any of the **`VpDeviceType`** values defined in Table 3–1. This function takes a pointer to a device context (`pDevCtx`) and a pointer to a device object (`pDevObj`), both of which must point to memory allocated for these structures. **`VpMakeDeviceObject()`** returns `VP_STATUS_INVALID_ARG` if `pDevCtx` or `pDevObj` is `VP_NULL`. Otherwise, both the device context and device object are initialized with appropriate values based on the device type. The device context can be passed to other VP-API functions after it is initialized by this function.

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 calls.*
2. *No other VP-API functions 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 has no way to check this condition, and the application could fail if there is a mismatch.*
4. *The VP-API will not know if more than one device object is created to refer to the same physical device. If more than one device objects are used interchangeably, it could cause unpredictable results. See VpMakeDeviceCtx(), on page 26 for details on creating more than one device context referring to the same device object.*
5. *The definition of `VpDeviceIdType` may be modified as required by the application. The VP-API itself does not interpret `deviceId`, but simply passes `deviceId` down to the HBI/MPI HAL when attempting to access the physical device associated with this device object.*

RETURNS

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

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

All

3.4.2 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 creates a line object and line context in memory allocated by the application. The termination type parameter (`termType`) describes the circuitry associated with the termination. Compile time options are provided so that the user may omit the code for unsupported termination types and hence minimize the compiled code size. The following termination types are defined:

```
Enumeration Data Type: VpTermType:
    VP_TERM_FXS_GENERIC
    VP_TERM_FXS_ISOLATE
    VP_TERM_FXS_TITO_TL_R
    VP_TERM_FXS_75181
    VP_TERM_FXS_75282
    VP_TERM_FXS_RR
    VP_TERM_FXS_TO_TL
    VP_TERM_FXO_GENERIC
    VP_TERM_FXO_DISC
```

Not all termination types are supported by all device families. For a list of termination types and compatible device classes, see *Supported Hardware Configurations*, on page 5.

The `channelId` parameter determines which channel of the VTD is linked to the new line object and line context. Each VTD has a pre-defined limit on the number of channels it supports. For example, the VCP device supports up to 32 channels. The `channelId` argument must be between 0 and (max channels - 1). This function should be called once for each channel managed by each VTD.

This function takes a pointer to a line context (`pLineCtx`) and a pointer to a line object (`pLineObj`), both of which must point to memory allocated for these structures. **`VpMakeLineObject()`** returns `VP_STATUS_INVALID_ARG` if `pLineCtx` or `pLineObj` is `VP_NULL`. Otherwise, both the line context and line object are initialized with appropriate values based on the device type indicated by the device context (`pDevCtx`) argument. The line context can be passed to other VP-API functions after it is initialized by this function.

Notes:

1. *No line-specific VP-API 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 has no way to check this condition, and the application could fail if there is a mismatch. See* Table 3–1 *for a list of device types and matching line object types.*
4. *VCP and VPP devices must be boot-loaded before calling this function because this function may need to communicate with the VTD.*
5. *This function must be called before executing* ***`VpInitDevice()`***. *See* *VpInitDevice()*, on page 66 *for more information.*
6. *The VP-API will not know if more than one line object is created to refer to the same physical line. If more than one line objects are used interchangeably, it could cause unpredictable results. See* *VpMakeLineCtx()*, on page 27 *for more details on creating more than one line context referring to the same line object.*

FUNCTION RETURNS

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

EVENTS GENERATED

None

DEVICES	All
TERMINATIONS	All

3.4.3 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 associates a device object with a device context and initializes the device context. It is useful in multitasking applications where more than one process accesses a single device.

Only one process should initialize the device object by calling **VpMakeDeviceObject()**. It is possible to initialize a device object without initializing a device context by calling the **VpMakeDeviceObject()** function with VP_NULL for the pDevCtx argument.

Subsequently, any process that wants to refer to the same device object could call this function to create a device context that can be used with other VP-API functions. The deviceType argument must indicate the device type that was specified during the device object creation. The pDevObj argument must point to the same device object that was used during its creation.

The pDevCtx argument must contain a pointer to the device context that needs to be initialized. This function call initializes the members of the device context to point to the indicated device object and also initializes the function pointers. Thus, the process that invoked this function has a context that it can use and refer to a global device object.

Notes:

1. *The process of creating new device contexts does not affect any state information that is stored in the device object. The VTD's state is also not changed.*

RETURNS

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

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

All

3.4.4 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 associates a line object with a line context and initializes the line context. It is useful in multitasking applications where more than one process wants to access a single line.

Only one process should initialize the line object by calling **`VpMakeLineObject()`**. It is possible to initialize a line object without initializing a line context by calling the **`VpMakeLineObject()`** function with `VP_NULL` for the `pLineCtx` argument.

Subsequently, any process that wants to refer to the same line object could call this function to create a line context that can be used with other VP-API functions. The `pLineObj` argument must point to the same line object that was used during its creation. The `pDevCtx` argument must point to an existing device context. The `pLineCtx` argument must point to the line context that needs to be initialized.

This function call initializes the members of the line context to point to the indicated line object and also associates the line context with the given device context.

Notes:

The process of creating new line contexts does not affect any state information that is stored in the line object.

RETURNS

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

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

All

3.4.5 VpFreeLineCtx()

SYNTAX

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

DESCRIPTION

Calling this function tells the VP-API that the application no longer requires services from the line associated with `pLineCtx` and would like to reclaim the memory allocated to the line context and line object.

The VP-API needs to know when the services of the line are no longer needed so that it can perform cleanup activities as necessary. The application must call this function if it intends to stop the services of a line and reclaim the memory resources allocated to it.

Note that more than one line context could be associated with one line object (see *Multi-Tasking Applications*, on page 21). This function must be called for all such line contexts to completely release all resources.

Notes:

This function does not alter the state of the physical line. The application is expected to perform any such cleanup tasks, like placing the line in Disconnect mode and disabling the interrupts for the line.

RETURNS

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

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

All

3.4.6 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 */
    uint8 numLines;                         /* Number of lines */
    uint8 revCode;                          /* Silicon revision of the device */
} VpDeviceInfoType;
```

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.

If `pDeviceInfo` 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.

Notes:

The `VpDeviceInfoType::revCode` *field contains valid information only for the CSLAC devices.*

RETURNS

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

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

All

3.4.7 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 Function Return Type*, on page 11

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

All

3.4.8 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. This identifier is not used by the VP-API and is reported along with the event in the **VpGetEvent()** and **VpGetLineInfo()** functions. The line identifier (`VpLineIdType`) is defined as a user defined type that can be modified by the implementation.

RETURNS

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

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

All

CHAPTER

4 OPTIONS

4.1 OVERVIEW

This chapter covers the VoicePath API options that are controlled by the application software at run-time. 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 (CSLAC, VCP, VPP, All) 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()`** and **`VpGetOption()`** functions. See *VpSetOption()*, on page 90 and *VpGetOption()*, on page 105 for complete descriptions of these functions.

4.2 OPTION SUMMARY

`VpOptionIdType` defines the set of options that **`VpSetOption()`** and **`VpGetOption()`** can write and read, respectively. Table 4–1 lists all valid options along with the applicable VTD and termination types. Each option is described in detail later in this chapter. Note that most options are automatically set to default values by the VP-API when the VTD is initialized. The default option settings are defined in `vp_api_cfg.h`.

Some options apply to individual lines, while other options apply to an entire VTD and all lines controlled by it. Global device option names begin with `VP_DEVICE_OPTION_ID_`. All other 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. See *VpSetOption() Behavior*, on page 90 and *VpGetOption() Behavior*, on page 105 for details.

Table 4–1 Available options

Options	Devices	Terminations	Page
VP_DEVICE_OPTION_ID_PULSE	All	FXS	35
VP_DEVICE_OPTION_ID_PULSE2	CSLAC	FXS	36
VP_DEVICE_OPTION_ID_CRITICAL_FLT	All	FXS	37
VP_OPTION_ID_ZERO_CROSS	CSLAC, VCP	FXS	37
VP_DEVICE_OPTION_ID_RAMP2STBY	VCP	FXS	
VP_OPTION_ID_PULSE_MODE	CSLAC, VCP	FXS	38
VP_OPTION_ID_TIMESLOT	CSLAC, VCP	All	38
VP_OPTION_ID_CODEC	CSLAC, VCP	All	38
VP_OPTION_ID_PCM_HWY	CSLAC, VCP	All	39
VP_OPTION_ID_LOOPBACK	All	All	39
Options	**Devices**	**Terminations**	**Page**

Table 4–1 Available options

VP_OPTION_ID_LINE_STATE	All	FXS	40
VP_OPTION_ID_EVENT_MASK	All	All	41
VP_OPTION_ID_RING_CNTRL	CSLAC, VCP	FXS	42
VP_OPTION_ID_DTMF_MODE	VCP, VPP	FXS	
VP_DEVICE_OPTION_ID_DEVICE_IO	All	All	43
VP_OPTION_ID_PCM_TXRX_CNTRL	CSLAC, VCP	All	43
VP_OPTION_ID_US_TRANSCODEC_[0...3]	VPP	All	
VP_OPTION_ID_DS_TRANSCODEC_[0...3]	VPP	All	
VP_OPTION_ID_ECHO_CANCELER	VPP	All	
VP_OPTION_ID_UTD_[1...8]_COEF	VPP	All	
VP_OPTION_ID_CLR_PKT_CNTR	VPP	All	
VP_DEVICE_OPTION_ID_DEV_IO_CFG	VCP2	All	44
VP_OPTION_ID_LINE_IO_CFG	VCP2	All	45
VP_OPTION_ID_DTMF_SPEC	VCP2	All	

4.3 OPTION DESCRIPTIONS

4.3.1 VP_DEVICE_OPTION_ID_PULSE

DESCRIPTION

The pulse options allow the application to set the timing limits used by the VP-API/VTD to decode pulse digits and hook-switch flashes. All of the times are in units of 125 µs. This option is device-specific and applies to all lines controlled by the VTD. The pulse mode option (`VP_OPTION_ID_PULSE_MODE`) determines whether automatic flash and pulse digit decoding is enabled for each line. 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 */
    uint16 onHookMin;                   /* Minimum on-Hook time */
} VpOptionPulseType;
```

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

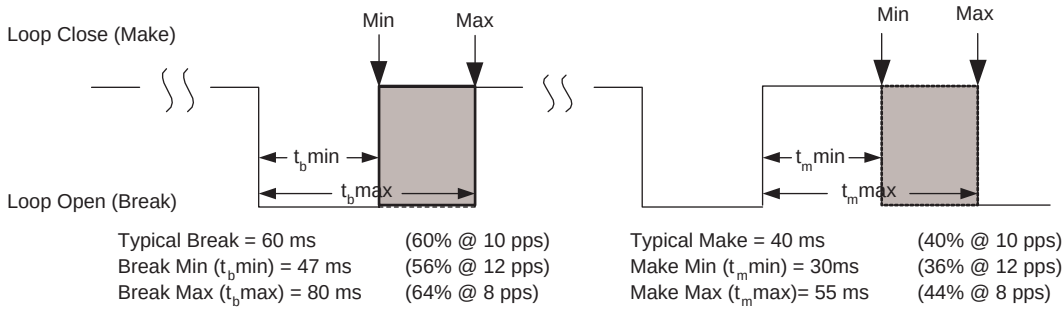
1. `breakMin` < `breakMax` < `flashMin` < `flashMax`
2. `makeMin` < `makeMax` < `interDigitMin`

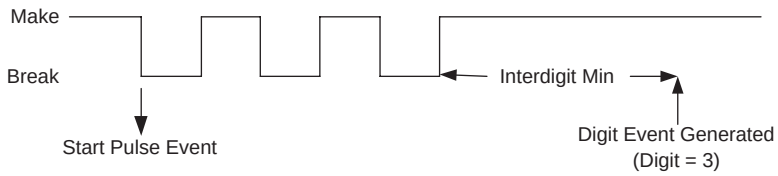
Figure 4–1 shows an example of a typical setting for this option. It also shows the timing relationship between the dialed digit and the events generated.

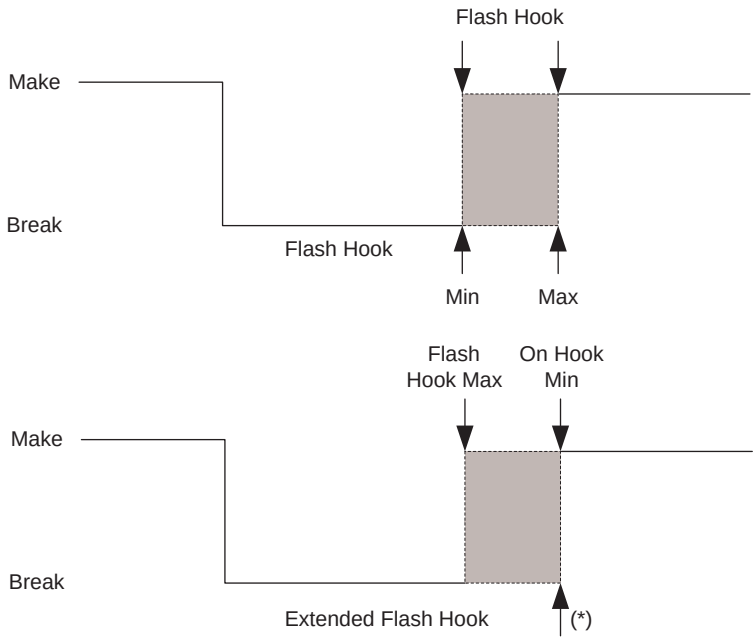
Notes:

1. *Parameter* `onHookMin` *is compiled out of the API-II by default and is not accessible by applications. To enable* `onHookMin`*, the value* EXTENDED_FLASH_HOOK *must be changed to* #define *in file vp_api_cfg.h.*
2. *Parameter* `onHookMin` *is not supported by VCP or VPP devices and therefore forced equal to* (`flashMax+1`) *by the API-II library.*

Figure 4–1 Typical VP Option Pulse Timing Diagrams







* On-Hook Event is generated if line is on-hook longer than the 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

4.3.2 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 details. Default values are set to provide backward compatibility.

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

TERMINATIONS FXS

4.3.3 VP_DEVICE_OPTION_ID_CRITICAL_FLT

DESCRIPTION

This option determines whether or not a line is automatically forced into Disconnect mode when a critical fault (AC, DC, or thermal fault) is detected on that line. Placing the line in Disconnect mode (VP_LINE_DISCONNECT) involves putting the SLIC device into the Disconnect mode and putting the LCAS, if present, in All-Off mode. 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, CSLAC-890,and VCP-880 configurations do not support AC/DC fault detection and therefore automatic disconnect on AC/DC fault.* ***VpSetOption()*** *returns* VP_STATUS_INVALID_ARG *if the application attempts to enable automatic disconnect on AC or DC fault. If the API-II compile time default settings for options used during initialization are inconsistent with this limitation, the API-II will initialize AC/DC fault to FALSE rather than return an error during initialization.*
2. *The relay state (if a relay is defined in the termination type), is set to the least harmful setting allowed by the configuration without applying Ringing to the subscriber line. The application should set the relay to* VP_RELAY_NORMAL *using* ***VpSetRelayState()*** *to return to normal operation.*

DEFAULT

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

DEVICES

All

TERMINATIONS

FXS

4.3.4 VP_OPTION_ID_ZERO_CROSS

DESCRIPTION

The VTD and LCAS (if present) provide automatic Zero-Cross control. The VTD and LCAS will enter and exit the Ringing state when the line crosses the zero-voltage point (on ring entry) or zero-current point (on ring exit). This option is line-specific. This option is passed through a variable of type **VpOptionZeroCrossType**, shown below.

```
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 */
```

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 LCAS timing and control will not occur (no operation of the LCAS will occur at all) and any operation pertaining to LCAS 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 and VCP-790 classes of devices.*

DEFAULT

VP_OPTION_ZC_M4B

DEVICES

CSLAC, VCP

TERMINATIONS

FXS

4.3.5 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 CSLAC, VCP

TERMINATIONS FXS

4.3.6 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. This option is line-specific. Timeslot option parameters are passed through the **VpOptionTimeslotType** structure shown below.

```
typedef struct {
    uint8 tx;                                   /* 8-bit Transmit timeslot */
    uint8 rx;                                   /* 8-bit Receive timeslot */
} VpOptionTimeslotType;
```

Notes:

1. *The transmit direction refers to the data transmission from the VTD towards the network. Receive direction refers to receiving the data from the network to the VTD.*
2. The application should assign timeslots before activating the device's PCM interface. *See VpSetLineState(), on page 78 for more information on which line states activate the PCM highway.*

DEFAULT None

DEVICES CSLAC, VCP

TERMINATIONS All

4.3.7 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              /* Select G.711 A-law PCM encoding */
    VP_OPTION_MLAW              /* Select G.711 Mu-law PCM encoding */
    VP_OPTION_LINEAR            /* Select Linear PCM encoding */
    VP_OPTION_WIDEBAND          /* Select Wideband 16-bit, 16kHz PCM encoding */
```

Notes:

1. *The* VP_OPTION_WIDEBAND *CODEC type is supported for CSLAC-880 and CSLAC-890 devices only.*
2. *The VTD requires two adjacent 8-bit timeslots when in 16-bit linear PCM mode. The timeslot assigned by VP_OPTION_ID_TIMESLOT, on page 38 is the lowest numbered timeslot of the two timeslots occupied by a single channel in linear mode. Therefore, the host must not assign the next adjacent timeslot to any other line. The VTD requires two adjacent 8-bit timeslots at the first programmed timeslot, and two adjacent 8-bit timeslots located at (PCLK Freq / 128*10^3) offset from the programmed timeslot when selecting Wideband mode.*

DEFAULT VP_OPTION_ALAW

DEVICES CSLAC, VCP

TERMINATIONS All

4.3.8 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 */
```

DEFAULT VP_OPTION_HWY_A

DEVICES CSLAC, VCP

TERMINATIONS All

4.3.9 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 */
```

Refer to the appropriate device's *Chip Set User's Guide* for details about each loopback mode.

Notes:

The VCP-880, CSLAC-880, and CSLAC-890 configurations do not support full-digital loopback mode.

DEFAULT VP_OPTION_LB_OFF

DEVICES All

TERMINATIONS All

4.3.10 VP_OPTION_ID_LINE_STATE

DESCRIPTION

If the device controlled by the VP-API supports different battery levels, then the line state option allows for these modifiers to be applied to the appropriate line state. Thus, when a line of the device is subsequently placed in a particular state, the modifiers defined by the line state option are automatically applied where appropriate. This option is 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 of the following states: VP_LINE_ACTIVE, VP_LINE_ACTIVE_POLREV, VP_LINE_TALK, and VP_LINE_TALK_POLREV to any state in that same list and of opposite polarity. Otherwise, the transition between these states is abrupt. The smooth ramp time is approximately 64 ms and is not programmable. Any other state transition (including polarity reversals with the battRev variable is set to TRUE) uses the abrupt mode.

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 */
```

See the device's *Chip Set User's Guide* for further information about polarity and battery levels. Note that the specific modifier options available depend upon the device controlled by the VP-API.

