# VoicePath™ API-II Reference Guide

**Part Number:** ZL880/miSLIC™

**Document Id#:** PD-000328629

**Revision Number:** 11

**Issue Date:** July 23, 2018

This page left intentionally blank

# TABLE OF CONTENTS

CHAPTER 1

# INTRODUCTION

## 1.1 ABOUT THIS USER'S GUIDE

This document describes the Microsemi VoicePath™ Application Program Interface (VP-API-II) for the ZL880 and miSLIC devices. The VP-API-II is an API layer between the Application and Microsemi silicon used to control Microsemi's telephony Voice Termination Devices (VTDs). This chapter highlights the document structure and summarizes the VP-API-II architecture and features.

## 1.2 CHAPTER OVERVIEW

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 1.3 FREQUENTLY USED TERMS

The following terms are used extensively throughout this document:

**Device:** The *device* refers to the Microsemi silicon connected directly to the Host Controller. See Table 1–3 for a list of the supported devices.

**Channel:** The *channel* refers to the set of resources inside the silicon that are used to provided some of the functionality for a given *line*. In general, line specific functions can be performed on all lines of the device at the same time unless otherwise stated. If a conflict occurs, the VP-API-II will return a “Resource Busy” error.

***Notes:***

*Device resources alone are not enough to perform all functionality of a line. Complete line functionality requires external components (e.g., relays, detector/sense circuits, overvoltage/overcurrent protection, supplies, etc.).*

**Line:** The *line* refers to the complete set of hardware (silicon and components) controlled by the VP-API-II that is used to provide the set of functionality supported. The *line* concept is the most important concept in the VP-API-II and will be discussed repeatedly throughout this document.

For the remainder of this document the terms *line* and *termination type* will be used interchangeably.

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

## 1.4 DOCUMENTATION CONVENTIONS

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

**Table 1–1 Documentation Conventions**

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

## 1.5 VP-API-II OVERVIEW

The VP-API-II library is an ANSI C99 compliant set of C source code that provides a standard software interface for controlling/querying and passing digitized voice through a set of subscriber lines using Microsemi silicon. The VP-API-II hides the details of the silicon, allowing software developers to focus on the Application instead of the hardware.

### 1.5.1 Features

Listed below are some of the key features of the VP-API-II.

- Provides a common software interface.
- OS agnostic (used with any OS and Kernel/User mode)
- Supports dynamic and static memory models.
- Seamless integration with Microsemi VeriVoice™ Professional Line Test SW (LT-API).
- Operates from the calibration values generated by Microsemi VeriVoice™ Manufacturing Test Software (VVMT)

## 1.5.2 Profiles

Profiles are used by the VP-API-II to configure the device and lines with Application specific parameters. They are C strings of `uint8` data containing a combination of silicon (i.e., commands and command data) and VP-API-II usable values. Table 1–2 lists the supported ZL880/miSLIC profiles.

**Table 1–2 Profile Type to Function Mapping**

| Profile Type | Description | Used In Function |
|---|---|---|
| Device | Device Configuration and System Parameters. | `VpInitDevice()` |
| AC | 2-Wire Impedance, Transmit and Receive Level and Frequency Response, and 4-Wire Transmission Loss (known also as "Hybrid Balance") | `VpInitDevice()` |
| | | `VpInitLine()` |
| | | `VpConfigLine()` |
| | | `VpCal()` (VE890-FXO only) |
| DC | Line Feed Characteristics and Loop Supervision Parameters | `VpInitDevice()` |
| | | `VpInitLine()` |
| | | `VpConfigLine()` |
| Ringing | Ringing configuration (frequency, type, amplitude, DC bias). | `VpInitDevice()` |
| | | `VpInitLine()` |
| | | `VpConfigLine()` |
| Tone | Specifies frequency and Amplitude for up to 4 tones. | `VpSetLineTone()` |
| Cadence (Tone / Ringing) | Cadence Profiles are user defined (pre-generated) sequences of `VpLineState` changes and tone generator output to the line. | `VpSetLineTone()` (used with Tone Cadences) |
| | | `VpInitRing()` (used with Ringing Cadences) |
| Caller ID | Specifies the Caller ID signal type (DTMF or FSK), level, line state changes, tone generation and detection, and other Caller ID related elements to support Type I and Type II Caller ID. | `VpInitRing()` (used for Type I CID) |
| | | `VpSendCid()` (generally used for Type II CID) |
| Metering | Specifies Metering Signal Type and maximum reflected signal. | `VpInitMeter()` |
| Calibration (generated by the VP-API-II or VVMT Software Package) | Device and Line values necessary to operate the system to within Data Sheet Specifications. | `VpCal()` |
| All | For Profile Tables, which has limited use in ZL880 and miSLIC Applications. | `VpInitProfile()` |