Notes:

1. *This option allows the application to minimize power dissipation due to DC feed by selecting the most appropriate battery supply for each line. Also, using smooth battery reversal can reduce noise on the line when the polarity is reversed.*
2. *The only battery type that is supported for CSLAC-880, CSLAC-890, and VCP-880 class of devices is VP_OPTION_BAT_AUTO.*

DEFAULT

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

DEVICES All

TERMINATIONS FXS

4.3.11 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 */
    uint16 packet;              /* Packet events category masks */
} VpOptionEventMaskType;
```

This structure contains a 16-bit mask for each event category (faults, signaling, etc.). *An event that is masked is not reported to the application.* The composite masks for each event category are created by logically summing (using the *bitwise or* operation) 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**
- **VpPacketEventType**

These event types are described in Chapter 5. 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 90 for more information on how device-specific and line-specific options are applied based on the values of `pDevCtx` and `pLineCtx`.

Notes:

1. *Changing the event mask does not affect events that are already in the event queue, waiting to be delivered to the application. Therefore, it is possible for the application to receive an event even after having masked that type of event.*
2. *Some events are non-maskable. The mask bits corresponding to these events are ignored. Refer to* Chapter 5 *for details.*

DEFAULT All events are masked except for non-maskable events.

DEVICES All

TERMINATIONS All

4.3.12 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 exactly the same setting as the VP_OPTION_ID_ZERO_CROSS option. That is, VP_OPTION_ID_RING_CNTRL and VP_OPTION_ID_ZERO_CROSS modify the same internal VCP/VP-API internal variable. The only difference between these two options is that VP_OPTION_ID_RING_CNTRL allows the application to set the ring-exit debounce time and ring-exit line state as well. See *VP_OPTION_ID_ZERO_CROSS, on page 37* for more information on zero-cross control settings.

The ringExitDbncDur variable determines the ring-exit debounce time when the VTD is implementing ringing cadencing and is specified in units of 125 µs. The ring-exit debounce period starts at the end of each *ringing-on* period of the ringing cadence. 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. No hook-switch events are reported during this period. Ring-exit hook debouncing can be disabled by setting this variable to 0.

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

Notes:

The CSLAC-880, CSLAC-890 and VCP-880 configurations do not allow ringing into an off-hook phone. If ringTripExitSt is set to VP_LINE_RINGING or VP_LINE_RINGING_POLREV, the 880 and 890 device actually puts the line in the VP_LINE_TALK state when ring-trip occurs.

DEFAULT

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

DEVICES

CSLAC, VCP

TERMINATIONS

FXS

4.3.13 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 reserved by the reference design for driving LCAS devices or relays can be controlled by this option. The application must not attempt to control I/O pins that are reserved by the reference design. The VP-API performs no error checking on this option. The I/O pin restrictions for each reference design type are as follows:

- Generic FXS Termination / VCP-790
  Each channel has one available I/O pin (assuming Le79228 SLAC 80-pin package).
  The output type cannot be changed.
- Generic FXS Termination / VCP-880
  Each channel has two available I/O pins.
  The output type can be changed for I/O Pin 0 but not for I/O Pin 1.
- FXS with Test Out Relay / VCP-790
  Each channel has one available I/O pin (assuming Le79228 SLAC 80-pin package).
  The output type cannot be changed.
- FXS with LCAS / VCP-790
  No I/O pins available.

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`) or open-collector/open-drain (`VP_OUTPUT_OPEN_PIN`) output.

DEFAULT None, VTDs retain their hardware reset value.

DEVICES All

TERMINATIONS All

4.3.14 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 both 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 */
```

Notes:

Line state transitions (requested through VpSetLineState(), on page 78) do not change this option.

DEFAULT `VP_OPTION_PCM_BOTH`

DEVICES CSLAC, VCP

TERMINATIONS All

4.3.15 VP_DEVICE_OPTION_ID_DEV_IO_CFG

DESCRIPTION

This option configures the general-purpose input/output (GPIO) pins controlled by a device. New applications should use this option instead of VP_DEVICE_OPTION_ID_DEVICE_IO.

The arguments to this option are passed to VpSetOption() in a `VpOptionDeviceIoConfigType` struct:

```
typedef struct {
    VpOptionLineIoConfigType lineIoConfig[VP_MAX_LINES_PER_DEVICE];
VpOptionDeviceIoConfigType;
```

where `VP_MAX_LINES_PER_DEVICE` is a compile-time option specified in vp_api_cfg.h. The `lineIoConfig` array (indexed by channel ID) contains a `VpOptionLineIoConfigType` struct for each line controlled by the device. For each element of `lineIoConfig`, the array index equals the channel ID. Please see the VP_OPTION_ID_LINE_IO_CFG description on page 45 for futher information about `VpOptionLineIoConfigType`.

DEFAULT

None; VTDs retain their hardware reset value.

DEVICES

VCP2

TERMINATIONS

All

4.3.16 VP_OPTION_ID_LINE_IO_CFG

DESCRIPTION

This option configures the general-purpose input/output (GPIO) pins associated with a particular line. The number of I/O pins configurable through this option for each line is dictated by the reference design (VTD and line termination type) being used. I/O pins that are reserved by the reference design for driving LCAS devices or relays (using the **VpSetRelayState()** function) cannot be configured by this option.

The arguments to this option are passed to VpSetOption() in a `VpOptionLineIoConfigType` struct:

```
typedef struct {
    uint8 direction;
    uint8 outputType;
} VpOptionLineIoConfigType;
```

Up to eight GPIO pins per channel can be configured. Each bit in the `direction` field specifies for an individual GPIO pin whether it is an input (0) or output (1). The corresponding bit in the `outputType` field specifies whether the pin (if configured as an output) is a TTL/CMOS output (`0`) or open-collector/open-drain (1) output.

```
typedef enum {
    VP_IO_INPUT_PIN = 0,
    VP_IO_OUTPUT_PIN = 1
} VpDeviceIoDirectionType;

typedef enum {
    VP_OUTPUT_DRIVEN_PIN = 0,
    VP_OUTPUT_OPEN_PIN   = 1
} VpDeviceOutputPinType;
```

The following table shows for each VTD and line termination type which GPIO pins can be configured by setting the correponding bits in the `direction` and `outputType` fields. Bits that are 0 in the table are ignored in these fields.

VTD(s)	Line Termination Type	Bitmask
VCP2-790	VP_TERM_FXS_GENERIC	00001011
	VP_TERM_FXS_TITO_TL_R	00001000
	VP_TERM_FXS_CO_TL	00001001
	VP_TERM_FXS_75181	00001110
	VP_TERM_FXS_75282	00000000
	VP_TERM_FXS_RDT	00001001
	VP_TERM_FXS_RR	00001001
	VP_TERM_FXS_TO_TL	00001001

DEFAULT None, VTDs retain their hardware reset value.

DEVICES VCP2

TERMINATIONS All

4.3.17 VP_OPTION_ID_DTMF_SPEC

DESCRIPTION This option selects the DTMF detection criteria (based on region specifications) to be associated with a particular line.

The arguments to this option are passed to VpSetOption() in a `VpOptionDtmfSpecType`:

```
Enumeration Data Type: VpOptionDtmfSpecType:
    VP_OPTION_DTMF_SPEC_ATT      /* Q.24 AT&T */
    VP_OPTION_DTMF_SPEC_NTT      /* Q.24 NTT */
    VP_OPTION_DTMF_SPEC_AUS      /* Q.24 Australian */
    VP_OPTION_DTMF_SPEC_BRZL     /* Q.24 Brazilian */
    VP_OPTION_DTMF_SPEC_ETSI     /* ETSI ES 201 235-3 v1.3.1 */
```

DEFAULT `VP_OPTION_DTMF_SPEC_ATT`

DEVICES VCP2

TERMINATIONS All

CHAPTER

5 EVENTS

5.1 OVERVIEW

The VoicePath API 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 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 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 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 functions, the application can provide a handle that is returned with the event carrying the results for that function. The VP-API 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 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 Some events have data associated with them that is larger than the 16-bit event data variable. The VP-API uses the concept of a *mailbox* to pass this data back to the application. The application can call **`VpGetResults()`** to retrieve the event data from the mailbox. The type of the result data is described in this section for each event. If an event has results associated with it but the application does not care about the results, the application must call **`VpClearResults()`** to empty the mailbox anyway. Some events do not have such results, in which case this field is marked "N/A."

DEVICES This field lists the devices (CSLAC, VCP, VPP, All) 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 VP-API functions related to event reporting are described elsewhere in this document.

- *VpGetEvent()*, on page 100
- *VpFlushEvents()*, on page 107
- *VpGetResults()*, on page 108
- *VpClearResults()*, on page 109
- *VpSetOption()*, on page 90 with *VP_OPTION_ID_EVENT_MASK*, on page 41
- *VpGetOption()*, on page 105 with *VP_OPTION_ID_EVENT_MASK*, on page 41

5.2 EVENT SUMMARY

Table 5–1 on page 48 lists all events that the VP-API can generate. The events are organized into categories, and these categories are defined in the software by the **`VpEventCategoryType`** enumeration. Notice that some events apply only to certain device types. The application will never

receive an event that is not generated by the device type(s) used in the system. Some event are device-specific, and some events are line-specific. The names of device-specific events begin with VP_DEV_EVID_, while the names of line-specific events begin with VP_LINE_EVID_.

Table 5–1 List of VP-API Events

Event ID	Devices	Terminations	Page
Fault Events			
VP_DEV_EVID_BAT_FLT	All	All	51
VP_DEV_EVID_CLK_FLT	All	All	51
VP_LINE_EVID_THERM_FLT	All	FXS	51
VP_LINE_EVID_DC_FLT	CSLAC-790, VCP-790, VPP	FXS	52
VP_LINE_EVID_AC_FLT	CSLAC-790, VCP-790, VPP	FXS	52
VP_DEV_EVID_EVQ_OFL_FLT	VCP, VPP	All	
VP_DEV_EVID_WDT_FLT	VCP, VPP	All	
Signaling Events			
VP_LINE_EVID_HOOK_OFF	All	FXS	53
VP_LINE_EVID_HOOK_ON	All	FXS	53
VP_LINE_EVID_GKEY_DET	All	FXS	53
VP_LINE_EVID_GKEY_REL	All	FXS	54
VP_LINE_EVID_FLASH	All	FXS	54
VP_LINE_EVID_STARTPULSE	All	FXS	54
VP_LINE_EVID_DTMF_DIG	All	FXS	55
VP_LINE_EVID_PULSE_DIG	All	FXS	55
VP_LINE_EVID_MTONE	VCP, VPP	FXS	
VP_DEV_EVID_TS_ROLLOVER	All	All	55
VP_LINE_EVID_US_TONE_DETECT	VPP	All	
VP_LINE_EVID_DS_TONE_DETECT	VPP	All	
VP_DEV_EVID_SEQUENCER	VPP	All	
VP_LINE_EVID_EXTD_FLASH	CSLAC	FXS	54
Response Events			
VP_DEV_EVID_BOOT_CMP	VCP, VPP	All	
VP_LINE_EVID_LLCMD_TX_CMP	All	All	56
VP_LINE_EVID_LLCMD_RX_CMP	All	All	56
VP_DEV_EVID_DNSTR_MBOX	VCP, VPP	All	
VP_LINE_EVID_RD_OPTION	All	All	56
VP_LINE_EVID_RD_LOOP	CSLAC-790, VCP, VPP	FXS	57
VP_EVID_CAL_CMP	CSLAC-790, VCP-790	All	58

VP_EVID_CAL_BUSY	CSLAC-790, VCP-790	All	58
VP_LINE_EVID_GAIN_CMP	VCP, CSLAC-880, CSLAC-890	All	59
VP_DEV_EVID_DEV_INIT_CMP	All	All	59
VP_LINE_EVID_LINE_INIT_CMP	All	All	59
VP_DEV_EVID_IO_ACCESS_CMP	All	All	60
VP_LINE_EVID_LINE_IO_RD_CMP	VCP2	All	60
VP_LINE_EVID_LINE_IO_WR_CMP	VCP2	All	60
Test Events			
VP_LINE_EVID_TEST_CMP	VCP-790-BT, VCP-790-AT, VPP	FXS	
VP_LINE_EVID_DTONE_DET	VCP-790-AT	FXS	
VP_LINE_EVID_DTONE_LOSS	VCP-790-AT	FXS	
VP_DEV_EVID_STEST_CMP	VCP	All	
VP_DEV_EVID_CHKSUM	VCP, VPP	All	
Process Events			
VP_LINE_EVID_MTR_CMP	All	FXS	61
VP_LINE_EVID_MTR_ABORT	All	FXS	61
VP_LINE_EVID_MTR_CAD	VCP-790	FXS	
VP_LINE_EVID_CID_DATA	All	FXS	61
VP_LINE_EVID_RING_CAD	All	FXS	62
VP_LINE_EVID_SIGNAL_CMP	CSLAC, VCP	All	62
VP_LINE_EVID_TONE_CAD	All	All	62
FXO Events			
VP_LINE_EVID_RING_ON	CSLAC	FXO	63
VP_LINE_EVID_RING_OFF	CSLAC	FXO	63
VP_LINE_EVID_LIU	CSLAC	FXO	63
VP_LINE_EVID_LNIU	CSLAC	FXO	63
VP_LINE_EVID_FEED_DIS	CSLAC	FXO	63
VP_LINE_EVID_FEED_EN	CSLAC	FXO	64
VP_LINE_EVID_DISCONNECT	CSLAC	FXO	64
VP_LINE_EVID_RECONNECT	CSLAC	FXO	64
VP_LINE_EVID_POLREV	CSLAC	FXO	64
Packet Events			
VP_LINE_EVID_US_PKT_RDY	VPP	All	
VP_LINE_EVID_NEED_DS_PKT	VPP	All	

VP_LINE_EVID_PKT_ERROR	VPP	All	
VP_LINE_EVID_PKT_LOST	VPP	All	
VP_LINE_EVID_RD_PKT_STATS	VPP	All	

5.3 FAULT EVENTS

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

5.3.1 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 N/A

EVENT DATA Event data indicates the source of the battery fault and can be any of the values shown below. See *Battery Name Interpretation*, on page 58 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 N/A

DEVICES All

TERMINATIONS All

5.3.2 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 N/A

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

RESULTS N/A

DEVICES All

TERMINATIONS All

5.3.3 VP_LINE_EVID_THERM_FLT

DESCRIPTION This event occurs when a thermal fault is detected or is no longer detected. The line may be forced into the Disconnect mode when this event occurs, depending on how the `VP_DEVICE_OPTION_ID_CRITICAL_FLT` option is configured (see Section 4.3.3). Refer to the appropriate *Chip Set User's Guide* for details on the thermal fault condition.

T.S. OR HANDLE N/A

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

RESULTS N/A

DEVICES All

TERMINATIONS FXS

5.3.4 VP_LINE_EVID_DC_FLT

DESCRIPTION	This event occurs when a DC fault is detected or is no longer detected. The line may be forced into the Disconnect mode when this event occurs, depending on how VP_DEVICE_OPTION_ID_CRITICAL_FLT is configured (see Section 4.3.3). Refer to the appropriate *Chip Set User's Guide* for details on the DC fault condition.
T.S. OR HANDLE	N/A
EVENT DATA	Event data bit 0 indicates whether the fault condition is present (1) or absent (0).
RESULTS	N/A
DEVICES	CSLAC-790, VCP-790, VPP
TERMINATIONS	FXS

5.3.5 VP_LINE_EVID_AC_FLT

DESCRIPTION	This event occurs when a AC fault is detected or is no longer detected. The line may be forced into the Disconnect mode when this event occurs, depending on how VP_DEVICE_OPTION_ID_CRITICAL_FLT is configured (see Section 4.3.3). Refer to the appropriate *Chip Set User's Guide* for details on the AC fault condition.
T.S. OR HANDLE	N/A
EVENT DATA	Event data bit 0 indicates whether the fault condition is present (1) or absent (0).
RESULTS	N/A
DEVICES	CSLAC-790, VCP-790, VPP
TERMINATIONS	FXS

5.4 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.

5.4.1 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*, on page 35 for details.

If pulse-digit decoding is enabled, this event occurs when the VTD/VP-API determines that the line is off-hook beyond the pulse-digit make period. In other words, this event does not occur during pulse dialing.

If pulse-digit decoding is disabled, this event occurs every time the VTD/VP-API detects the off-hook condition. This event is reported during pulse dialing.

T.S. OR HANDLE Time stamp

EVENT DATA N/A

RESULTS N/A

DEVICES All

TERMINATIONS FXS

5.4.2 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*, on page 35 for details.

If pulse-digit decoding is enabled, this event occurs when the VTD/VP-API 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 detects the on-hook condition. This event is reported during pulse dialing and a hook-switch flash.

T.S. OR HANDLE Time stamp

EVENT DATA N/A

RESULTS N/A

DEVICES All

TERMINATIONS FXS

5.4.3 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 Time stamp

EVENT DATA N/A

RESULTS N/A

DEVICES All

TERMINATIONS FXS

5.4.4 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	Time stamp
EVENT DATA	N/A
RESULTS	N/A
DEVICES	All
TERMINATIONS	FXS

5.4.5 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*, on page 35 for details.
T.S. OR HANDLE	Time stamp
EVENT DATA	N/A
RESULTS	N/A
DEVICES	All
TERMINATIONS	FXS

5.4.6 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*, on page 35 for details.
T.S. OR HANDLE	Time stamp
EVENT DATA	N/A
RESULTS	N/A
DEVICES	All
TERMINATIONS	FXS

5.4.7 VP_LINE_EVID_EXTD_FLASH

DESCRIPTION	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*, on page 35 for details.
T.S. OR HANDLE	Time stamp
EVENT DATA	N/A
RESULTS	N/A
DEVICES	CSLAC
TERMINATIONS	FXS

5.4.8 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 88 for more information.

T.S. OR HANDLE Time stamp

EVENT DATA Event data bits 3 to 0 contain the received digit information, which can be decoded by comparing with the **`VpDigitType`** (see *VpSetLineTone()*, on page 80) enumeration constants. Bit 4 indicates whether this event corresponds to the start (1) or the end (0) of the DTMF digit.

RESULTS N/A

DEVICES All

TERMINATIONS FXS

5.4.9 VP_LINE_EVID_PULSE_DIG

DESCRIPTION This event occurs when a pulse digit is detected.

T.S. OR HANDLE '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 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 `(if` VP_DEVICE_OPTION_ID_PULSE2 `is supported by the device).`

RESULTS N/A

DEVICES All

TERMINATIONS FXS

5.4.10 VP_DEV_EVID_TS_ROLLOVER

DESCRIPTION This event occurs when the VP-API 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 N/A

EVENT DATA N/A

RESULTS N/A

DEVICES All

TERMINATIONS All

5.5 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.

5.5.1 VP_LINE_EVID_LLCMD_TX_CMP

DESCRIPTION	This event occurs after a low-level write command is issued to a device. Refer to *VpLowLevelCmd()*, on page 94 for information on issuing low-level commands.
T.S. OR HANDLE	N/A
EVENT DATA	N/A
RESULTS	N/A
DEVICES	All
TERMINATIONS	All

5.5.2 VP_LINE_EVID_LLCMD_RX_CMP

DESCRIPTION	This event occurs after a low-level read command is issued to a device and the resulting data is available. Refer to *VpLowLevelCmd()*, on page 94 for information on issuing low-level commands. This event is non-maskable.
T.S. OR HANDLE	Handle
EVENT DATA	N/A
RESULTS	The application must call **VpGetResults()** with a pointer to a byte (uint8) buffer into which the resulting data is copied. The application is responsible for allocating enough storage for the data string and is responsible for interpreting the data. Note that the number of bytes copied into the result buffer is equivalent to the len (length) argument in the original call to **VpLowLevelCmd()**.
DEVICES	All
TERMINATIONS	All

5.5.3 VP_LINE_EVID_RD_OPTION

DESCRIPTION	This event occurs as a result of calling **VpGetOption()** and indicates that the VP-API has retrieved the requested option setting from the VTD. See *VpGetOption()*, on page 105 for information on that function. This event is non-maskable.
T.S. OR HANDLE	Handle
EVENT DATA	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 33 for the complete list of VP-API options.
RESULTS	The data type 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 4 describes the result type for each VP-API option.
DEVICES	All
TERMINATIONS	All

5.5.4 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 104 for information on that function.

This event is non-maskable.

T.S. OR HANDLE

Handle

EVENT DATA

N/A

RESULTS

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 Table 5–2. Note that these equations assume that the application is using VE790 or VE880 devices with the recommended external components. Refer to the appropriate Zarlink Semiconductor documentation (*Chip Set User's Guide* or *Data Sheet*) for recommended application circuits.

Table 5–2 Loop Condition Results Conversion

Loop Condition	VCP-790 / CSLAC-790	VCP-880
Loop Resistance	(`rloop` / 32768) x 11.67 kΩ	(`rloop` / 32768) x 16 kΩ
Longitudinal Current	(`ilg` / 32768) x 101.32 mA	(`ilg` / 32768) x 42 mA
Metallic Current	(`imt` / 32768) x 101.66 mA	(`imt` / 32768) x 60 mA
Metallic Voltage	(`vsab` / 32768) x 153 V	(`vsab` / 32768) x 240 V
Battery *N* Voltage	(`vbat`*N* / 32768) x 99.2 V	(`vbat`*N* / 32768) x 240 V
Metering Signal Peak Voltage	(`mspl` / 32768) x 10.2 V	(`mspl` / 32680) x 6.52 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 5–3 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. In the *Normal* configuration the high battery sense (SHB) is connected to the high battery supply (VBH) and the low battery sense (SLB) is connected to the low battery supply (VBL). In the *Crossed* configuration the high battery sense (SHB) is connected to the low battery supply (VBL) and the low battery sense (SLB) is connected to the high battery supply (VBH).

Table 5–3 Battery Name Interpretation

Device Type	Normal Config.		Crossed Config.		
	vBat1	**vBat2**	**vBat1**	**vBat2**	**vBat3**
CSLAC-790	VBL	VBH	VBH	VBL	VBP
VPP	VBL	VBH	VBH	VBL	VBP
VCP-790	VBL	VBH	VBH	VBL	VBP
VCP-880	VBL	VBH	VBH	VBL	VBM

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

DEVICES CSLAC-790, VCP, VPP

TERMINATIONS FXS

5.5.5 VP_EVID_CAL_CMP

DESCRIPTION This event occurs as a result of calling **`VpCalCodec()`** or **`VpCalLine()`** and indicates that the requested device or line is calibrated. Note that disabling (masking) this event blocks this event for both the **`VpCalCodec()`** and **`VpCalLine()`** functions. See *VpCalCodec()*, on page 70 for information on these functions.

T.S. OR HANDLE N/A

EVENT DATA N/A

RESULTS N/A

DEVICES CSLAC-790, VCP-790

TERMINATIONS All

5.5.6 VP_EVID_CAL_BUSY

DESCRIPTION This event occurs as a result of calling **`VpCalCodec()`** or **`VpCalLine()`** and indicates that the requested device or line can not be calibrated at this time. In the case of a codec calibration, the line number returned with this event is the lowest-numbered line controlled by the target SLAC device. Note that disabling (masking) this event blocks this event for both the **`VpCalCodec()`** and **`VpCalLine()`** functions. See *VpCalCodec()*, on page 70 for information on these functions.

T.S. OR HANDLE N/A

EVENT DATA N/A

RESULTS N/A

DEVICES CSLAC-790, VCP-790

TERMINATIONS All

5.5.7 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 82 for information on that function.

This event is non-maskable.

T.S. OR HANDLE Handle

EVENT DATA N/A

RESULTS 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 VCP, CSLAC-880, CSLAC-890

TERMINATIONS All

5.5.8 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 66 for information on that function.

This event is non-maskable.

T.S. OR HANDLE N/A

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

RESULTS N/A

DEVICES All

TERMINATIONS All

5.5.9 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 68 for information on that function.

This event is non-maskable.

T.S. OR HANDLE N/A

EVENT DATA N/A

RESULTS N/A

DEVICES All

TERMINATIONS All

5.5.10 VP_DEV_EVID_IO_ACCESS_CMP

DESCRIPTION This event occurs as a result of calling **`VpDeviceIoAccess()`** or **VpDeviceIoAccessExt()** nd indicates that the requested I/O access is done. See *VpDeviceIoAccess()*, on page 91 and *VpGetDeviceStatusExt()*, on page 109 for information on these functions.

This event is non-maskable.

T.S. OR HANDLE N/A

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

RESULTS 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 91 describes how to map the I/O read results in these variables to physical I/O pin logic states.

For VpDeviceIoAccessExt(), a `VpDeviceIoAccessExtType` struct should be passed. See *VpGetDeviceStatusExt()*, on page 109 for a description of this struct.

DEVICES All

TERMINATIONS All

5.5.11 VP_LINE_EVID_LINE_IO_RD_CMP

DESCRIPTION This event occurs as a result of calling **`VpLineIoAccess()`** with `direction = VP_IO_READ` and indicates that the requested operation is complete. See *VpLineIoAccess()*, on page 96 for information on this function.

This event is non-maskable.

T.S. OR HANDLE Handle

EVENT DATA N/A

RESULTS The application should call **`VpGetResults()`** with a pointer to a `VpLineIoAccessType` struct to receive the results associated with this event. See *VpLineIoAccess()*, on page 96 for a description of this struct.

DEVICES VCP2

TERMINATIONS All

5.5.12 VP_LINE_EVID_LINE_IO_WR_CMP

DESCRIPTION This event occurs as a result of calling **`VpLineIoAccess()`** with `direction = VP_IO_WRITE` and indicates that the requested operation is complete. See *VpLineIoAccess()*, on page 96 for information on this function.

T.S. OR HANDLE Handle

EVENT DATA N/A

RESULTS N/A

DEVICES VCP2

TERMINATIONS All

5.6 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.

5.6.1 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. See *VpStartMeter()*, on page 89 for information on that function.

T.S. OR HANDLE N/A

EVENT DATA N/A

RESULTS N/A

DEVICES All

TERMINATIONS FXS

5.6.2 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. See *VpStartMeter()*, on page 89 for information on that function.

T.S. OR HANDLE N/A

EVENT DATA Event data returns the number of complete metering pulses transmitted. If a pulse was interrupted it is not included in this count.

RESULTS N/A

DEVICES All

TERMINATIONS FXS

5.6.3 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 72, *VpSendCid()*, on page 86, or *VpContinueCid()*, on page 87 for information on these functions.

T.S. OR HANDLE N/A

EVENT DATA 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 Caller ID data buffer has 16 bytes left to transmit. The application can load up to 16 more bytes of Caller ID data into the buffer when this event occurs.

The VP_CID_DATA_TX_DONE event occurs after the VTD transmits the last byte of Caller ID data and completes the Caller ID sequence. This event also occurs if Caller ID transmission is aborted due to off-hook detection.

RESULTS N/A

DEVICES All

TERMINATIONS FXS

5.6.4 VP_LINE_EVID_RING_CAD

DESCRIPTION This event occurs when the VP-API/VTD performs automatic ringing cadencing. It notifies the application of ringing-on and ringing-off state transitions as the VP-API/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 71 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 N/A

EVENT DATA 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 N/A

DEVICES All

TERMINATIONS FXS

5.6.5 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 83 for information on that function.

T.S. OR HANDLE N/A, except for `VP_SENDSIG_MOMENTARY_LOOP_OPEN` which indicates '1' if a parallel off-hook is detected, '0' otherwise.

EVENT DATA 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 83.

RESULTS N/A

DEVICES CSLAC, VCP

TERMINATIONS All

5.6.6 VP_LINE_EVID_TONE_CAD

DESCRIPTION This event occurs when the VP-API/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 N/A

EVENT DATA N/A

RESULTS N/A

DEVICES All

TERMINATIONS All

5.7 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.

5.7.1 VP_LINE_EVID_RING_ON

DESCRIPTION	This event occurs when ringing is detected on a FXO line.
T.S. OR HANDLE	Time stamp
EVENT DATA	N/A
RESULTS	N/A
DEVICES	CSLAC
TERMINATIONS	FXO

5.7.2 VP_LINE_EVID_RING_OFF

DESCRIPTION	This event occurs when ringing is no longer detected on a FXO line.
T.S. OR HANDLE	Time stamp
EVENT DATA	N/A
RESULTS	N/A
DEVICES	CSLAC
TERMINATIONS	FXO

5.7.3 VP_LINE_EVID_LIU

DESCRIPTION	This event indicates that another FXO is using the line (e.g. a parallel telephone went off-hook).
T.S. OR HANDLE	Time stamp
EVENT DATA	N/A
RESULTS	N/A
DEVICES	CSLAC
TERMINATIONS	FXO

5.7.4 VP_LINE_EVID_LNIU

DESCRIPTION	This event indicates that the line is no longer being used by a parallel FXO termination.
T.S. OR HANDLE	Time stamp
EVENT DATA	N/A
RESULTS	N/A
DEVICES	CSLAC
TERMINATIONS	FXO

5.7.5 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	Time stamp
EVENT DATA	N/A
RESULTS	N/A
DEVICES	CSLAC
TERMINATIONS	FXO

5.7.6 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	Time stamp
EVENT DATA	N/A
RESULTS	N/A
DEVICES	CSLAC
TERMINATIONS	FXO

5.7.7 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	Time stamp
EVENT DATA	N/A
RESULTS	N/A
DEVICES	CSLAC
TERMINATIONS	FXO

5.7.8 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	Time stamp
EVENT DATA	N/A
RESULTS	N/A
DEVICES	CSLAC
TERMINATIONS	FXO

5.7.9 VP_LINE_EVID_POLREV

DESCRIPTION	This event occurs when the FXO port detects loop voltage polarity reversal.
T.S. OR HANDLE	Time stamp
EVENT DATA	'0' if the polarity transition was from reverse to normal, '1' if the polarity transition was from normal to reverse.
RESULTS	N/A
DEVICES	CSLAC
TERMINATIONS	FXO

CHAPTER

6 INITIALIZATION FUNCTIONS

6.1 OVERVIEW

This chapter discusses VP-API 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.
- **`VpSetBatteries()`** – Sets the battery settings in the device, used to improve dc feed performance on devices that support this function.
- **`VpCalCodec()`** – Issues a calibrate analog circuit command to a SLAC device.
- **`VpInitRing()`** – Sets the ringing cadence, ringing source, and ringing parameters 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.
- **`VpInitCustomTerm()`** – Configures the SLAC to Line I/O Connections used to set line state and detect line conditions.
- **`VpInitProfile()`** – Initializes the device's profile tables.

6.2 FUNCTION DESCRIPTIONS

6.2.1 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 Zarlink Semiconductor reference designs, which gives the application the maximum flexibility.

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

EVENTS GENERATED None

DEVICES CSLAC-580

TERMINATIONS FXS

6.2.2 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 passed device context. This includes performing the recommended power-up sequence specified in the device's *Chip Set User's Guide*, including executing **`VpCalCodec()`** for each line as necessary. This function should be called after creating the device and line objects via **`VpMakeDeviceObject()`** and **`VpMakeLineObject()`**, respectively. *All device and line options are overwritten with their default values as a result of calling this function.*

The `pDevProfile` parameter takes a pointer to a Device Profile and is required for both FXS and FXO implementations. This function returns an error code if `pDevProfile` does not point to a valid profile.

The VTD requires parameters for operation that may include AC characteristics, DC feed options, and ringing parameters for its FXS terminations. The `pAcProfile`, `pDcProfile`, and `pRingProfile` arguments of this function provide the required parameters for the configuration of FXS lines. If a VTD supports FXO terminations, then it requires a different set of parameters for FXO AC characteristics and FXO configuration. The `pFxoAcProfile` and `pFxoCfgProfile` arguments provide the FXO line configuration parameters for the VTD. Note that if there are no FXO or FXS lines in the application, the profile pointers arguments corresponding to the unused termination type should equal `VP_PTABLE_NULL`.

The profile types accepted by this function are described in *Profile Types*, on page 15. For each of these profiles, the application can supply either a pointer to a valid profile or a profile table index. Remember that valid profiles must be loaded into the profile table before a profile table index can be used. See *Profiles*, on page 15.

If `VP_PTABLE_NULL` is passed for any of the profiles (other than the Device Profile), then the device uses its default parameters for the associated profile. *No overall system performance is guaranteed when using default parameters*. With the exception of the Device Profile, any profiles not set using this function can still be set by the application at a later time by calling **`VpInitLine()`** or **`VpConfigLine()`**.

Upon completion of the this function, all initialized FXS lines are placed in the `VP_LINE_DISCONNECT` line state, thus disconnecting the lines from the loop. The FXO lines are set to the `VP_LINE_FXO_LOOP_OPEN` line state.

The following relay states are used for various line termination types as the relay states at the end of this function.

FXS Line Termination Type	Relay State
VP_TERM_FXS_GENERIC	VP_RELAY_NORMAL
VP_TERM_FXS_ISOLATE	VP_RELAY_NORMAL
VP_TERM_FXS_TITO_TL_R	VP_RELAY_NORMAL
VP_TERM_FXS_75181	VP_RELAY_NORMAL
VP_TERM_FXS_75282	VP_RELAY_RESET
VP_TERM_FXS_RR	VP_RELAY_NORMAL
VP_TERM_FXS_TO_TL	VP_RELAY_NORMAL

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

EVENTS GENERATED

VP_DEV_EVID_DEV_INIT_CMP, on page 59

DEVICES

All

TERMINATIONS

All

6.2.3 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 attributes of the line associated with `pLineCtx`. The application can use this function to reset a line without affecting the other lines.

This function places the line in a known state. The FXS lines are placed in the `VP_LINE_DISCONNECT` state and the FXO lines are placed in the `VP_LINE_FXO_LOOP_OPEN` state. The relay states at the end of this function are same as those described in the function *VpInitDevice()*, on page 66.

This function uses profiles in the same manner as *VpInitDevice()*, on page 66. See Chapter 2, on page 15 for a complete description of the profiles.

All profile settings for the target line are lost when the line is reset. Therefore, this function should be called with appropriate pointers to profiles or profile indices. Alternatively, the **`VpConfigLine()`** function could be called later to set the profiles for this line.

If the target line is an FXS termination, then this function assumes that the `pDcFeedOrFxoCfgProfile` argument points to a DC profile. Otherwise, this function assumes that the `pDcFeedOrFxoCfgProfile` argument points to a FXO configuration profile, and the `pRingProfile` argument is ignored.

Notes:

1. *The following options are eventually reset to default values as a result of calling this function: `VP_OPTION_ID_ZERO_CROSS`, `VP_OPTION_ID_PULSE_MODE`, `VP_OPTION_ID_CODEC`, `VP_OPTION_ID_PCM_HWY`, `VP_OPTION_ID_LOOPBACK`, `VP_OPTION_ID_LINE_STATE`, `VP_OPTION_ID_RING_CNTRL`, `VP_OPTION_ID_DTMF_MODE` and `VP_OPTION_ID_PCM_TXRX_CNTRL`.*
2. *If the line is currently performing Caller ID transmission, metering, or ringing, all such processes are stopped. No events are reported as a consequence of aborting these processes.*

RETURNS

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

EVENTS GENERATED

VP_LINE_EVID_LINE_INIT_CMP, on page 59

DEVICES

All

TERMINATIONS

All

6.2.4 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, 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. This function is useful for applying unique profiles to a particular line.