## 1.5.3 Options

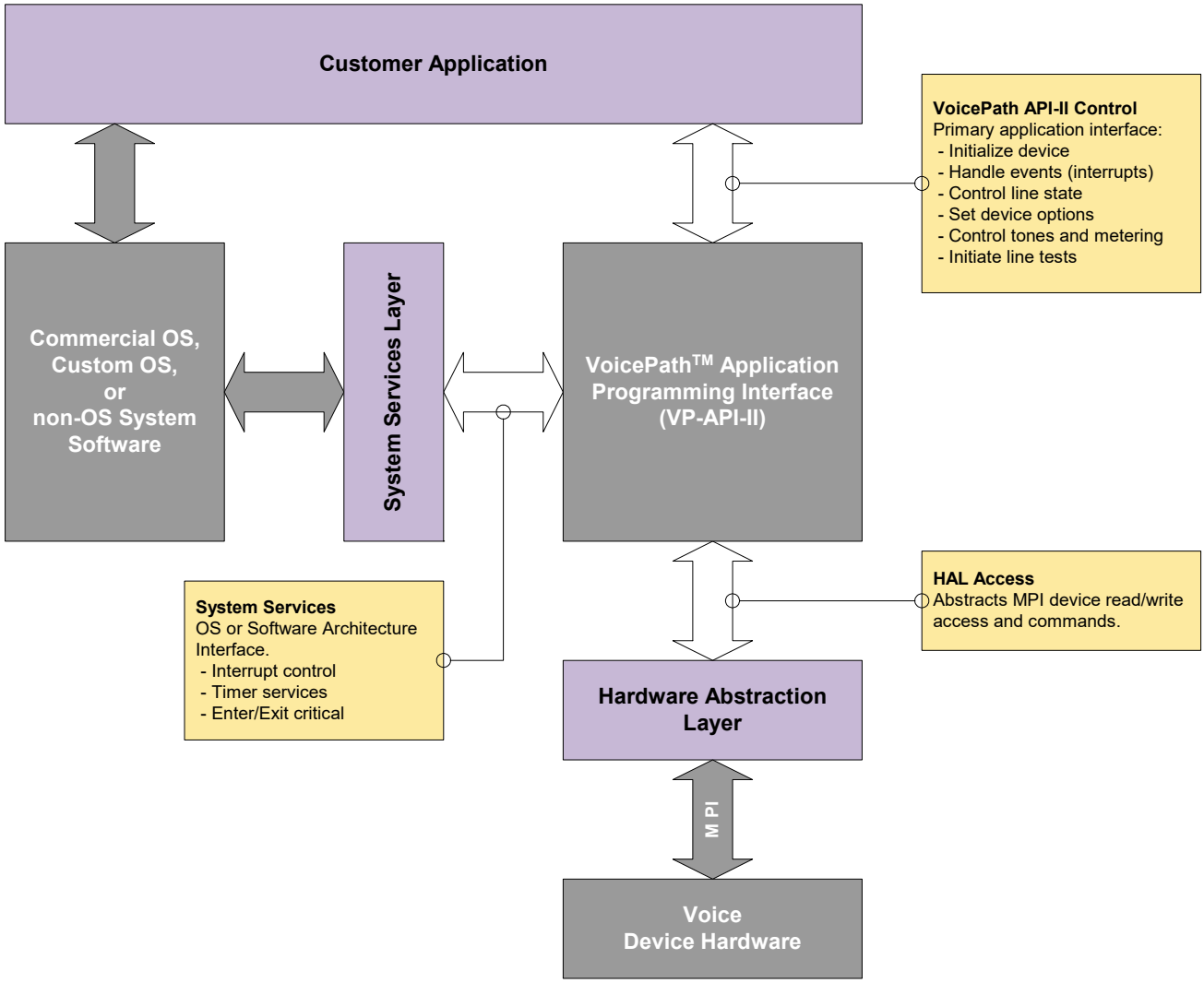
From an Application perspective, Options are similar to Profiles with the following key differences:

- Unless otherwise stated, Options can be retrieved by the VP-API-II whereas Profiles cannot.
- Option values can be changed at run-time, Profile content cannot (defined as `const` data).
- Options are set using `VpSetOption()` and retrieved using `VpGetOption()`. A complete list of the VP-API-II Options can be found at Chapter 3, Options. Profiles are used in the functions listed in Table 1–2.

## 1.6 SOFTWARE ARCHITECTURE USING THE VP-API-II

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

**Figure 1–1 Software Block Diagram**




### 1.6.1 VP-API-II

This block is the set of code described in this document and is provided by Microsemi. It abstracts the silicon details from the Application and runs on the host microprocessor.

*Note that although the VP-API-II is a common interface for a wide variety of Microsemi voice products, this document describes only the features that are available with the ZL880 and miSLIC VP-API-II.*

Microsemi provides two versions of the VP-API-II:

- VP-API-II (OPN Le71SK0002): Supports all operations described in this document.
- VP-API-II Lite (OPN Le71SDKAPIL): Same as Le71SK0002 except removes support for VP-API-II Tone Cadencing (i.e., tone generation on/off-times must be controlled by Application) and functions: `VpInitRing()`, `VpInitCid()`, `VpInitMeter()`, `VpSendSignal()`, `VpSendCid()`, `VpContinueCid()`, and `VpStartMeter()`.

### 1.6.2 Customer Application

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

### 1.6.3 Operating System

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

### 1.6.4 Hardware Abstraction Layer

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

### 1.6.5 System Services Layer

This block is implemented by the user. It abstracts platform-specific functions such as interrupt control and timing services. This layer derives the functions required by the VP-API-II from the facilities provided by the underlying hardware or operating system. The VP-API-II accesses this layer through the functions described in Chapter 11, System Services.