This function uses profiles in the same manner as *VpInitDevice()*, on page 66. See Chapter 2, on page 15 for a complete description of the profiles.

If the target line is an FXS termination, then this function assumes that the `pDcFeedOrFxoCfgProfile` argument points to a DC profile. Otherwise, this function assumes that the `pDcFeedOrFxoCfgProfile` argument points to a FXO configuration profile, and the `pRingProfile` argument is ignored. Any profile pointer argument equal to `VP_PTABLE_NULL` is ignored and the line's configuration is unchanged.

RETURNS

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

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

All

6.2.5 VpCalCodec()

SYNTAX

```
VpStatusType
VpCalCodec(
 VpLineCtxType *pLineCtx,          /* Pointer to line context */
 VpDeviceCalType mode)             /* Sets the calibration mode */
```

DESCRIPTION

This function calibrates analog circuits in the SLAC device that controls the specified line. The calibration procedure affects the entire SLAC device, so all lines driven by that SLAC device must be removed from service during the calibration procedure.

This procedure involves calibrating analog blocks, such as A/D converter offset voltages, and can take approximately 10 ms. The policy followed when performing a calibration is given by the `mode` argument, which can be one of the following enumerated values:

```
Enumeration Data Type: VpDeviceCalType:
    VP_DEV_CAL_NOW              /* Calibrate immediately */
    VP_DEV_CAL_NBUSY            /* Calibrate if all lines are "on-hook" */
```

If `VP_DEV_CAL_NOW` is specified, then the device is halted immediately, all lines forced to Standby and calibration started, regardless of the state of each line.

If `VP_DEV_CAL_NBUSY` is specified, then the device will be halted for calibration only if all channels of the device are inactive. If one or more of the lines on the specified device are in use (in a state other than `VP_LINE_DISCONNECT` or `VP_LINE_STANDBY`), then the event `VP_EVID_CAL_BUSY` is returned and no calibration is performed.

When calibration is done, each line of the calibrated SLAC is left in the `VP_LINE_STANDBY` state, and the `VP_EVID_CAL_CMP` event occurs. The line number returned with the `VP_EVID_CAL_CMP` event is the first (lowest) line number controlled by the calibrated SLAC device.

The lines serviced by the device being calibrated will be placed in the Standby state during calibration. If the lines were originally in the Disconnect state, they will not be transitioned to the Standby state using the ramp to Standby time specified by the `VP_DEVICE_OPTION_ID_RAMP2STBY` option; instead, the lines will be moved immediately into the Standby state. If a slow transition to Standby is desired, then each of the lines should be put into the Standby state before calling **`VpCalCodec()`**.

Notes:

1. *Calibration is automatically performed for a given device during the **`VpInitDevice()`** function. If **`VpCalCodec()`** is called directly, the host should re-run its system level calibration routine to recalculate the offsets that may be affected by the device's re-calibration. The host controller should also re-run the **`VpCalLine()`** function after executing the **`VpCalCodec()`** function.*
2. *The SLAC device cannot handle MPI activity while performing its internal calibration routine. Therefore, no command affecting the target SLAC device can be issued during calibration.*

RETURNS

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

EVENTS GENERATED

VP_EVID_CAL_CMP, on page 58
VP_EVID_CAL_BUSY, on page 58

DEVICES

CSLAC-790, VCP-790

TERMINATIONS

FXS

6.2.6 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

VpInitRing() initializes Ringing state parameters for the line associated with pLineCtx. These parameters determine the ringing cadence and Caller ID Profile to be used while the line is in the VP_LINE_RINGING state.

The Caller ID behavior and the Cadence employed during the Ringing state are defined by profiles. Like other profiles, the ringing 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 16 for more information.

The pCadProfile argument selects the profile to be used for the ringing cadence. If pCadProfile is VP_PTABLE_NULL, then the default "always on" ringing cadence is used.

The pCidProfile argument determines which Caller ID Profile is used during the ringing cadence. This function should be called with a Caller ID Profile when implementing Type-I Caller ID. If pCidProfile is a valid profile table index or profile pointer (not VP_PTABLE_NULL) and the ringing cadence includes Caller ID transmission, then Caller ID data is transmitted during subsequent ringing cadences on this line. Caller ID events (VP_LINE_EVID_CID_DATA) also occur during these ringing cadences. If pCidProfile equals VP_PTABLE_NULL, then Caller ID data is not automatically transmitted during subsequent ringing cadences on this line. The application can still manually transmit Caller ID using the **VpSendCid()** function. See *VpSendCid()*, on page 86 for details.

If automatic Type-I Caller ID is enabled for a line, the application should 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 72 for more information.

Each line controlled by the VP-API defaults to internal ringing with an "always on" cadence and no Caller ID, if otherwise not initialized. If the defaults are not acceptable, then this function should be called at least once for each line before putting the lines into Ringing mode. It is only necessary to initialize each line once, usually after a system reset, unless different ringing parameters are needed per call, such as a unique ringing cadence.

This function does not start ringing on the line. The application must call **VpSetLineState()** with VP_LINE_RINGING as the state argument in order to actually start ringing the line. The line will continue ringing, with the cadence defined by pCadProfile, until **VpSetLineState()** is called with a non-ringing state argument or an off-hook is detected.

Notes:

1. *Type-I Caller ID has the CLI (Caller Line Identity) frame transmitted as part of the caller alert sequence (ringing signal plus any other signals, such as Caller ID). This type of CLI is only transmitted while the line is on-hook. Different countries have different methods for performing Type-I Caller ID.*
2. *Type-II Caller ID is transmitted with the line off-hook, and provides caller line identity to lines with call waiting calls. Usually Type-II Caller ID requires a type of handshake with the CPE before FSK transmission begins.*

RETURNS

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

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

FXS

6.2.7 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 should be called before placing a line into the Ringing state if the associated line was set-up for Caller ID by **VpInitRing()**. If **VpInitCid()** is not called before the line is placed into Ringing, then the VTD will transmit a Caller ID message containing undefined data. Note that this function is necessary when implementing Caller ID Type-I.

The `length` argument should specify the total length in bytes of the entire message to be transmitted.

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

The VCP and CSLAC devices provide a 32-byte Caller ID message buffer, and this function returns an error if `length` is greater than 32. Since Caller ID messages can be longer than 32 bytes, the application may need to make several VP-API function calls to transmit a complete Caller ID message. To facilitate this, the VP-API generates the `VP_LINE_EVID_CID_DATA` event (with `eventData` equal to `VP_CID_DATA_NEED_MORE_DATA`) when 16 bytes of Caller ID data remain in the transmit buffer. It takes approximately 133 ms to send 16 bytes of CID 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 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.

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 Function Return Type*, on page 11

EVENTS GENERATED

VP_LINE_EVID_CID_DATA, on page 61

DEVICES

All

TERMINATIONS

FXS

6.2.8 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 16 for more information.

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

RETURNS

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

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

FXS

6.2.9 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 profile table. The VP-API implements *soft* profile tables for the CSLAC devices, wherein this function simply saves the profile pointer (`pProfile`) for later use. When subsequent VP-API function calls reference this profile by its index, the VP-API looks-up the saved pointer and reads the profile from application memory. See *Profile Tables*, on page 16 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 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 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 Function Return Type*, on page 11

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

All

6.2.10 VpSetBatteries()

SYNTAX

```
VpStatusType
VpSetBatteries(
    VpLineCtxType *pLineCtx,            /* Pointer to line context */
    VpBatteryModeType battMode,         /* Indicates to enable or disable use of
                                           programmed battery voltages */
    VpBatteryValuesType *pBatt)         /* Pointer to structure specifying battery
                                           voltages */
```

DESCRIPTION

This function enables or disables the use of the programmed battery voltages used by the device for dc feed purposes. This affects all lines on the device. It has potential improvment if the application battery voltages are stable to within a lesser percentage from what the device itself can measure.

The battMode parameter determines whether the programmed values should be used of the device internal battery sense used for dc feed computations. The range of battMode is:

```
Enumeration Data Type: VpBatteryModeType:
    VP_BATT_MODE_DIS              /* Use device measured batteries */
    VP_BATT_MODE_EN               /* Use programmed batteries */
```

When battMode is VP_BATT_MODE_EN, the structure passed in pBatt must be filled out as follows:

```
typedef struct VpBatteryValuesType {
    uint16 batt1;
    uint16 batt2;
    uint16 batt3;
};
```

Where batt1, batt2, and batt3 correspond to the battery configuration found in Table 5–3. Values for batt1, batt2, and batt3 are in 1.15 format ranging from +/-99.2V (3.027mV/step).

If battMode is VP_BATT_MODE_EN and pBatt is VP_NULL this function returns VP_STATUS_INVALID_ARG

When battMode is VP_BATT_MODE_DIS, the structure passed is ignored and may be VP_NULL.

RETURNS

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

EVENTS GENERATED

None

DEVICES

VCP

TERMINATIONS

All

CHAPTER

7 CONTROL FUNCTIONS

7.1 OVERVIEW

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

- **`VpSetLineState()`** – Sets a line to the requested state.
- **`VpSetLineTone()`** – Generates a cadenced call progress tone on a FXS line.
- **`VpSetRelayState()`** – Sets the line relay configuration.
- **`VpSetRelGain()`** – Sets the relative transmit or receive gain for a 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 for the purposes of implementing Type-II Caller ID.
- **`VpStartMeter()`** – Starts metering on a FXS line.
- **`VpSetOption()`** – Sets various device and line specific options.
- **`VpDeviceIoAccess()`** – Controls device input/output pins.
- **`VpVirtualISR()`** – Services CSLAC device interrupts.
- **`VpApiTick()`** – Called periodically to provide a timing basis for the VP-API.
- **`VpLowLevelCmd()`** – Allows the application to issue low level commands directly to the VTD. This function is an internal debugging tool that should not be used by the application.
- **`VpSetBFilter()`** – Enables with the coefficients provided or disables the B-Filter.

7.2 FUNCTION DESCRIPTIONS

7.2.1 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. The line state is typically set by the application in response to events that have occurred on the line (on-hook, off-hook) and commands from the back-end call control system (incoming call). All valid line states are listed below.

```
Enumeration Data Type: VpLineStateType:
    /* The following states are supported for FXS termination only */
    VP_LINE_STANDBY        /* On-hook, low power with normal polarity */
    VP_LINE_STANDBY_POLREV /* On-hook, low power mode with reverse polarity */
    VP_LINE_TIP_OPEN       /* Ground-start idle signaling state */
    VP_LINE_ACTIVE         /* Normal off-hook Active State; Voice Disabled */
    VP_LINE_ACTIVE_POLREV  /* Normal Active with reverse polarity; Voice Disabled */
    VP_LINE_TALK           /* Normal off-hook Active State; Voice Enabled */
    VP_LINE_TALK_POLREV    /* Normal Active with reverse polarity; Voice Enabled */
    VP_LINE_OHT            /* On-hook transmission state */
    VP_LINE_OHT_POLREV     /* On-hook transmission state with reverse polarity */
    VP_LINE_DISCONNECT     /* Line out of service State*/
    VP_LINE_RINGING        /* Place Ringing on the Line */
    VP_LINE_RINGING_POLREV /* Place Ringing on the Line with reverse polarity */

    /* The following states are supported for FXO termination only */
    VP_LINE_FXO_OHT        /* FXO Line providing Loop Open with voice feed */
    VP_LINE_FXO_LOOP_OPEN  /* FXO Line providing Loop Open without voice feed */
    VP_LINE_FXO_LOOP_CLOSE /* FXO Line providing Loop Close without voice feed */
    VP_LINE_FXO_TALK       /* FXO Line providing Loop Close with voice feed */
    VP_LINE_FXO_RING_GND   /* FXO Line providing Ring Ground (ground-start only) */
```

In the `VP_LINE_RINGING` state, the VP-API may perform ringing cadencing and/or Caller ID transmission, according to the profiles specified in the last call to **`VpInitRing()`**. Ringer may be applied to and removed from the line synchronized with the zero-crossing of the ringing signal. The operation of the ringing entry and exit can be modified using the **`VpSetOption()`** function. If a relay is supported by the termination type, it must be set to `VP_RELAY_NORMAL` to allow the VTD to control the ringing relay when using external ringing. See *VP_OPTION_ID_RING_CNTRL*, on page 42 and *VP_OPTION_ID_ZERO_CROSS*, on page 37 for more details.

The states `VP_LINE_ACTIVE` and `VP_LINE_ACTIVE_POLREV` place the line in normal off-hook state. However, voice data exchange through the PCM interface is disabled in this state.

The states `VP_LINE_TALK` and `VP_LINE_TALK_POLREV` place the line in normal off-hook state with voice data exchange enabled.

The on-hook transmission states (`VP_LINE_OHT` and `VP_LINE_OHT_POLREV`) enable the VTD voice path so that Caller ID data can be transmitted by the VTD through its PCM or packet interface.

If the line is configured as a FXO termination, only the following states are valid: `VP_LINE_FXO_OHT`, `VP_LINE_FXO_LOOP_OPEN`, `VP_LINE_FXO_LOOP_CLOSE`, `VP_LINE_FXO_TALK` and `VP_LINE_FXO_RING_GND`.

The `VP_LINE_FXO_LOOP_OPEN` and `VP_LINE_FXO_LOOP_CLOSE` states put the line in loop open (on-hook) and loop closed (off-hook) state respectively. Voice data exchange through the PCM interface is disabled in `VP_LINE_FXO_LOOP_CLOSE` state. The `VP_LINE_FXO_TALK` state is similar to the `VP_LINE_FXO_LOOP_CLOSE` state except that the voice path is enabled in the `VP_LINE_FXO_TALK` state.

The `VP_LINE_FXO_OHT` state puts the line in the loop open state with the ability to receive on-hook signals such as Caller ID

The `VP_LINE_FXO_RING_GND` state is used to apply a ground to the Ring lead as used in ground-start signaling. This line state is not currently supported by a Zarlink Semiconductor FXO Termination type.

Notes:

1. ***`VpInitDevice()`*** *places all FXS lines in the* `VP_LINE_DISCONNECT` *state, which effectively disconnects the lines from the loop. The application must call this function after initialization to enable service to the customer.*
2. *The* `VP_LINE_STANDBY_POLREV` *line state is supported for the CSLAC-880 and CSLAC-890 devices only.*
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. Therefore, setting a line state will immediately stop a currently running Caller ID cadence, Ringing Cadence, or other line state control from* ***`VpSendSignal()`****. Setting a line to a state does not affect Tone Cadencing (except possibly the analog performance by changing the line state). If metering is being generated, the current meter pulse will continue until complete, then will terminate without generating the remaining meter pulses (if any). If the pulse is defined as "always on", it will terminate immediately.*

RETURNS	See *VP-API Function Return Type*, on page 11
EVENTS GENERATED	None.
DEVICES	All
TERMINATIONS	All

7.2.2 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 downstream direction */
    VP_DIRECTION_US             /* Tone generation in upstream direction */

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

Notes:

1. *Call progress tone and DTMF tone generation cannot be performed simultaneously.*
2. *Only VPP devices support DTMF tone generation in both the upstream and downstream directions. CSLAC and VCP devices support DTMF tone generation only in the downstream direction.*
3. *The combination of any valid tone profile (except for* `VP_PTABLE_NULL`*) with a special cadence profile is used to generate a UK or Australian Howler tone.*
4. *VCP-VP790 does not support DTMF Tone generation using* `pDtmfControl`*.*
5. *The event is generated for CSLAC and VCP devices only.*

RETURNS

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

EVENTS GENERATED

VP_LINE_EVID_TONE_CAD, on page 62

DEVICES

All

TERMINATIONS

All

7.2.3 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 24 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 relay(s), 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 141 describes simple connection diagrams for applicable relay states for various line termination types.

RETURNS

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

EVENTS GENERATED

None

DEVICES

CSLAC, VCP-790, VPP

TERMINATIONS

All

7.2.4 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 made relative to the gain levels set in the AC Profile applied to the line. Setting the `txLevel` or `rxLevel` to 1.0 resets the respective path to the default gain from the AC Profile.

The transmit and receive gains are specified as 2.14 fixed-point unsigned numbers with a range of 0 to 4.0 (actually 3.9999) of absolute gain adjustment. 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 5.5.7 with the definition of this event.

RETURNS

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

EVENTS GENERATED

VP_LINE_EVID_GAIN_CMP, on page 59

DEVICES

VCP, CSLAC-880, CSLAC-890

TERMINATIONS

All

7.2.5 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 80 for the definition of **VpDigitType**.

There is no data associated with a hook flash signal (VP_SENDSIG_HOOK_FLASH), so pSignalData is ignored in that case.

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. Depending on the state of the line prior to entering Tip Open, ground key and/or hook events may occur while performing Tip Open Pulse Send Signal and after the 100ms recovery time.

When sending a Momentary Loop Open, pSignalData is ignored. The FXO line will apply a loop open for >=10ms and measure the T/R voltage at the end of the loop open interval. 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. Other signals generated will continue to completion.

The VP_LINE_EVID_SIGNAL_CMP event occurs when signal generation is done.

Notes:

The VP_SENDSIG_FWD_DISC, VP_SENDSIG_POLREV_PULSE, *and* VP_SENDSIG_MOMENTARY_LOOP_OPEN *signal types are supported for the CSLAC-880 and CSLAC-890 devices only.*

The VP_SENDSIG_TIP_OPEN *signal type is supported for the CSLAC-880 devices only.*

RETURNS	See *VP-API Function Return Type*, on page 11
EVENTS GENERATED	*VP_LINE_EVID_SIGNAL_CMP*, on page 62
DEVICES	CSLAC-880, CSLAC-890, VCP-880
TERMINATIONS	All

7.2.6 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 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 72 for move information on handling the Caller ID data.

RETURNS

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

EVENTS GENERATED

VP_LINE_EVID_CID_DATA, on page 61

DEVICES

All

TERMINATIONS

FXS

7.2.7 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 writes more Caller ID data to the VTD Caller ID data buffer. This function should be called each time the `VP_LINE_EVID_CID_DATA:VP_CID_DATA_NEED_MORE_DATA` event occurs and there is additional Caller ID data to transmit on the relevant line. If **`VpContinueCid()`** is not called in response to this event, then the VTD transmits the remaining message data followed by the checksum with the previous buffer contents. This function is necessary for both Type-I and Type-II Caller ID implementations.

The `pCidData` argument should point to a buffer containing the next segment of Caller ID data, up to 16 bytes in length. Data blocks longer than 16 bytes are not supported.

The `VP_LINE_EVID_CID_DATA:VP_CID_DATA_TX_DONE` event occurs once all of the Caller ID data is transmitted.

RETURNS

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

EVENTS GENERATED

VP_LINE_EVID_CID_DATA, on page 61

DEVICES

CSLAC, VCP

TERMINATIONS

FXS

7.2.8 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 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 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. When the VP-API 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 does not need the services of the DTMF decoder anymore. See *VpSysDtmfDetEnable(), VpSysDtmfDetDisable()*, on page 119 for further information about these functions.

During the Detect Interval, **VpDtmfDigitDetected()** should be called to notify the VP-API 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 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 80.

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 even though the VP-API 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 Function Return Type*, on page 11

EVENTS GENERATED

VP_LINE_EVID_DTMF_DIG, on page 55

DEVICES

CSLAC

TERMINATIONS

FXS

7.2.9 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 73. 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 Function Return Type*, on page 11

EVENTS GENERATED

VP_LINE_EVID_MTR_CMP, on page 61
VP_LINE_EVID_MTR_ABORT, on page 61

DEVICES

All

TERMINATIONS

FXS

7.2.10 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 4, on page 33 for a complete list and definition of all VP-API options.

This function acts on one or more lines 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 7–1 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 4, on page 33 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 66). 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 Function Return Type*, on page 11

EVENTS GENERATED

None.

DEVICES

All

TERMINATIONS

All

7.2.11 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, on page 43* 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. During an I/O read operation, the state of any pin with its accessMask bit set to 0 is reported as 0. 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 following enumeration type can be used to set accessMask bits:

```
Enumeration Data Type: VpDeviceIoAccessMask:
    VP_DEVICE_IO_IGNORE             /* Ignore I/O access */
    VP_DEVICE_IO_ACCESS             /* Perform I/O access */
```

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.

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 even occurs. Refer to Section 5.5.10 for details.

Notes:

It is the user's responsibility to determine how the I/O pins of the VTD (or SLAC devices in the case of a VCP) are used in their system. Users must take care not to change the state of any I/O pins that are reserved by the reference design to control relays or LCAS devices. Refer to VP_DEVICE_OPTION_ID_DEVICE_IO, on page 43 for more information on I/O pin configuration and restrictions.

RETURNS

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

EVENTS GENERATED

VP_DEV_EVID_IO_ACCESS_CMP, on page 60

DEVICES

All

TERMINATIONS

All

7.2.12 VpVirtualISR()

SYNTAX

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

DESCRIPTION

This function is used by the VP-API when the level-triggered or edge-triggered interrupt modes are used (see Chapter 11, on page 125). 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 that an interrupt has occurred and must be serviced.

RETURNS

None

EVENTS GENERATED

None

DEVICES

CSLAC

TERMINATIONS

All

7.2.13 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 Time Base*, on page 125 for more information. Any timing needed by the VP-API is derived from this periodic function call. This function writes `TRUE` to the location pointed to by `pEventStatus` if a VP-API 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 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 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 tick and the tick rate.

RETURNS

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

EVENTS GENERATED

None

DEVICES

CSLAC

TERMINATIONS

All

7.2.14 VpLowLevelCmd()

SYNTAX

```
VpStatusType
VpLowLevelCmd(
 VpLineCtxType *pLineCtx,          /* Pointer to line context */
 uint8 *pCmdData,                  /* Pointer to command and data */
 uint8 len,                        /* The length of the cmd/data string */
 uint16 handle)                    /* Handle value returned with event */
```

DESCRIPTION

This function is not recommended for use by the customer. Improper use of this function could break the synchronization between VP-API and the VTD, resulting in unpredictable behavior by the VP-API. The VP-API will not prevent the user from improperly using this function. This function is implemented to enable debugging of the VP-API.

This function circumvents the VP-API and allows direct VTD access. It should be used for debugging purposes only. The VP-API maintains copies of some VTD registers and information about the VTD current state, so this function must be used with caution.

The set of low-level commands available through this function depends on which Zarlink Semiconductor devices are used in the design. Refer to the appropriate *Chip Set User's Guide* for a description of the low-level commands. This function can issue both read and write commands.

The `pLineCtx` argument identifies the device/channel to which the command is issued. The `pCmdData` arguments points to a command/data string whose format and content is device-dependent. The `len` argument specifies the length of the command/data string in bytes *minus one*. Finally, the `handle` argument determines the value of the handle attached to the events generated by this function.

For write operations, the `VP_LINE_EVID_LLCMD_TX_CMP` event occurs when the low-level write command is done. For read operations, the `VP_LINE_EVID_LLCMD_RX_CMP` event occurs when the low-level read command is done. Upon receiving the `VP_LINE_EVID_LLCMD_RX_CMP` event, the **`VpGetResults()`** function should be called to place the results into a buffer of `len` size.

RETURNS

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

EVENTS GENERATED

VP_LINE_EVID_LLCMD_TX_CMP, on page 56
VP_LINE_EVID_LLCMD_RX_CMP, on page 56

DEVICES

All

TERMINATIONS

All

7.2.15 VpSetBFilter()

SYNTAX

```
VpStatusType
VpSetLineState(
 VpLineCtxType *pLineCtx,              /* Pointer to line context */
 VpBFilterModeType bFiltMode,          /* Selects the desired B-Filter Mode
                                          (enable or disable) */
 VpProfilePtrType pAcProfile)          /* Pointer to AC profile containing
                                          desired B-Filter values to program.
                                          Used if B-Filter being enabled */
```

DESCRIPTION

This function enables or disables the B-Filter on the ilne associated with `pLineCtx`. The valid settings for `bFiltMode` are listed below.

```
Enumeration Data Type: VpBFilterModeType:
    /* The following states are supported for FXS termination only */
    VP_BFILT_DIS          /* Disable the B-Filter */
    VP_BFILT_EN           /* Enable the B-Filter */
```

If `VP_BFILT_EN` is passed, then values provided in `pAcProfile` are loaded into the device. If `VP_BFILT_EN` is passed and `pAcProfile` is `VP_PTABLE_NULL`, this function returns `VP_STATUS_INVALID_ARG`. If `VP_BFILT_DIS` is passed, the B-Filter is disabled.

RETURNS

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

EVENTS GENERATED

None.

DEVICES

All

TERMINATIONS

All

7.2.16 VpLineIoAccess()

SYNTAX

```
VpStatusType
VpLineIoAccess(
 VpLineCtxType *pLineCtx,                    /* Pointer to line context */
 VpLineIoAccessType *pLineIoAccess,          /* Struct containing access type
                                                and values to be written */
 uint16 handle)                              /* Handle value returned with
                                                event */
```

DESCRIPTION

This function accesses some or all of the general-purpose input/output (GPIO) pins associated with a particular line. Refer to *VP_OPTION_ID_LINE_IO_CFG*, on page 45 for information on I/O pin configuration and restrictions. This function takes a pointer to the following structure type:

```
typedef struct {
    VpIoDirectionType direction;
    VpLineIoBitsType ioBits;
} VpLineIoAccessType;
```

The `direction` field determines whether a read or write operation is performed on the I/O pins. It can take on the following values:

```
typedef enum {
    VP_IO_WRITE,
    VP_IO_READ,
} VpIoDirectionType;
```

The `ioBits` field is a `VpLineIoBitsType` struct:

```
typedef struct {
    uint8 mask;
    uint8 data;
} VpLineIoBitsType;
```

The `mask` field contains a bit for each GPIO pin associated with the line. For each bit in this field, if the bit is set, then the corresponding GPIO pin is accessed; if the bit is 0, then the corresponding GPIO pin is left alone.

The `data` field also contains a bit for each GPIO pin associated with the line. For write operations (`VP_IO_WRITE`), the GPIO pins are set to the values specified in this field only if the corresponding bit in the `mask` field is set. For read operations (`VP_IO_READ`), the values of the GPIO pins are returned with the `VP_LINE_EVID_IO_RD_CMP` event onlly if the corresponding bit in the `mask` field is set (otherwise 0 is returned).

For write operations, the `VP_LINE_EVID_LINE_IO_WR_CMP` event occurs when the GPIO write command is done. For read operations, the `VP_LINE_EVID_LINE_IO_RD_CMP` event occurs when the GPIO read command is done. Upon receiving the `VP_LINE_EVID_LINE_IO_RD_CMP` event, the **`VpGetResults()`** function should be called to place the results into a `VpLineIoAccessType` buffer.

Notes:
It is the user's responsibility to determine how the I/O pins of the VTD (or SLAC devices in the case of a VCP) are used in their system. Users must take care not to change the state of any I/O pins that are reserved by the reference design to control relays or LCAS devices. Refer to *VP_OPTION_ID_LINE_IO_CFG*, on page 45 *for more information on I/O pin configuration and restrictions.*

RETURNS

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

EVENTS GENERATED

VP_LINE_EVID_LINE_IO_RD_CMP, on page 60
VP_LINE_EVID_LINE_IO_WR_CMP, on page 60

DEVICES

VCP2

TERMINATIONS

All

7.2.17 VpDeviceIoAccessExt()

SYNTAX

```
VpStatusType
VpDeviceIoAccessExt(
 VpDevCtxType *pDevCtx,                          /* Pointer to device context */
 VpDeviceIoAccessExtType *pDeviceIoAccess)       /* Struct containing access
                                                 type and values to be written
                                                 */
```

DESCRIPTION

This function accesses some or all of the general-purpose input/output (GPIO) pins associated with a device in a single operation. This function is now preferred instead of the deprecated **VpDeviceIoAccess()** function. Refer to *VP_DEVICE_OPTION_ID_DEV_IO_CFG*, on page 44 and *VP_OPTION_ID_LINE_IO_CFG*, on page 45 for information on I/O pin configuration and restrictions. This function takes a pointer to the following structure type:

```
typedef struct {
    VpIoDirectionType direction;
    VpLineIoBitsType lineIoBits[VP_MAX_LINES_PER_DEVICE];
} VpDeviceIoAccessExtType;
```

The `direction` field determines whether a read or write operation is performed on the I/O pins. It can take on the following values:

```
typedef enum {
    VP_IO_WRITE,
    VP_IO_READ,
} VpIoDirectionType;
```

The `lineIoBits` array contains a `VpLineIoBitsType` element for each line controlled by the device. See *VpLineIoAccess()*, on page 96 for a description of this struct containing line-specific GPIO access information. The number of array elements is `VP_MAX_LINES_PER_DEVICE`, a compile-time option specified in vp_api_cfg.h.

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 even occurs. Refer to Section 5.5.10 for details.

RETURNS

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

EVENTS GENERATED

VP_DEV_EVID_IO_ACCESS_CMP, on page 60

DEVICES

VCP2

TERMINATIONS

All

CHAPTER

8 STATUS AND QUERY FUNCTIONS

8.1 OVERVIEW

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

- **`VpGetEvent()`** – Returns events corresponding to a device.
- **`VpGetLineStatus()`** – Returns the state of a particular status flag for one line.
- **`VpGetDeviceStatus()`** – Returns the state of a particular status flag for up to 32 lines.
- **`VpGetLoopCond()`** – Reads a snapshot of loop conditions for an FXS line and returns parameters such as voltage, current, and resistance.
- **`VpGetOption()`** – Returns the current setting of an option.
- **`VpGetLineState()`**– Reads the current line state.
- **`VpFlushEvents()`** – Flushes all outstanding events.
- **`VpGetResults()`** – Reads the data associated with an event.
- **`VpClearResults()`** – Discards the data associated with an event.

8.2 FUNCTION DESCRIPTIONS

8.2.1 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 should be called if the **VpApiTick()** function sets its event flag to TRUE, indicating that an event has occurred. See *Handling Interrupts from CSLAC Devices*, on page 125 for more information.

The pDevCtx argument must point to the context of the VTD that is 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 equal to VP_STATUS_SUCCESS and **VpGetEvent()** returned TRUE
  An event was retrieved from the VTD, event data valid.
- status equal to VP_STATUS_SUCCESS and **VpGetEvent()** returned FALSE
  No event was retrieved from the VTD, ignore event data.
- status not equal to VP_STATUS_SUCCESS
  VpGetEvent() encountered an error that may need debugging. Ignore event data and function return value. See *VP-API 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 always 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 */
    VP_EVCAT_PACKET             /* Packet event category */
```

The individual event ID is passed through the eventId member of the event structure. Refer to Chapter 5, on page 47 for a complete list of the individual event ID names. Note that the event ID constants are only unique within the applicable event category.

The parmHandle and eventData variables contain additional event-specific information. The boolean hasResults variable indicates whether additional data related to the event is present in the mailbox. The application must either retrieve the additional data using **VpGetResults()** or de-queue the data by calling **VpClearResults()**. Chapter 5, on page 47 describes the parmHandle, eventData, and extended results for each VP-API 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

8.2.2 VpGetLineStatus()

SYNTAX

```
VpStatusType
VpGetLineStatus(
 VpLineCtxType *pLineCtx,        /* Pointer to line context */
 VpInputType input               /* Test the status of this input type */
 bool *pStatus)                  /* 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 */
    VP_INPUT_HOOK                /* Hook Status (ignoring pulse & flash) */
    VP_INPUT_RAW_HOOK            /* Hook Status (include pulse & flash) */
    VP_INPUT_GKEY                /* Ground-Key/Fault Status */
    VP_INPUT_THERM_FLT           /* Thermal Fault Status */
    VP_INPUT_CLK_FLT             /* Clock Fault Status */
    VP_INPUT_AC_FLT              /* AC Fault Status */
    VP_INPUT_DC_FLT              /* DC 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 */

    /* FXO Status types */
    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 */
```

The boolean status result, pointed to by `pStatus`, is interpreted differently depending on the type of input queried:

- Fault flags are either active (`TRUE`) or inactive (`FALSE`).
- Hook status is either off-hook (`TRUE`) or on-hook (`FALSE`). When automatic pulse-digit decoding is disabled, the values for `VP_INPUT_HOOK` and `VP_INPUT_RAW_HOOK` are identical. When pulse-digit decoding is enabled, `VP_INPUT_RAW_HOOK` reflects the current state of the line's hook detector, while `VP_INPUT_HOOK` represents the logical hook state after filtering pulse-digit and hook-switch flash events.
- Ground key status is either active (`TRUE`) or inactive (`FALSE`).
- FXO line conditions are simply either `TRUE` or `FALSE` (e.g., line is in a reversed polarity if `VP_INPUT_POLREV is TRUE`).

This function returns the `VP_STATUS_INVALID_ARG` error if the application requests status information for the wrong type of line termination (e.g. requesting FXS status for an FXO termination).

Notes:

1. *This function returns the status of one input for a single line.* ***`VpGetDeviceStatusExt()`*** *returns an array of status flags for all lines controlled by a device. See* *VpGetDeviceStatusExt(), on page 109 for further information.*
2. *An active ground-key may be considered a fault by the application when the line is not used in a ground-start system. Ground-key and DC fault are polarity sensitive and although they both monitor longitudinal currents, either one may appear as an indication of a line fault depending on the polarity of the fault current.*
3. *See Table 5–3 to convert generic battery names (bat1, bat2, etc.) to device-specific battery names (VBH, VBL, etc.).*

RETURNS

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

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

All

8.2.3 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 supports (up to 32 lines). This function is now deprecated in favor of **VpGetDeviceStatusExt()** (see *VpGetDeviceStatusExt()*, on page 109), which can return status information for more than 32 lines.

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 102 for the type definition of the `input` argument and for information on decoding the status flags.

Notes:

1. *If a device supports both FXS and FXO terminations, the returned status information is only valid for lines whose termination type matches the requested input type. For example, if the application requests the status of an FXO input, the result bits for all FXS lines should be ignored.*
2. *An active ground-key may be considered a fault by the application when the line is not employed in a ground-start system.*
3. *This function returns the status for all lines multiplexed into one 32-bit result.* ***`VpGetLineStatus()`*** *returns a boolean result for one specific line. See* *VpGetLineStatus()*, on page 102 *for further information.*
4. *If the device has more than 32 channels, the 32-bit result contains the results for the first 32 channels on the device. If status information is needed about other channels, see* *VpGetDeviceStatusExt()*, on page 109.

RETURNS

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

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

All

8.2.4 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 57 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.*

RETURNS

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

EVENTS GENERATED

VP_LINE_EVID_RD_LOOP, on page 57

DEVICES

CSLAC-790, VCP, VPP

TERMINATIONS

FXS

8.2.5 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 options see Chapter 4.

`VpGetOption()` actually starts a process in the VP-API/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 8–1 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 56 for more information on retrieving the option data associated with this event.

RETURNS

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

EVENTS GENERATED

VP_LINE_EVID_RD_OPTION, on page 56

DEVICES

All

TERMINATIONS

All

8.2.6 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`. *VpSetLineState()*, on page 78 describes the line states.

RETURNS

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

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

All

8.2.7 VpFlushEvents()

SYNTAX

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

DESCRIPTION

This function empties the VP-API event queue. This function does not clear the VP-API results buffer; it only clears the pending event queue.

RETURNS

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

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

All

8.2.8 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 an event. Recall from Chapter 5 that events with attached results have their `hasResults` members set to `TRUE`. The application must either read the results using **`VpGetResults()`** or discard the results by calling **`VpClearResults()`** (see *VpClearResults()*, on page 109).

To read 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 application's buffer and frees the VP-API result buffer. The function uses the `pEvent` argument to determine what type of results are being copied. The application should simply pass a pointer to the same event returned by **`VpGetEvent()`**. Table 8–2 lists all VP-API functions that generate results along with the corresponding event ID and results data type.

Table 8–2 VP-API Functions with Extended Results

Function	Event ID	Result Type
`VpLowLevelCmd()`	`VP_LINE_EVID_LLCMD_RX_CMP`	**`uint8p`**
`VpGetLoopCond()`	`VP_LINE_EVID_RD_LOOP`	**`VpLoopCondResultsType`**
`VpGetOption()`	`VP_LINE_EVID_RD_OPTION`	See Section 8.2.5.
`VpDeviceIoAccess()`	`VP_DEV_EVID_IO_ACCESS_CMP`	**`VpDeviceIoAccessDataType`**

Notes:

1. *The application can also use this function to determine the type of result data waiting in the result buffer. To use this mode of operation, this function should be called with the* `pResults` *argument equal to* `VP_NULL`*. When* ***`VpGetResults()`*** *is called in this mode, it overwrites the* `eventCategory` *and* `eventId` *members of the event structure passed to this function. Note that the event passed to this function must contain a valid* `deviceId` *and device context pointer (*`pDevCtx`*). If there are no results waiting in the buffer, the* `eventId` *member in event structure (*`pEvent`*) is overwritten with all zeros.*
2. *When reading options using* ***`VpGetOption()`****, the application can use the* `eventData` *member of the event structure (****`VpEventType`****) to determine the option type (****`VpOptionIdType`****) that was read.*

RETURNS

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

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

All

8.2.9 VpClearResults()

SYNTAX

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

DESCRIPTION

This function clears the VP-API results buffer, thereby making room for more results. The application can call this function instead of **VpGetResults()** if it does not care about the results associated with an event.

The VP-API results queue provides access to only one result. In other words, calling this function deletes only the top results queue entry. This function can be called even when there are no outstanding results in the queue, in which case it simply returns VP_STATUS_SUCCESS.

RETURNS

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

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

All

8.2.10 VpGetDeviceStatusExt()

SYNTAX