## 1.7 SUPPORTED HARDWARE CONFIGURATIONS

The VP-API-II allows Applications to manage any combination of device types (specified as `VpDeviceType`) and line types (specified as `VpTermType`) supported by the VP-API-II. This document lists the `VpDeviceType` and `VpTermType` supported for the ZL880/miSLIC silicon only. Note from previous discussion:

- `VpDeviceType` refers to the device directly connected to the Host Processor
- `VpTermType` refers to the complete set of silicon and circuitry used to provide the full FXS functionality supported in the VP-API-II.

Table 1–3 lists the **`VpDeviceType`** constants supported by the ZL880/miSLIC VP-API-II. The *Part Number* column indicates the device the host microprocessor is connected to. The *Configuration Name* column lists the terminology used throughout this document to refer to the ZL880/miSLIC device configuration. The VP-API-II supports simultaneously any combination of the configurations shown in Table 1–3 subject to the limitations of the target hardware.

**Table 1–3 Supported ZL880/miSLIC Device Configurations**

| VpDeviceType | Operating Mode | Part Numbers | Part Family |
|---|---|---|---|
| VP_DEV_887_SERIES | Tracking Supply | ZL88701/2 | ZL880 |
| VP_DEV_886_SERIES | Auto-Battery Switch (ABS) Supply | ZL88601/2 | |
| VP_DEV_887_SERIES | Shared Tracking Supply | ZL88801 | |
| VP_DEV_886_SERIES | Shared Buckboost ABS Supply | | |
| VP_DEV_887_SERIES | Tracking Supply | Le9672, Le9661, Le9652, Le9651, Le9641 | miSLIC |
| VP_DEV_886_SERIES | Shared Buckboost ABS Supply | Le9662, Le9642 | |
| VP_DEV_887_SERIES | Shared Tracking Supply | Le9662 | |

**Important:** When using the Shared Buckboost ABS design, BOTH lines contexts must be initialized and linked to the device context when calling any line specific VP-API-II function.

The ZL880/miSLIC configurations support the termination types listed in Table 1–4.

**Table 1–4 Supported ZL880/miSLIC Termination Types**

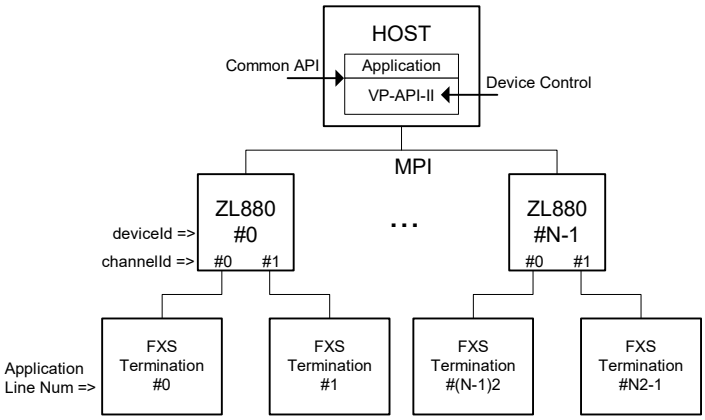
| VpTermType | Description |
|---|---|
| VP_TERM_FXS_GENERIC | Generic FXS termination |
| VP_TERM_FXS_LOW_PWR | Similar to Generic FXS. Uses a high resistive (i.e., “weak”) voltage feed mode when set to VP_LINE_STANDBY. This is done to achieve the lowest amount of power while being able to monitor hook activity. |

**Important: Characteristics of the ZL880/miSLIC VP-API-II**

- An interrupt from the silicon does not always cause the VP-API-II to generate an event. Conversely, masking a VP-API-II event does not always result in masking of the corresponding silicon level interrupt (i.e., the silicon interrupt that is used to generate the VP-API-II event).
- The silicon contains very little memory. In order to accommodate a wide variety of Application architectures while being memory friendly, the VP-API-II requires that the Application maintain memory being used by the VP-API-II and provides functions to meet any memory architecture design. Also, unneeded features of the VP-API-II may be compiled out using the macros discussed in *Code Size Management*, on page 59.

## 1.7.1 MPI Mode Hardware System Architecture

**Figure 1–2 MPI System Architecture Using ZL880 Devices (MPI not supported by miSLIC)**




### MPI System Limitations

MPI System limitations are only the amount of MPI traffic that can be passed within the system tick time (if using polling architecture) or 20ms (if using interrupt based architecture). In general, the following guidelines should be met:

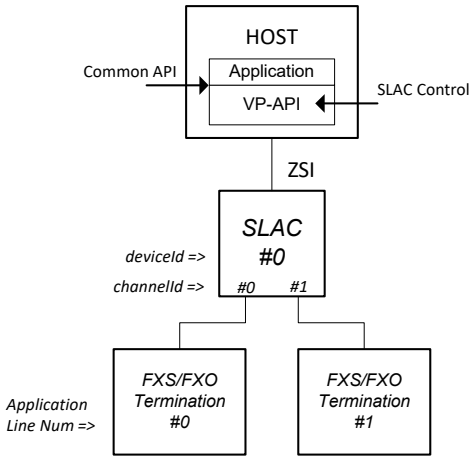
- Maximum number of lines per MPI BUS: 12

Systems that comply with the maximum number of lines can use all available VP-API-II features and meet system level requirements.