```
VpDevCtxType *pDevCtx, VpDeviceStatusType
*pDeviceStatus
VpGetDeviceStatusExt(
  VpDevCtxType *pDevCtx,                    /* Pointer to device context */
  VpDeviceStatusType *pDeviceStatus)        /* Struct containing
                                               parameters and results. */
```

DESCRIPTION

This function is an extended version of **VpGetDeviceStatus()** (see page 103) with support for devices with more than 32 channels. The two functions differ only in the format of the results struct. This function accepts a pointer to a VpDeviceStatusType struct:

```
typedef struct {
    VpInputType input;
    uint8 status[VP_LINE_FLAG_BYTES];
} VpDeviceStatusType;
```

It returns the status of the requested input for all lines controlled by the device. Each bit in the status field represents the status of input for one line.

The number of array elements, VP_LINE_FLAG_BYTES = (VP_MAX_LINES_PER_DEVICE + 7) / 8, is the number of eight-bit integers required to store a flag for each channel in the device. VP_MAX_LINES_PER_DEVICE is a compile-time option defined in vp_api_cfg.h. Bit 0 in array element 0 represents line 0, bit 1 represents line 1, and so on. Bit 0 in array element 1 represents line 8.

Refer to *VpGetLineStatus()*, on page 102 for the type definition of the input argument and for information on decoding the status flags.

Notes:

1. *If a device supports both FXS and FXO terminations, the returned status information is only valid for lines whose termination type matches the requested input type. For example, if the application requests the status of an FXO input, the result bits for all FXS lines should be ignored.*
2. *An active ground-key may be considered a fault by the application when the line is not employed in a ground-start system.*
3. *This function returns the status for all lines multiplexed into an array of eight-bit results.* ***VpGetLineStatus()*** *returns a boolean result for one specific line. See* *VpGetLineStatus()*, on page 102 *for further information.*

RETURNS

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

EVENTS GENERATED	None.
DEVICES	VCP2
TERMINATIONS	All

CHAPTER

9 SYSTEM SERVICES

9.1 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 VoicePath API is used. The following functions are included in the System Services layer.

- **`VpSysEnterCritical()`** – Blocks entry into a critical section of VP-API code through some user-defined method.
- **`VpSysExitCritical()`** – Marks the end of a VP-API critical code section.
- **`VpSysWait()`** – Implements a software delay that is only used during CSLAC device initialization.
- **`VpSysDisableInt()`** – Required by the Level-Triggered interrupt mode to disable the CSLAC device interrupt in the host microprocessor.
- **`VpSysEnableInt()`** – Required by the Level-Triggered interrupt mode to 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()`**, **`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.

9.2 VP-API REENTRENCY

The term “reentrant” is defined as:

> *A computer program or routine is described as reentrant if it is designed such that a single copy of the program's instructions in memory can be shared by multiple users or separate processes. The key to the design of a reentrant program is to ensure that no portion of the program code is modified by the different users/processes, and that process-unique information (such as local variables) is kept in a separate area of memory that is distinct for each user or process. Reentrant programming is key to many systems of multitasking.*
>
> *http://en.wikipedia.org/wiki/Reentrant*

The VP-API supports reentrant application development. Reentrency does not mean that a function always completes its intended action when called more than once simultaneously (reentered). Reentrency means that a function can detect whether it has been reentered and either completes its intended action or returns an error notifying the caller that the function was not successful. In either case the behavior of the reentrant function must be consistent and well-defined.

The VP-API follows this strategy. VP-API functions return an error code (`VP_STATUS_IN_CRTCL_SECTN`) when they are reentered but are unable to perform the desired function because more than one thread needs access to a shared resource. There are a few different types of shared resources in the VoicePath system that must be protected from simultaneous access: device objects, line objects, and device resources. Recall from Section 3.2 that device objects and line objects are data structures that retain state information for devices and lines respectively. VP-API functions that modify these objects are protected from reentrant execution. Device resources are physical components or features of a device. Some device resources, such as the HBI or MPI, must also be protected from simultaneous access. The VP-API reentrancy protection scheme behaves differently depending on the type of shared resource being protected. Table 9–1 summarizes this behavior.

Table 9–1 VP-API Reentrency Behavior

Objects Accessed by First API Func. Call	Objects Accessed by Reentrant API Call(s)	Reentrency Behavior
Line-Specific VP-API Functions		
Line: X Device: X	Line: X Device: X	Executing any two line-specific VP-API functions simultaneously on the same line of the same device returns a critical section error for the reentrant function call(s).
Line: X Device: X	Line: Any other than X Device: X	Executing any two line-specific VP-API functions simultaneously on different lines of the same device may return a critical section error, depending on whether any shared device resources are accessed by the functions. Also, if a line-specific function is using a device resource, then calling a device-specific function on the same device results in a critical section error because the device resource is unavailable.
Line: Any Device: X	Line: Any Device: Any other than X	All VP-API functions complete successfully when the devices they access are different.
Device-Specific VP-API Functions		
Line: Any Device: X	Line: Any Device: X	Executing any two device-specific VP-API functions simultaneously on the same device results in a critical section error for the reentrant function call(s).

9.3 FUNCTION DESCRIPTIONS

9.3.1 VpSysEnterCritical()

SYNTAX

```
uint8
VpSysEnterCritical(
 VpDeviceIdType deviceId,                /* Selects the target device */
 VpCriticalSecType criticalSecType)      /* Indicates critical section type
                                            */
```

DESCRIPTION

This function protects critical sections of VP-API code from reentrant execution. The application developer must implement this function such that, for a given device, no VP-API 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_HBI_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, HBI/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. 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 calls are made from only one thread of execution, then this function can simply return 1. That is, no critical section protection is actually required in this case.

RETURNS The nesting level after the call is completed.

EVENTS GENERATED None

DEVICES All

TERMINATIONS All

9.3.2 VpSysExitCritical()

SYNTAX

```
uint8
VpSysExitCritical(
 VpDeviceIdType deviceId,              /* Selects the target device */
 VpCriticalSecType criticalSecType)    /* Indicates critical section type */
```

DESCRIPTION

The VP-API calls this function at the end of a critical code section. This function should restore the critical section protection (interrupt, semaphore, etc.) state to its condition prior to the last **`VpSysEnterCritical()`** call. This is typically done by enabling appropriate interrupts or "giving" a task-blocking semaphore. See *VpSysEnterCritical()*, on page 113 for descriptions of the `deviceId` and `criticalSecType` input parameters.

This function should be implemented such that nesting of any critical section beneath a code critical section is allowed. **`VpSysExitCritical()`** should return the current critical section nesting level.

RETURNS

The nesting level after the call is completed.

EVENTS GENERATED

None

DEVICES

All

TERMINATIONS

All

9.3.3 VpSysWait()

SYNTAX

```
void
VpSysWait(
 uint8 time)                          /* Number of frame syncs to wait */
```

DESCRIPTION

The VP-API calls this function when it needs to wait certain amount of time. The `time` argument determines the wait time in terms of 125 µs frame sync periods. The VP-API uses this function during CSLAC initialization, before starting real-time operation.

RETURNS

None

EVENTS GENERATED

None

DEVICES

CSLAC

TERMINATIONS

All

9.3.4 VpSysDisableInt()

SYNTAX

```
void
VpSysDisableInt(
  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 128. This function is called by the VP-API 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`. This function can be an "empty" function in systems that use edge-triggered interrupts or poll for CSLAC interrupts.

RETURNS

None

EVENTS GENERATED

None

DEVICES

CSLAC

TERMINATIONS

All

9.3.5 VpSysEnableInt()

SYNTAX

```
void
VpSysEnableInt(
  VpDeviceIdType deviceId)          /* Selects the target device */
```

DESCRIPTION

This System Services function must be implemented by the customer if the design is based on CSLAC family of devices and uses interrupt architecture as defined in *Mode 2: Interrupt Driven, Level-Triggered*, on page 128. This function is called by the VP-API just before returning from **VpApiTick()**. The CSLAC device for which interrupt needs to be enabled is indicated by `deviceId`. This function can be an "empty" function in systems that use edge-triggered interrupt mode or interrupt polling mode.

RETURNS

None

EVENTS GENERATED

None

DEVICES

CSLAC

TERMINATIONS

All

9.3.6 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 is being used for CSLAC family of devices with an implementation architecture described in *Mode 3: Efficient Polled Mode*, on page 129. 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

9.3.7 VpSysDtmfDetEnable(), 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 88* for further information on this function.

RETURNS

None

EVENTS GENERATED

None

DEVICES

CSLAC

TERMINATIONS

FXS

CHAPTER

10 HARDWARE ABSTRACTION LAYER

10.1 OVERVIEW

The Hardware Abstraction Layer (HAL) defines functions for communicating with a target VTD through the MPI or HBI. These functions hide the details of the platform MPI/HBI hardware design from the VP-API. The customer must implement these functions as appropriate for their specific platform. Zarlink Semiconductor provides example implementations of these functions. The following functions are included in the HAL:

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

All HAL functions take a `deviceId` argument that identifies the target VTD. This argument is of type **`VpDeviceIdType`**, which is defined by the customer. The `deviceId` is part of the device object created by the application at initialization. Note that the VP-API does not access the `deviceId`; it merely passes the `deviceId` down to the HAL when accessing the associated VTD. This feature is useful for addressing a specific VTD in systems where a single host microprocessor controls more than one VTD. This parameter may be ignored in designs with only one VTD per host microprocessor.

MPI transactions are atomic operations in that once once starts on a particular device, it must complete before another transaction can start for the same device. The VP-API protects all MPI transactions with **`VpSysEnterCritical()`** and **`VpSysEnterCritical()`** to guarantee that they run without interruption on the same device from another source. For more information please see Chapter 9, on page 111.

10.2 FUNCTION DESCRIPTIONS

10.2.1 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. The `cmd` argument contains the first MPI command in the block (note: the total data passed to VpMpiCmd() may be a series of MPI cmd+data, not necessarily just a single MPI cmd+data set). Bit zero of this command 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 (max 255) following the command byte from the location pointed by the `dataPtr` argument to the CSLAC device. If the command byte indicates a read operation, this function reads and stores `cmdLen` number of bytes (max 255) from the CSLAC device to the location pointed by the `dataPtr` argument (after the command byte).

This function must ensure that MPI transactions adhere to the timing requirements set forth in the CSLAC device's *Chipset User's Guide*.

Note that this function call is blocking by nature. One method to overcome this scenario is based on an implementation where this function could make use of a hardware supporting function that can completely handle an MPI block transaction with maximum 256 bytes (`cmd` + data specified by `cmdLen`). While the hardware supporting function handles the MPI transaction, this function could be put on hold to wait for a signal for the MPI transaction to be completed and at the same time the CPU, which is executing this function, could be made free to attend to other services.

The API-II provided by Zarlink Semiconductor contains an example implementation of this function.

RETURNS None

DEVICES CSLAC

TERMINATIONS All

10.2.2 VpMpiReset()

SYNTAX

```
void
VpMpiReset(
  VpDeviceIdType deviceId,        /* Selects the target CSLAC device */
  VpDeviceType deviceType         /* Device type */
```

DESCRIPTION

This function performs a hardware reset of the VTD. Different devices might employ different schemes to reset the device, specified in the respective device's *Chipset User's Guide*. The VP-API calls this function to reset the device and start from a known state.

The argument `deviceId` identifies the chip that needs to be reset and the argument `deviceType` specifies the type of device (see *Objects and Contexts*, on page 19).

RETURNS

None

DEVICES

CSLAC

TERMINATIONS

All

CHAPTER

11 INTERRUPT HANDLING

11.1 OVERVIEW

This chapter discusses how the VP-API handles VTD interrupts. VP-API functions must be invoked in response to VTD hardware interrupts. Depending on the type of interrupt, the VP-API may update its internal state or carry out other actions.

The complexity of the VP-API interrupt service routines varies depending on the type of device(s) being controlled. Specifically, the interrupt service routines for CSLAC devices are significantly more complex than those for the VCP and VPP devices. This is due to the fact that much of the real-time functionality implemented in the VP-API for CSLAC is actually implemented within the VCP and VPP devices themselves. This version of the document describes only the CSLAC device interrupt handling requirements.

11.2 HANDLING INTERRUPTS FROM CSLAC DEVICES

CSLAC devices provide comparatively less advanced functions compared to the VCP and VPP devices. This forces the VP-API to implement necessary extra functions to provide a common look and feel at the VP-API level. One of the support functions VP-API needs for the CSLAC class of devices is a timing function.

This timing information is needed to evaluate and control various types of functions in VP-API like debouncing supervisory data, cadencing etc. This periodic source of interrupt to VP-API could be combined with interrupts from CSLAC devices to produce various types of implementation architectures. In the following sections we will study the need for time base, and how it can be combined with CSLAC interrupts to produce various implementation architectures.

11.2.1 VP-API Time Base

The VP-API performs some timing and time-based operations, and therefore requires a method for keeping time. To provide this time base, applications utilizing the VP-API are required to make a periodic function call at regular intervals. Specifically, the application should call **`VpApiTick()`** once at the beginning of every “tick” period. The tick period is user-defined. The VP-API uses the periodic tick to:

- Debounce the SLAC device’s supervisory inputs[1]
- Provide cadence timing for call progress tones and meter signals
- Service Caller ID transmission
- Sequence the line tests

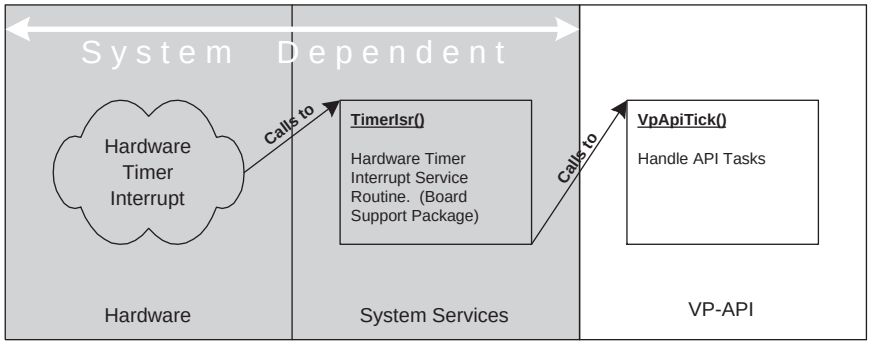
Generally, the customer application sets up a timer interrupt with an interrupt service routine (ISR) calling, or setting a flag to call the **`VpApiTick()`** function.

11.2.1.1 VP-API Tick derived from a Timer Interrupt - Example

Zarlink Semiconductor’s demo applications implement the VP-API Tick rate run off from a hardware timer interrupt. The demo applications attach the timer’s ISR (located in the system services layer) to a repeating, hardware timer interrupt. This timer ISR sets a flag to call **`VpApiTick()`** every time the interrupt occurs. The flag applies to every SLAC device in the system, and thus causes the tick function to be called once for each SLAC device. In Zarlink Semiconductor’s VP-API the tick interval is configurable, but for Zarlink Semiconductor’s demo applications, each interrupt occurs every 5 ms. Figure 11–1 shows the hardware interrupt event chain.

1. *The SLAC device hardware, in most cases, performs all required debouncing. However, there are some circumstances in which the VP-API tick function will perform additional debouncing (for example, ring-trip on the QSLAC/VE580 Series devices and longer debounce required that cannot be implemented by the device alone).*

Figure 11–1 Basic Event Chain



Notes:

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.

11.2.1.2 VP-API Tick Details

In addition to providing the timing base for the VP-API, **`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 11.2.2), 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.

During each tick period, more than one interrupt may occur. **`VpApiTick()`**, however, limits the number of interrupts serviced during one invocation by a value set in the Device Profile created by running ProfileWizard. The maximum interrupts and VP-API tick rate should be set such that the tick function execution time can never be longer than one VP-API tick rate period. This will be determined according to each customer's hardware platform, the number of SLAC devices in the system, and the efficiency of their HAL layer functions.

11.2.2 SLAC Device Interrupt Architecture Variations

The final requirement of an application utilizing the VP-API is providing the VP-API 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 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 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 tick function services SLAC device interrupts only when the SLAC device interrupt line is active, which is polled every VP-API 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 Tick for an interrupt, whether one has occurred or not.

All four methods are supported by Zarlink Semiconductor'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 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 9, on page 111. 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.

11.2.2.1 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 tick (See Figure 11–2).
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 11–3).

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 interrupt(s) on the next VP-API 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 tick. See Section 11.2.1 for further information.*

Notes:

The QSLAC/VE580 Series Data Sheets recommend using Level Triggered Interrupts with QSLAC/VE580 series devices. See Section 11.2.2.2 for further information on using Level Triggered Interrupts with the VP-API.

Figure 11–2 Mode 1: Interrupt Driven, Edge-Triggered (Flow)

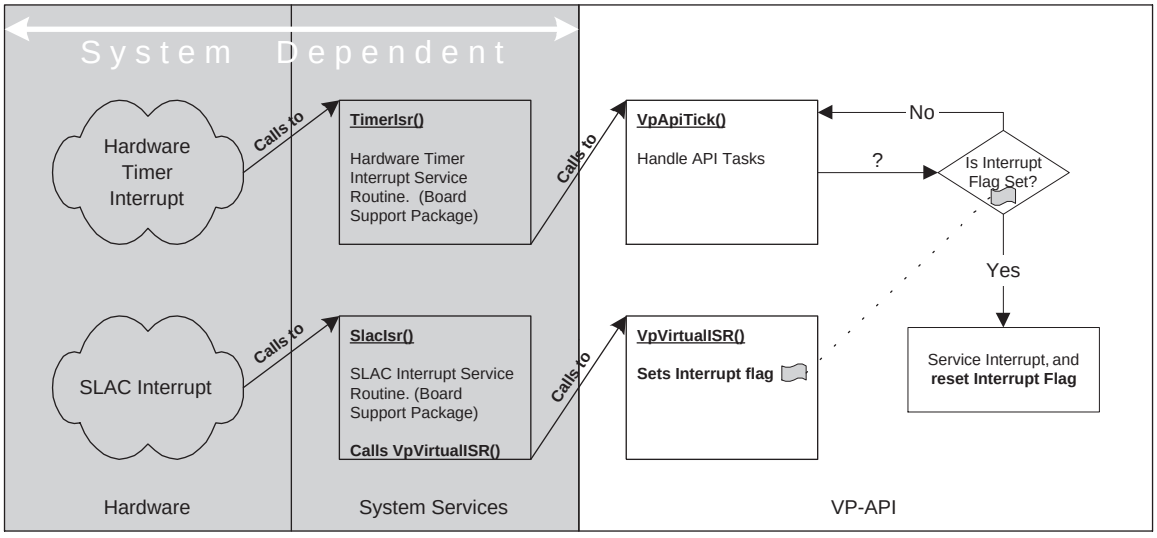
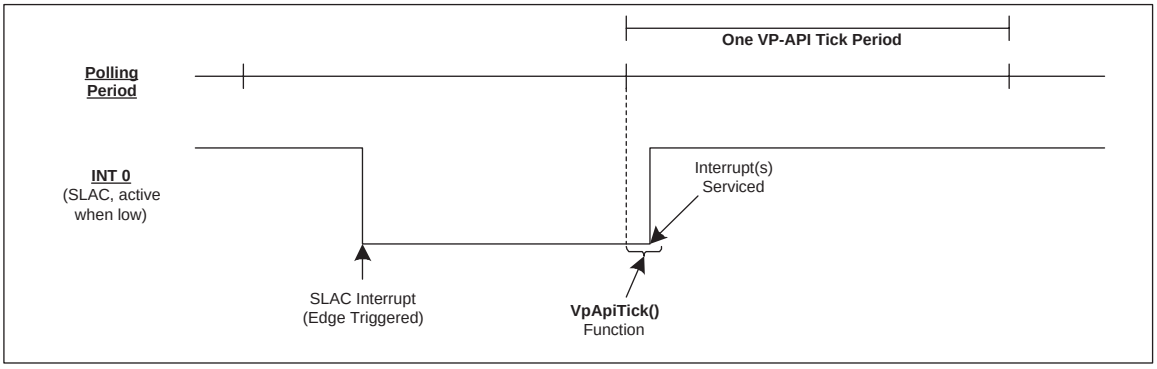


Figure 11–3 Mode 1: Interrupt Driven, Edge-Triggered (Timing)



11.2.2.2 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 11–4.

4. **`VpApiTick()`** services the interrupt indicated by the flag. See Figure 11–5.
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()`**.

Zarlink Semiconductor'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 tick. See Section 11.2.1 for further information.*

Notes:

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

Figure 11–4 Mode 1: Interrupt Driven, Level Triggered (Flow)

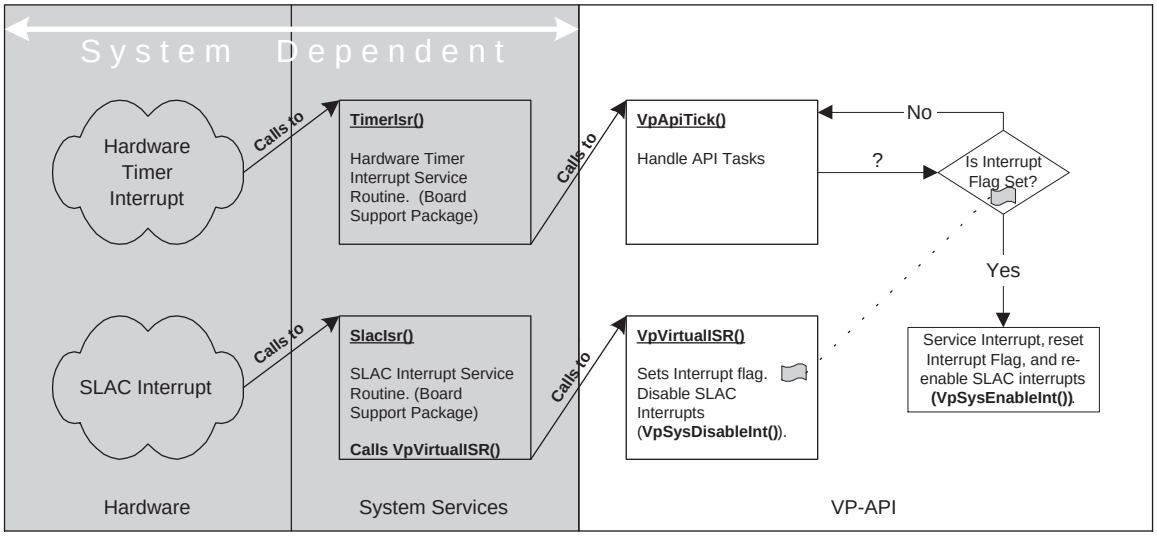
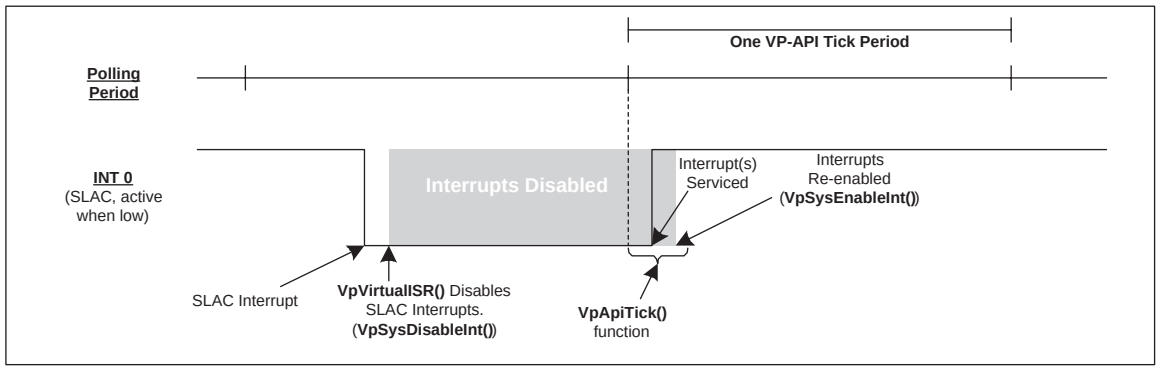


Figure 11–5 Mode 1: Interrupt Driven, Level Triggered (Timing)



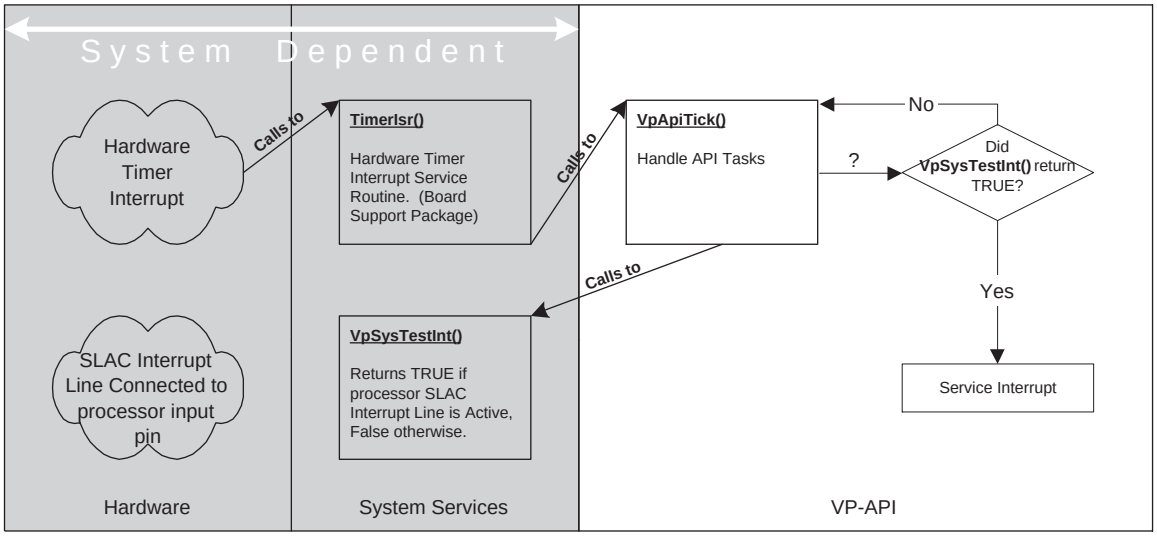
11.2.2.3 Mode 3: Efficient Polled Mode

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 tick. See* Section 11.2.1 *for further information.*

Figure 11–6 Efficient Polling



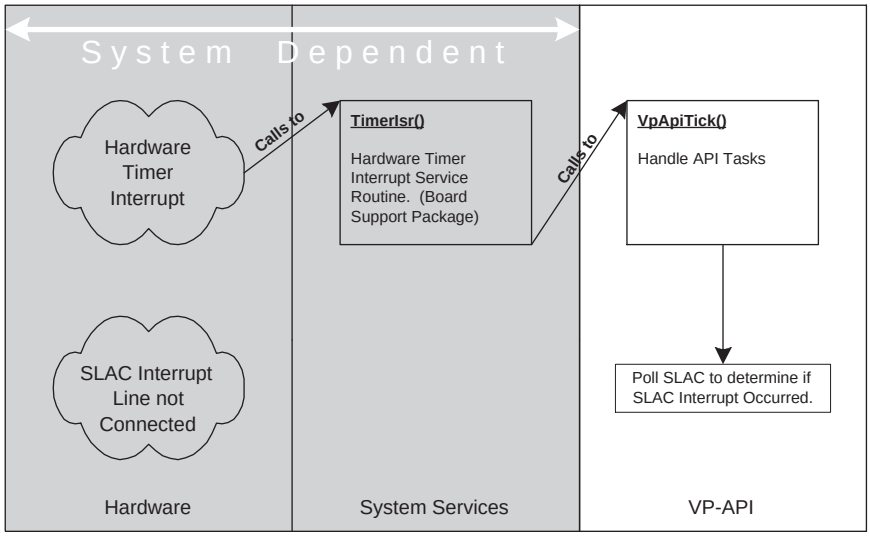
11.2.2.4 Mode 4: Simple Polled Mode

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 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 tick. See* Section 11.2.1 *for further information.*

Figure 11–7 Polled Interrupt Servicing



CHAPTER

A GLOSSARY

Channel	See Section 1.1.2
Codec	Coder/Decoder
CSLAC	Conventional Zarlink Semiconductor 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 VPP/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.
HBI	Host Bus Interface. This is the host's interface to the VCP or VPP.
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 Zarlink Semiconductor's serial control interface for CSLAC devices.
NTR	Network Timing Reference
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 Zarlink Semiconductor trademark.
SLIC	Subscriber Line Interface Circuit

SPI	Serial Peripheral Interface. This is a four wire serial control interface between the VCP/VPP 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)	An Application Program Interface that provides access to Zarlink Semiconductor's VTDs via the HBI or MPI. It is the primary component of the VoicePath Software Development Kit (VP SDK).
VoicePath™ SDK (VP-SDK)	A collection of tools to assist in the development of software for Zarlink Semiconductor devices. The VoicePath API and ProfileWizard are components of the VP SDK.
VCP	Voice Control Processor, the new name for the device formerly known as Digital Voice Processor (VCP). For a complete list of VCP products please see *Supported Hardware Configurations*, on page 5.
VPP	Voice Packet Processor. For a complete list of VPP products, please see *Supported Hardware Configurations*, on page 5.
VTD	Voice Termination Device. A VTD can be a SLAC device, VCP or VPP. Note that SLAC devices are *controlled by* a VCP but are *contained within* VPPs.

CHAPTER

B FUNCTION INDEX

Table B–1 lists all VoicePath API functions, along with their input types, return type, applicable devices, and applicable termination types. 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	23
VpMakeLineObject	VpTermType termType, uint8 channelId, VpLineCtxType *pLineCtx, void *pLineObj, VpDevCtxType *pDevCtx	VpStatusType	All	All	24
VpMakeDeviceCtx	VpDeviceType deviceType, VpDevCtxType *pDevCtx, void *pDevObj	VpStatusType	All	All	26
VpMakeLineCtx	VpLineCtxType *pLineCtx, void *pLineObj, VpDevCtxType *pDevCtx	VpStatusType	All	All	27
VpFreeLineCtx	VpLineCtxType *pLineCtx	VpStatusType	All	All	28
VpGetDeviceInfo	VpDeviceInfoType *pDeviceInfo	VpStatusType	All	All	29
VpGetLineInfo	VpLineInfoType *pLineInfo	VpStatusType	All	All	30
VpMapLineId	VpLineCtxType *pLineCtx, VpLineIdType lineId	VpStatusType	All	All	31
Initialization Functions					
VpBootLoad	VpDevCtxType *pDevCtx, VpBootStateType state, VpImagePtrType pImageBuffer, uint32 bufferSize, VpScratchMemType *pScratchMem, VpBootModeType validation	VpStatusType	VCP, VPP	All	
VpInitDevice	VpDevCtxType *pDevCtx, VpProfilePtrType pDevProfile, VpProfilePtrType pAcProfile, VpProfilePtrType pDcProfile, VpProfilePtrType pRingProfile, VpProfilePtrType pFxoAcProfile, VpProfilePtrType pFxoCfgProfile	VpStatusType	All	All	66

Table B–1 VoicePath™ API II Functions Summary(Continued)

Function Name	Arguments	Return Type	Devices	Terminations	Page
VpInitLine	VpLineCtxType *pLineCtx, VpProfilePtrType pAcProfile, VpProfilePtrType pDcFeedOrFxoCfgProfile, VpProfilePtrType pRingProfile	VpStatusType	All	All	68
VpConfigLine	VpLineCtxType *pLineCtx, VpProfilePtrType pAcProfile, VpProfilePtrType pDcFeedOrFxoCfgProfile, VpProfilePtrType pRingProfile	VpStatusType	All	All	69
VpCalCodec	VpLineCtxType *pLineCtx, VpDeviceCalType mode	VpStatusType	CSLAC-790, VCP-790	FXS	70
VpCalLine	VpLineCtxType *pLineCtx	VpStatusType	VCP-790	FXS	
VpInitRing	VpLineCtxType *pLineCtx, VpProfilePtrType pCadProfile, VpProfilePtrType pCidProfile	VpStatusType	All	FXS	71
VpInitCid	VpLineCtxType *pLineCtx, uint8 length, uint8p pCidData	VpStatusType	All	FXS	72
VpInitMeter	VpLineCtxType *pLineCtx, VpProfilePtrType pMeterProfile	VpStatusType	All	FXS	73
VpInitProfile	VpDevCtxType *pDevCtx, VpProfileType type, VpProfilePtrType pProfileIndex, VpProfilePtrType pProfile	VpStatusType	All	All	74
VpSoftReset	VpDevCtxType *pDevCtx	VpStatusType	VCP, VPP	All	
VpSetBatteries	VpLineCtxType *pLineCtx, VpBatteryModeType battMode, VpBatteryValuesType *pBatt	VpStatusType	VCP	All	75
Control Functions					
VpSetLineState	VpLineCtxType *pLineCtx, VpLineStateType state	VpStatusType	All	All	78
VpSetLineTone	VpLineCtxType *pLineCtx, VpProfilePtrType pToneProfile, VpProfilePtrType pCadProfile, VpDtmfToneGenType *pDtmfControl	VpStatusType	All	All	80

Table B–1 VoicePath™ API II Functions Summary(Continued)