**Important:** The number of devices allowed on the MPI bus may decrease if other device types are using the same MPI bus. Customers intending to use multiple device types or more than 8 devices on the same MPI bus should contact Microsemi Customer Support for recommendations.

## 1.7.2 ZSI Mode Hardware System Architecture

**Figure 1–3 ZSI System Architecture Using miSLIC or ZL880 Devices**




### ZSI System Limitations

- There MUST be only 1 ZL880 or miSLIC device on the same ZSI bus. This is due to the ZSI interface definition itself which does not contain a unique CS.

All MPI and ZSI communication to the device is done through the HAL function **`VpMpiCmd()`**. Selecting the specific device in **`VpMpiCmd()`** is done using the `deviceId`. The `deviceId` is provided by the Application when creating the device object using **`VpMakeDeviceObject()`**. Since `deviceId` is a user defined type of `VpDeviceIdType`, the Application can define `deviceId` to contain any data necessary in order to uniquely identify a particular device in **`VpMpiCmd()`**. Refer to Chapter 12 for additional **`VpMpiCmd()`** requirements for the devices covered by this document.

## 1.8 VP-API-II FUNCTION SUMMARY

This section provides a brief overview of each of the VP-API-II functions supported by the ZL880/miSLIC VP-API-II.

### 1.8.1 System Configuration

The System Configuration functions manage device objects, line objects, device contexts, and line contexts. The concept of objects and contexts is discussed next. The functions that support System Configuration are discussed in Section 1.8.1.3.

#### 1.8.1.1 Device and Line Objects

The VP-API-II uses the concept of *device objects* and *line objects* to manage run-time support for different types of VTDs and line terminations. The rules for device and line objects in any given system are:

- There can be only one instance of a device object for each device in the system. The device object represents a physical device controlled by the VP-API-II. The ZL880/miSLIC device object type is **`Vp886DeviceObjectType`**.
- There can be only one instance of a line object for each line in the system. The line object represents a physical line controlled by the VP-API-II. The ZL880/miSLIC line object type is **`Vp886LineObjectType`**.

The VP-API-II is memory efficient. The VP-API-II can be configured at compile-time to include or exclude support for specific device types used in the system. These settings are in `vp_api_cfg.h`. To include support for ZL880 or miSLIC in the VP-API-II, the string `VP_CC_886_SERIES` must be set to `#define`. If set to: `#undef VP_CC_886_SERIES`, the types **`Vp886DeviceObjectType`** and **`Vp886LineObjectType`** will not exist in the VP-API-II.

Table Table 1–5 summarizes the compile-time setting required in `vp_api_cfg.h` and the ZL880/miSLIC specific device and line object types.

**Table 1–5 ZL880/miSLIC System Configuration**

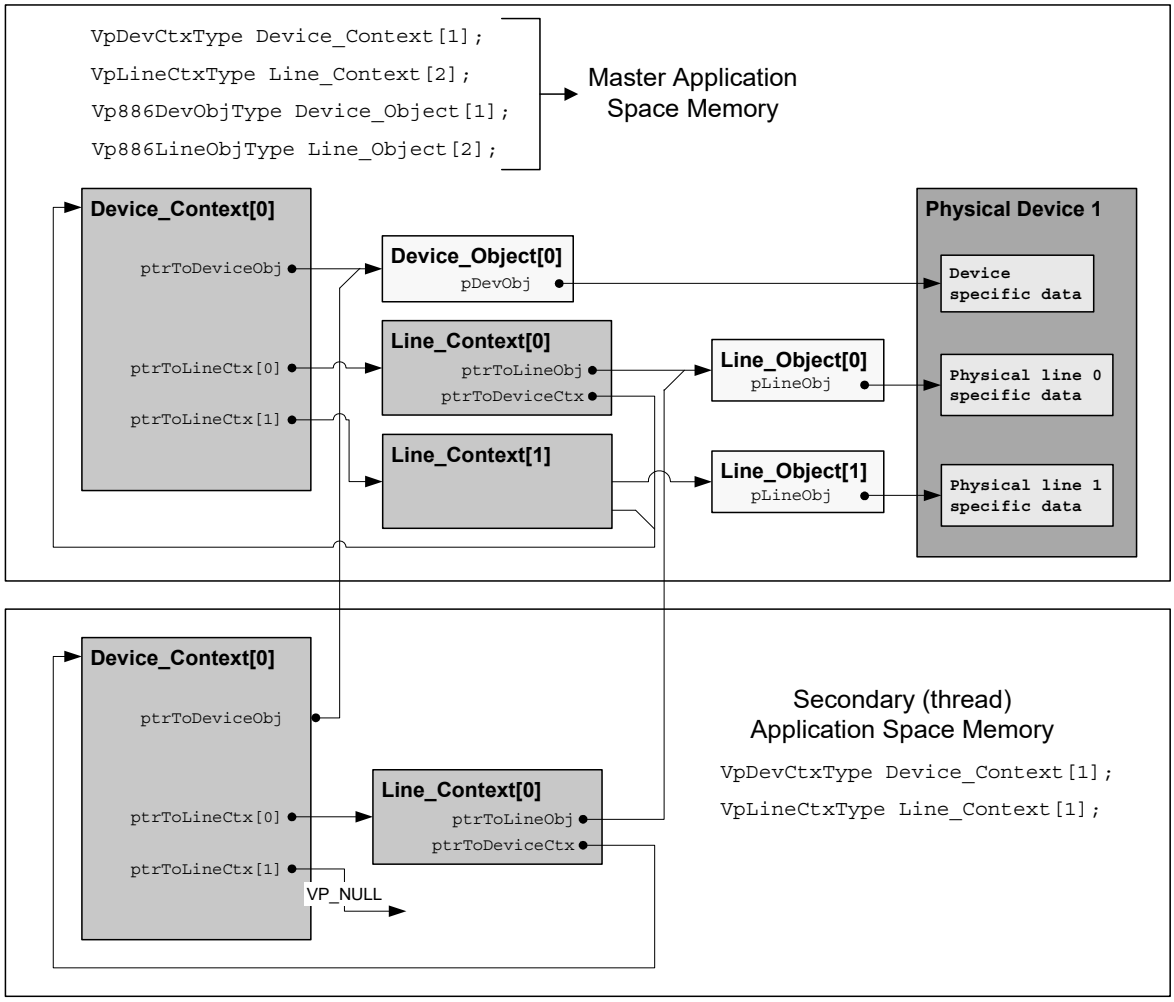
| VpDeviceType | Compile-Time Switch | Device Object Type | Line Object Type |
|---|---|---|---|
| VP_DEV_886_SERIES | VP_CC_886_SERIES | **Vp886DeviceObjectType** | **Vp886LineObjectType** |
| VP_DEV_887_SERIES | | | |