Function Name	Arguments	Return Type	Devices	Terminations	Page
VpSetRelayState	VpLineCtxType *pLineCtx, VpRelayControlType rState	VpStatusType	CSLAC, VCP-790, VPP	All	81
VpSetRelGain	VpLineCtxType *pLineCtx, uint16 txLevel, uint16 rxLevel, uint16 handle	VpStatusType	VCP, CSLAC-880, CSLAC-890	All	
VpSendSignal	VpLineCtxType *pLineCtx, VpSendSignalType signalType, void *pSignalData	VpStatusType	CSLAC-880, CSLAC-890, VCP-880	All	83
VpSendCid	VpLineCtxType *pLineCtx, uint8 length, VpProfilePtrType pCidProfile, uint8p pCidData	VpStatusType	All	FXS	86
VpContinueCid	VpLineCtxType *pLineCtx, uint8 length, uint8p pCidData	VpStatusType	CSLAC, VCP	FXS	87
VpDtmfDigitDetected	VpLineCtxType *pLineCtx, VpDigitType digit, VpDigitSenseType sense	VpStatusType	CSLAC	FXS	88
VpStartMeter	VpLineCtxType *pLineCtx, uint16 onTime, uint16 offTime, uint16 numMeters	VpStatusType	All	FXS	89
VpSetOption	VpLineCtxType *pLineCtx, VpDevCtxType *pDevCtx, VpOptionIdType option, void *pValue	VpStatusType	All	All	90
VpToneDetectionControl	VpLineCtxType *pLineCtx, VpToneDetectionType *pToneDetection	VpStatusType	VPP	All	
VpDeviceIoAccess	VpDevCtxType *pDevCtx, VpDeviceIoAccessDataType *pDeviceIoData	VpStatusType	All	All	91
VpVirtualISR	VpDevCtxType *pDevCtx	VpStatusType	CSLAC	All	92
VpApiTick	VpDevCtxType *pDevCtx, bool *pEventStatus	VpStatusType	CSLAC	All	93
VpSelfTest	VpLineCtxType *pLineCtx	VpStatusType	VCP	All	
VpFillTestBuf	VpLineCtxType *pLineCtx, uint16 length, VpVectorPtrType pData	VpStatusType	VCP-790-BT, VCP-790-AT	All	
VpLowLevelCmd	VpLineCtxType *pLineCtx, uint8 *pCmdData, uint8 len, uint16 handle	VpStatusType	All	All	94

Table B–1 VoicePath™ API II Functions Summary(Continued)

Function Name	Arguments	Return Type	Devices	Terminations	Page
VpSetBFilter	VpLineCtxType *pLineCtx, VpBFilterModeType bFiltMode, VpProfilePtrType pAcProfile	VpStatusType	All	All	95
VpLineIoAccess	VpLineCtxType *pLineCtx, VpLineIoAccessType *pLineIoAccess, uint16 handle	VpStatusType	VCP2	All	96
VpDeviceIoAccessExt	VpDevCtxType *pDevCtx, VpDeviceIoAccessExtType *pDeviceIoAccess	VpStatusType	VCP2	All	97
Status and Query Functions					
VpGetEvent	VpDevCtxType *pDevCtx, VpEventType *pEvent	bool	All	All	100
VpGetLineStatus	VpLineCtxType *pLineCtx, VpInputType input, bool *pStatus	VpStatusType	All	All	102
VpGetDeviceStatus	VpDevCtxType *pDevCtx, VpInputType input, uint32 *pDeviceStatus	VpStatusType	All	All	103
VpGetLoopCond	VpLineCtxType *pLineCtx, uint16 handle	VpStatusType	CSLAC-790, VCP, VPP	FXS	104
VpGetOption	VpLineCtxType *pLineCtx, VpDevCtxType *pDevCtx, VpOptionIdType option, uint16 handle	VpStatusType	All	All	105
VpGetLineState	VpLineCtxType *pLineCtx, VpLineStateType *pCurrentState	VpStatusType	All	All	106
VpFlushEvents	VpDevCtxType *pDevCtx	VpStatusType	All	All	107
VpGetResults	VpEventType *pEvent, void *pResults	VpStatusType	All	All	108
VpClearResults	VpDevCtxType *pDevCtx	VpStatusType	All	All	109
VpCodeCheckSum	VpDevCtxType *pDevCtx, uint16 handle	VpStatusType	VCP, VPP	All	
VpGetDeviceStatusExt	VpDevCtxType *pDevCtx, VpDeviceStatusType *pDeviceStatus	VpStatusType	VCP2	All	109

Table B–1 VoicePath™ API II Functions Summary(Continued)

Function Name	Arguments	Return Type	Devices	Terminations	Page
Packet Functions					
VpControlVoiceStream	VpLineCtxType *pLineCtx, uint8 streamId, VpVSConfControlType streamControl, VpVSConfModeType confMode	VpStatusType	VPP	All	
VpReadUpStreamPacket	VpLineCtxType *pLineCtx, VpPktDataPtrType pPacketData, uint8 streamId	VpStatusType	VPP	All	
VpWriteDownStreamPacket	VpLineCtxType *pLineCtx, VpPktDataPtrType pPacketData, uint8 streamId	VpStatusType	VPP	All	
VpGetTimeStamp	VpDevCtxType *pDevCtx, uint16 *pTimeStamp	VpStatusType	VPP	All	
VpReadPacketStatistics	VpLineCtxType *pLineCtx, uint16 handle	VpStatusType	VPP	All	
Testing Functions					
VpTestLine	VpLineCtxType *pLineCtx, VpTestIdType test, const void *pArgs, uint16 handle	VpStatusType	VCP-790-BT, VCP-790-AT, VPP	FXS	
System Services Layer Functions					
VpSysEnterCritical	VpDeviceIdType deviceId, VpCriticalSecType criticalSecTyp	uint8	All	All	113
VpSysExitCritical	VpDeviceIdType deviceId, VpCriticalSecType criticalSecType	uint8	All	All	114
VpSysWait	uint8 time	void	CSLAC	All	115
VpSysDisableInt	VpDeviceIdType deviceId	void	CSLAC	All	116
VpSysEnableInt	VpDeviceIdType deviceId	void	CSLAC	All	117
VpSysTestInt	VpDeviceIdType deviceId	bool	CSLAC	All	118
VpSysDtmfDetEnable	VpDeviceIdType deviceId	void	CSLAC	FXS	119
VpSysDtmfDetDisable	VpDeviceIdType deviceId	void	CSLAC	FXS	119
Hardware Abstraction Layer (HAL) Functions					

Table B–1 VoicePath™ API II Functions Summary(Continued)

Function Name	Arguments	Return Type	Devices	Terminations	Page
VpMpiCmd	VpDeviceIdType deviceId, uint8 ecVal, uint8 cmd, uint8 cmdLen, uint8 *dataPtr	void	CSLAC	All	122
VpMpiReset	VpDeviceIdType deviceId, VpDeviceType deviceType	void	CSLAC	All	123
VpHalHbiInit	VpDeviceIdType deviceId	bool	VCP, VPP	All	
VpHalHbiCmd	VpDeviceIdType deviceId, uint16 cmd	bool	VCP, VPP	All	
VpHalHbiWrite	VpDeviceIdType deviceId, uint16 cmd, uint8 numwords, uint16p pData	bool	VCP, VPP	All	
VpHalHbiWrite8	VpDeviceIdType deviceId, uint16 cmd, uint8 numwords, uint8p pData	bool	VPP	All	
VpHalHbiRead	VpDeviceIdType deviceId, uint16 cmd, uint8 numwords, uint16p pData	bool	VCP, VPP	All	
VpHalHbiRead8	VpDeviceIdType deviceId, uint16 cmd, uint8 numwords, uint8p pData	bool	VPP	All	
VpHalHbiBootWr	VpDeviceIdType deviceId, uint8 numwords, VpImagePtrType pData	bool	VCP, VPP	All	

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 81. Only those relay states that are described in this section are valid relay states for a given line termination types.

Figure 3–1 Relay States, VP_TERM_FXS_GENERIC Termination

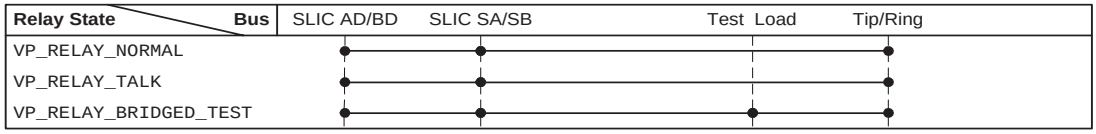
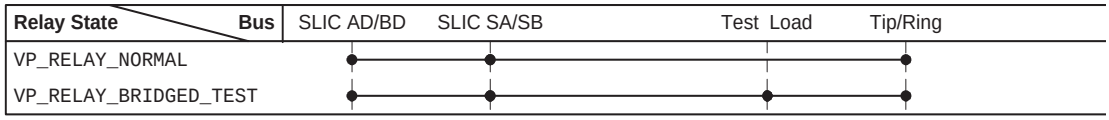


Figure 3–2 Relay States, VP_TERM_FXS_ISOLATE Termination



Note:

Even though the `VP_TERM_FXS_ISOLATE` line termination type has a drive isolate relay, this relay cannot be controlled through the `VpSetRelayState()` function. This relay is automatically controlled by the `VpSetLineState()` function. This limitation prevents damage to the device due to improper combination of the relay and line states which could result in very high voltages being generated from devices that have switcher circuitry.

Figure 3–3 Relay States, VP_TERM_FXS_TITO_TL_R Termination

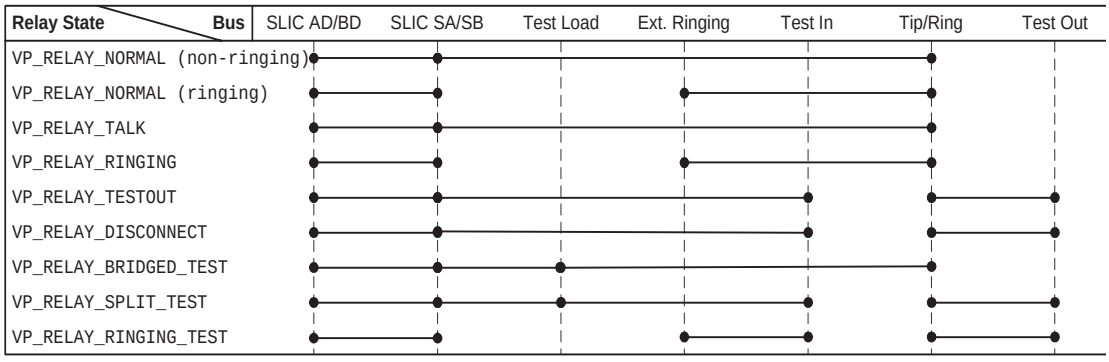


Figure 3–4 Relay States, VP_TERM_FXS_75181 Termination

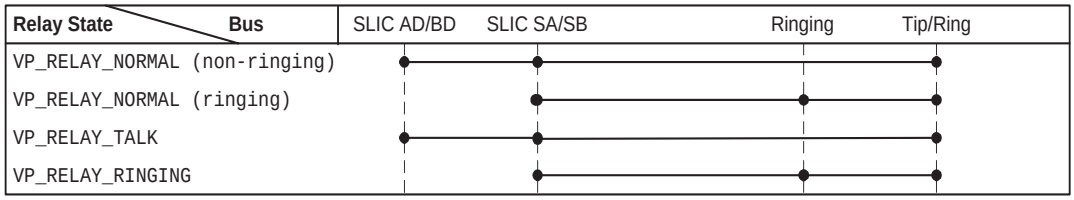


Figure 3–5 Relay States, VP_TERM_FXS_75282 Termination

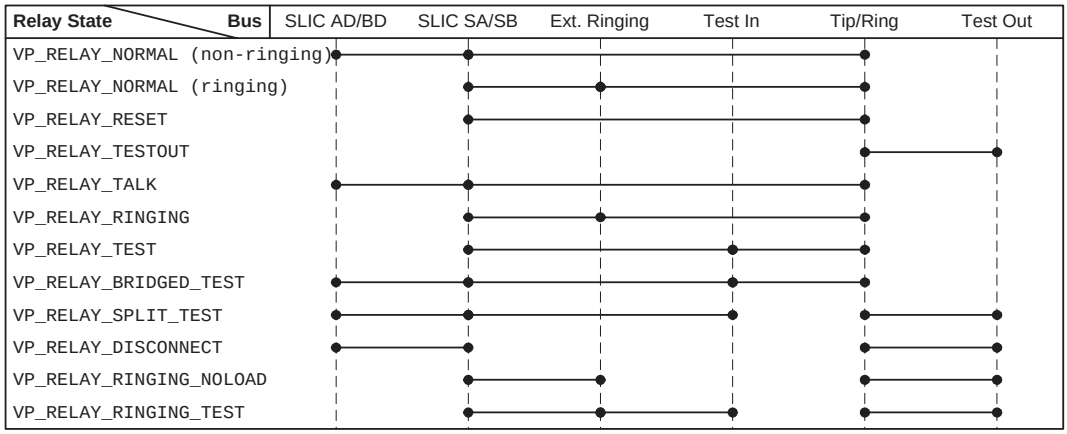


Figure 3–6 Relay States, VP_TERM_FXS_RR Termination

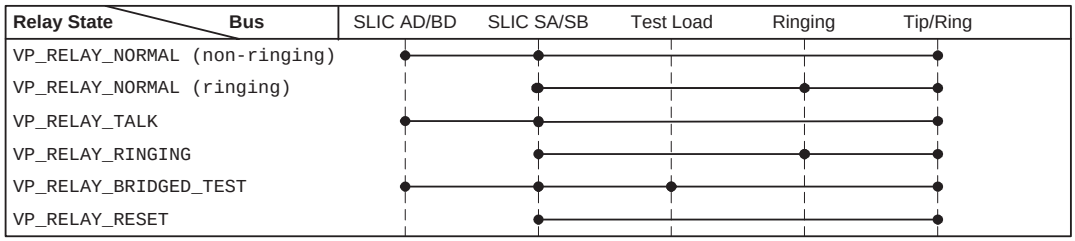
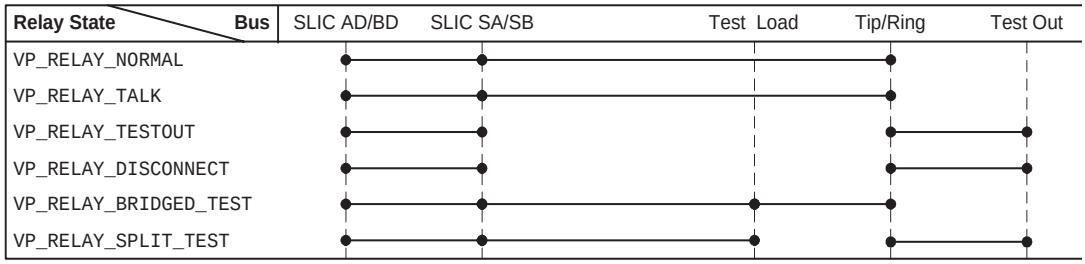


Figure 3–7 Relay States, VP_TERM_FXS_TO_TL Termination



CHAPTER

D REVISION HISTORY

REV B1 – 12/19/2005

- Added new line termination types. Updated sections of the document that deal with termination types. Also, changed the name of the previously un-documented `VP_FXS_TERM_RDT` to `VP_FXS_TERM_RR`.
- Updated **`VpInitDevice()`** function to indicate the end relay states for all the line termination types.
- Added a new appendix to illustrate all the applicable relay states for various line termination types. Also updated **`VpSetRelayState()`** function.
- Clarified the smooth polarity reversal scenarios in the `VP_OPTION_ID_LINE_STATE` option.
- Removed references to code in VpApiTick() regarding virtual register data stored in the API-II, and in regard to nested critical sections. Notes previously described that critical section nesting is not currently used in the API-II. This is not the case and has been removed from the document.
- Updated section regarding interrupt modes for CSLAC family. Maximum number of interrupts is set by Device Profile, previously documented as set by MAX_INTERRUPT (used in VP-API).
- Added VpMakeDeviceCtx() and VpMakeLineCtx() as functions that can return error code VP_STATUS_ERR_VTD_CODE in table 1.5.

REV D1 – 3/31/2006

- Added a new function to the VP-API. This function enables the applications to assign implementation specific system wide line identifier to a line. Please See *VpMapLineId()*, on page 31..

REV E1 - 08/02/2006

- Added new `VpSendSignal()` types for Forward Disconnect and Polarity Reversal generation on the FXS line.
- Added Wideband CODEC option to `VP_OPTION_ID_CODEC` Mode.
- Added new `VpLineStateType`, `VP_LINE_STANDBY_POLREV` for on-hook, low power, polarity reversal state.
- Added new member in `VpDeviceInfoType` structure for silicon revision (`uint8 revCode`).
- Added new option type to specify a second set of Dial Pulse Parameters (`VP_DEVICE_OPTION_ID_PULSE2`). Updated event data for event `VP_LINE_EVID_PULSE_DIG` to indicate the set of Dial Pulse Parameters that passed the currently detected digit.
- Updated event data for `VP_LINE_EVID_POLREV` to indicate direction of polarity reversal that was detected.
- Added `VP_INPUT_POLREV` for FXO line polarity reversal status to `VpGetLineStatus()`.
- Added new Process type event `VP_LINE_EVID_TONE_CAD` indicating completion of a Tone Cadence.
- Added CSLAC-880 to supported devices for `VpSetRelGain()`
- Provided clarification of DTMF Tone Generation using parameter `pDtmfControl` in function `VpSetLineTone()`.

REV E2 - 10/02/2006

- Added VP_TERM_FXO_DISC to termination types.
- Added System Configuration information for VP580 support.

REV E3 - 12/19/2006

- Updated VpSetLineTone() to indicate event generated.
- Provided clarification throughout with no API-II interface change.

REV G1 - 10/3/2007

- Added new functions VpSetBFilter() and VpSetBatteries() with documentation for use.
- Added new options VP_DEVICE_OPTION_ID_DEV_IO_CFG and VP_OPTION_ID_LINE_IO_CFG.
- Updated the VP_OPTION_ID_DTMF_MODE documentation to reflect the new struct member (dtmfDetectionEnabled[]) for supporting more than 32 lines.
- Added new API functions: VpLineIoAccess(), VpDeviceIoAccess(), and VpGetDeviceStatusExt().
- Added new events: VP_LINE_EVID_LINE_IO_RD_CMP, and VP_LINE_EVID_IO_WR_CMP.
- Updated description of VP_DEV_EVID_IO_ACCESS_CMP to discuss VpDeviceIoAccessExt().
- Added new option VP_OPTION_ID_DTMF_SPEC and content to describe use.
- Added new device type VP_DEV_VCP2_SERIES and updated configuration options.

REV G2 - 1/4/2007

- Removed Revision History more than 2 years old.
- Added new events: VP_LINE_EVID_EXTD_FLASH
- Added new parameter `onHookMin` to `VpOptionPulseType`.
- Added VP_SEND_SIG_TIP_OPEN_PULSE to `VpSendSignalType`.
- Added CSLAC-890 device to CSLAC family of API-II.