For convenience, the same compile-time switch, device object type and line object type are used for ZL880 and miSLIC devices.

#### 1.8.1.2 Device and Line Contexts

*Device contexts* are essentially handles to device objects. There can be many device contexts referring to a single device object. Similarly, *line contexts* are essentially handles to line objects. There can be many line contexts referring to a single line object. Figure 1–4 shows the relationship between the device and line contexts, objects and the hardware.

**Figure 1–4 Device and Line Contexts, Objects and Hardware Relationship**



### 1.8.1.3 System Configuration Functions

- **VpMakeDeviceObject()** – This function is required by all Applications. It initializes a device object and context used to access a physical device. It creates the link between the device object and device context in the “Master Application Space” shown in Figure 1–4.
- **VpMakeLineObject()** – This function is required by all Applications. It initializes a line object and context used to access a physical line. It creates the link between the line object and line context, and the between the line context and device context in the “Master Application Space” shown in Figure 1–4.
- **VpFreeLineCtx()** – Tells the API that the Application no longer needs a particular line context. This function is required only when dynamic memory methods are used.
- **VpMakeDeviceCtx()** – This function is required only in multi-threaded Applications. It creates a device context for an existing device object. It creates the link between the device object in the “Master Application Space” and the device context in the “Secondary Application Space” shown in Figure 1–4.
- **VpMakeLineCtx()** – This function is required only in multi-threaded Applications. It creates a line context for an existing line object. It creates the link between the line object in the “Master Application Space” and the line context in the “Secondary Application Space”, and between the device context and line context in the “Secondary Application Space” shown in Figure 1–4.
- **VpMapLineId()** – Associates a user defined line id (`VpLineIdType`) with the line.

## 1.8.2 Initialization

These functions initialize and configure the device and lines. Note that some of these functions are required in all Applications and must be called before a specific device or line can be accessed.

- **VpInitDevice()** – This function is required by all Applications. It initializes the device and all lines associated with the device using the profiles provided. When **VpInitDevice()** is complete it will generate the **VP_DEV_EVID_DEV_INIT_CMP** event.
- **VpInitLine()** – This function initializes the line using the profiles provided. Applications do not need to call this function if all lines are created (using **VpMakeLineObject()**) prior to calling **VpInitDevice()**. In that case, all lines will be initialized by **VpInitDevice()**. Line specific profiles (AC, DC and Ringing) may be changed without reinitializing the line using **VpConfigLine()**. If **VpInitLine()** is called by the Application, it indicates complete by generating the **VP_LINE_EVID_LINE_INIT_CMP** event.
- **VpConfigLine()** – Allows the Application to reconfigure the AC, DC or Ringing Parameters of the line without causing a line reset.
- **VpCal()** – Manages the Calibration Profile. If this function is called prior to **VpInitDevice()** it will not generate the **VP_EVID_CAL_CMP** event. That’s because devices of the VP-API-II cannot generate events prior to initialization. For further implementation details on **VpCal()** see Section 4.2 and Section 6.2.4.
- **VpCalLine()** – This function performs the run-time calibration procedures of the line necessary to meet Data Sheet Specifications. It may be avoided only if using **VpCal()** as discussed in Section 4.2.2.
- **VpInitProfile()** – This function copies the address of the profile passed into an entry in the device’s profile table. Note that all profiles passed to this function must remain in memory while in use by the VP-API-II. The Application must clear all previously set device profile table entries that no longer point to valid profile data by calling this function with a `VP_NULL` pointer for the corresponding profile type and profile table number.

The following Initialization functions are not available in VP-API-II Lite SW Package (OPN Le71SDKAPIL) and when `#undef CSLAC_SEQ_EN` is set in `vp_api_cfg.h`.

- **VpInitRing()** – Associates the Ringing Cadence and Caller ID Profiles with the line. The Caller ID Profile provided is only accessed if the Ringing Cadence Profile contains the “Start CID” or “Wait on CID..” elements.

- VpInitCid() – Initializes the CID message data buffer prior to Ringing associated Caller ID.
- VpInitMeter() – Configures the metering signal generator of the line.

### 1.8.3 Interrupt and Event Handling

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

- VpGetEvent() – Returns events corresponding to a device.
- VpFlushEvents() – Flushes all outstanding events.
- VpGetResults() – Reads the data associated with an event.

### 1.8.4 Control

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

- VpSetLineState() – Used to manage PLL Free Run mode (in case of PCLK Reset or Fault).
- VpSetLineState() – Sets a line to the requested state.
- VpSetRelayState() – Sets the line relay configuration.
- VpSetOption() – Sets various device and line specific options.
- VpSetLineTone() – Generates call progress tones on the line.

***Notes:***

*All line cadencing support including Tone cadencing is disabled when `#undef CSLAC_SEQ_EN` is set in `vp_api_cfg.h`. Applications with ONLY VP-API-II Lite source (OPN Le71SDKAPIL) must not set `#define CSLAC_SEQ_EN`, otherwise the link process will generate errors. The SW Package Le71SDKAPIL is not provided with the source necessary to avoid the linker errors if `CSLAC_SEQ_EN` is set to `#define`.*

- VpSetRelGain() – Sets the relative transmit or receive gain for a line.
- VpDeviceIoAccess() – Allows control over all GPIO pins at the same time.
- VpLineIoAccess() – Allows control over GPIO pins in per-line "groups".
- VpDtmfDigitDetected() – Reports a DTMF digit detected by an external resource. This function is generally used for implementing Type-II Caller ID, but can be called anytime by the Application to cause a VP_LINE_EVID_DTMF_DIG event to be generated.
- VpGenTimerCtrl() - Provides control of the general purpose timers.
- VpFreeRun() – Puts the device/lines into and takes it out of PLL Free Run mode.
- VpBatteryBackupMode() – Adjusts power supply timing parameters to maintain functionality in a battery backup mode.
- VpShutdownDevice() – Shuts down the device.

The following functions are not available in VP-API-II Lite (OPN Le71SDKAPIL) and when `#undef CSLAC_SEQ_EN` is set in `vp_api_cfg.h`.

- VpStartMeter() – Starts metering on the line.
- VpSendSignal() – Generates a DC signal on the line (e.g., Message Waiting Pulse, Forward Disconnect).
- VpSendCid() – Immediately starts a Caller ID sequence on the line. This function is most often used for Type II Caller ID (non-Ringing associated Caller ID).
- VpContinueCid() – Refreshes the Caller ID buffer for the line during message transmission.

### 1.8.5 Status and Query

These functions get information and events from the VTD.

- VpGetLoopCond() – Reads loop and battery conditions for the line.
- VpGetLineStatus() – Returns the state of a particular status for one line.

- `VpGetDeviceStatus()` – Returns the state of a particular status for all lines of the device.
- `VpGetLineInfo()` – Retrieves line-specific information from a device or line context.
- `VpGetDeviceInfo()` – Retrieves device-specific information from a device or line context.
- `VpGetOption()` – Returns the current setting of an option.
- `VpGetOptionImmediate()` – Same as `VpGetOption()` but doesn't require event processing to retrieve results. Results are provided with function call.
- `VpGetLineState()` – Reads the current line state.
- `VpQueryImmediate()` – Similar to `VpGetLineStatus()` but provides information that is non-boolean in nature about a particular line.

### 1.8.6 System Services

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

- `VpSysDebugPrintf()` – Print mechanism used by VP-API-II Debug features.
- `VpSysEnterCritical() and VpSysExitCritical()` – Blocks/Opens entry into a critical section of VP-API-II code or HAL access through a user-defined method. These functions are only required in multi-threaded Applications.
- `VpSysDtmfDetEnable() and VpSysDtmfDetDisable()` – These functions are used by the VP-API-II to control a DTMF digit decoding resource that is outside the scope of the VP-API-II. This is used for Type-II Caller ID implementation.
- `VpSysWait()` – Implements a software delay. Only needed if `VP_DEVICE_OPTION_ID_RING_PHASE_SYNC` is enabled on a device with revCode < 7.

### 1.8.7 Hardware Abstraction Layer

The Hardware Abstraction Layer (HAL) function contains the code used to directly access the ZL880/miSLIC silicon. This function is platform-specific and must be implemented by the user.

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

### 1.8.8 Debug Functions

The following functions are needed only during debug and should not be used without assistance from Microsemi Customer Support:

- `VpRegisterDump()` – This function generates output data of silicon register content.
- `VpObjectDump()` – This function generates output data of the line or device object.
- `VP_PRINT_DEVICE_ID()` – This macro prints the `deviceId` associated with the device.
- `VP_PRINT_LINE_ID()` – This macro prints the `lineId` associated with the line.

## 1.9 FUNCTION AVAILABILITY SUMMARY

**Figure 1–5 Function Summary by VP-API-II SW Package**

| Function Name | Events Generated | VP-API-II (Le71SK0002) | VP-API-II Lite (Le71SDKAPIL) |
|---|---|---|---|
| **System Configuration Functions** | | | |
| VpMakeDeviceObject() | None | ✓ | ✓ |
| VpMakeLineObject() | None | ✓ | ✓ |
| VpMakeDeviceCtx() | None | ✓ | ✓ |
| VpMakeLineCtx() | None | ✓ | ✓ |
| VpFreeLineCtx() | None | ✓ | ✓ |
| VpMapLineId() | None | ✓ | ✓ |
| **Initialization Functions** | | | |
| VpInitDevice() | VP_DEV_EVID_DEV_INIT_CMP | ✓ | ✓ |
| VpInitLine() | VP_LINE_EVID_LINE_INIT_CMP | ✓ | ✓ |
| VpConfigLine() | None | ✓ | ✓ |
| VpCal() | VP_EVID_CAL_CMP | ✓ | ✓ |
| VpCalLine() | VP_EVID_CAL_CMP | ✓ | ✓ |
| VpInitRing() | None | ✓ | |
| VpInitCid() | None | ✓ | |
| VpInitMeter() | None | ✓ | |
| VpInitProfile() | None | ✓ | ✓ |
| **Interrupt and Event Handling Functions** | | | |
| VpFlushEvents() | None | ✓ | ✓ |
| VpGetEvent() | None | ✓ | ✓ |
| VpGetResults() | None | ✓ | ✓ |
| **Control Functions** | | | |
| VpSetLineState() | Depending on Line State:<br>VP_LINE_EVID_RING_CAD<br>VP_LINE_EVID_CID_DATA | ✓ | ✓ |
| VpSetRelayState() | None | ✓ | ✓ |
| VpSetOption() | None | ✓ | ✓ |
| VpSetLineTone() | If using a Tone Cadence:<br>VP_LINE_EVID_TONE_CAD | ✓ | Tone Cadence Not Supported |
| VpSetRelGain() | VP_LINE_EVID_GAIN_CMP | ✓ | ✓ |
| VpSendSignal() | VP_LINE_EVID_SIGNAL_CMP | ✓ | |
| VpSendCid() | VP_LINE_EVID_CID_DATA | ✓ | |

| Function Name | Events Generated | VP-API-II (Le71SK0002) | VP-API-II Lite (Le71SDKAPIL) |
|---|---|---|---|
| VpContinueCid() | VP_LINE_EVID_CID_DATA | ✓ | |
| VpStartMeter() | VP_LINE_EVID_MTR_CMP or VP_LINE_EVID_MTR_ABORT | ✓ | |
| VpDtmfDigitDetected() | VP_LINE_EVID_DTMF_DIG | ✓ | ✓ |
| VpDeviceIoAccess() | VP_DEV_EVID_IO_ACCESS_CMP | ✓ | ✓ |
| VpLineIoAccess() | VP_LINE_EVID_LINE_IO_RD_CMP | ✓ | ✓ |
| VpGenTimerCtrl() | VP_LINE_EVID_GEN_TIMER | ✓ | ✓ |
| VpFreeRun() | None | ✓ | ✓ |
| VpBatteryBackupMode() | None | ✓ | ✓ |
| VpShutdownDevice() | None | ✓ | ✓ |
| **Status and Query Functions** | | | |
| VpGetLoopCond() | VP_LINE_EVID_RD_LOOP | ✓ | ✓ |
| VpGetLineStatus() | None | ✓ | ✓ |
| VpGetDeviceStatus() | None | ✓ | ✓ |
| VpGetLineInfo() | None | ✓ | ✓ |
| VpGetDeviceInfo() | None | ✓ | ✓ |
| VpGetOption() | VP_LINE_EVID_RD_OPTION | ✓ | ✓ |
| VpGetOptionImmediate() | None | ✓ | ✓ |
| VpGetLineState() | None | ✓ | ✓ |
| VpQueryImmediate() | None | ✓ | ✓ |
| **System Services Functions** | | | |
| VpSysDebugPrintf() | None | Required for debug only. Implemented by user. | Required for debug only. Implemented by user. |
| VpSysEnterCritical() and VpSysExitCritical() | None | Implemented by user. | Implemented by user. |
| VpSysDtmfDetEnable() and VpSysDtmfDetDisable() | None | Implemented by user. | Implemented by user. |
| **Hardware Abstraction Layer (HAL) Functions** | | | |
| VpMpiCmd() | None | Required. Implemented by user. | Required. Implemented by user. |
| **Debug Functions (required only for debug purposes)** | | | |
| VpRegisterDump() | None | ✓ | ✓ |
| VpObjectDump() | None | ✓ | ✓ |

| Function Name | Events Generated | VP-API-II (Le71SK0002) | VP-API-II Lite (Le71SDKAPIL) |
|---|---|---|---|
| VP_PRINT_DEVICE_ID() | None | Required for debug only. Implemented by user. | Required for debug only. Implemented by user. |
| VP_PRINT_LINE_ID() | None | Required for debug only. Implemented by user. | Required for debug only. Implemented by user. |

## 1.10 BASIC VP-API-II DATA TYPES

Table 1–6 lists the basic data types used by the VP-API-II. They are defined in `vp_api_types.h` file and are platform specific. These types must be modified as needed for the target platform.

**Table 1–6 Basic VP-API-II Data Types**

| Type | Description |
|---|---|
| `VpDeviceIdType` | Application-dependent device ID, user defined type. |
| `VpLineIdType` | Application-dependent line ID, user defined type. |
| `bool` | Boolean variable assigned `TRUE` or `FALSE`.<br><br>**NOTE:** TRUE must be assigned a non-zero value, and FALSE assigned 0 to be compatible with the VP-API-II. Throughout the code exists: `(if (value))` where: `bool value = TRUE;` is expected to pass this condition. |
| `uint8` | 8-bit unsigned integer. |
| `uint16` | 16-bit unsigned integer. |
| `uint32` | 32-bit unsigned integer. |
| `int8` | 8-bit signed integer. |
| `int16` | 16-bit signed integer. |
| `int32` | 32-bit signed integer. |
| `uint8p` | Pointer to 8-bit unsigned integer. |
| `uint16p` | Pointer to 16-bit unsigned integer. |
| `uint32p` | Pointer to 32-bit unsigned integer. |
| `VpProfilePtrType` | Pointer to profile data.<br><br>**NOTE:** This type must be defined as "`const uint8 *`" to be compatible with the ZL880/miSLIC VP-API-II |

## 1.11 VP-API-II FUNCTION RETURN TYPE

Most VP-API-II functions return a result code indicating whether the function executed successfully, and if not, what type of error occurred. The enumeration type `VpStatusType` is defined for this purpose. The `VpStatusType` codes supported by the VP-API-II are listed in Table 1–7.

**Table 1–7 VP-API-II Return Codes (`VpStatusType`)**

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

## 1.12 VP-API-II SOURCE VERSION NUMBER

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

```
#define VP_API_VERSION_TAG (0x021600)
```

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

The example shown is for Major release (0x02) 02, Minor release 22 (0x16), Revision (0x00) 0.

- Major Number: Interface level compatibility. Always = 02 for the VP-API-II.
- Minor Number: Functional change or addition. The following are modifications that warrant a "Minor" number increase:
  - Addition of a device or Termination Type

  - Addition of Options or Events the Application should be made aware of and possibly use. Addition or Options or Events that are considered features and not expected to be used by all Applications will generally warrant only a Revision Number change.
  - Backward compatibility is ensured in a Minor Revision change, excluding new functionality.
- Revision Number: Bug fix or minor Option or Event addition, or minor change to existing Event (e.g., adding information to `eventData` that was previously ignored). The following are modifications that warrant a Revision number increase:
  - Bug Fix requiring "extensive" changes to the source. If a bug change is made that affects one line (or very few lines), such changes are released as a "Patch" level release.
  - Minor Option or Event change that is not necessary for the Application but could be useful under certain circumstances.
  - Backward compatibility (excluding bug fixes) is ensured with Revision changes.
- Patch Release: Patch releases are indicated by a lower case "p" and 4th number (e.g., **p**2.20.0.**1** is patch release '1' from Production Release P2.20.0). These are very minor releases, generally consisting of a few modified lines of code and will be released as part of the VP-API-II (i.e., not all files are provided).
  - Patch releases MUST be applied to specific production releases as indicated by the first 3 digits of the patch release.
  - Multiple patch releases must be applied in order (e.g., patch releases p2.20.0.**1** and p2.20.0.**2** must be applied in order '1' and then '2').
  - Patch releases are identified by the source file revision number in the source header. These revision numbers correspond to specific releases/tags in Microsemi's SVN repository. Customers unsure if specific patches are applied should contact Microsemi field and customer support.

## 1.13 TECHNICAL SUPPORT

For technical support, logon to https://www.microsemi.com/voice-line-circuits then log into your MyMicrosemi account and follow the link to the "Product Issue Tracker".

CHAPTER

# 2 PROFILES

## 2.1 OVERVIEW

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

## 2.2 PROFILE TYPES

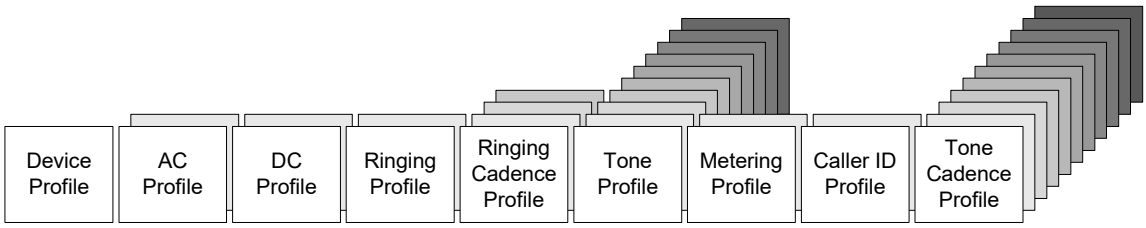
**Table 2–1 VP-API-II Profile Types**

| Profile Type | Description |
|---|---|
| Device | Contains values required to initialize the device. |
| AC | 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 | 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 | Configures the ringing signal parameters such as waveform type, frequency, amplitude, and DC offset. |
| Metering | Contains metering signal parameters such as pulse type, current limits, voltage limits and frequency (if 12Khz/16Khz metering signal). |
| Tone | Specifies tone frequency and level for up to 4 simultaneous tones (number of tones depending on device). |
| Ringing Cadence | Defines the ringing on/of durations used when ringing the line. Also specifies when CID is generated for Ringing associated CID. |
| Tone Cadence | Defines the tone generator on/of durations used when sending tones on the line. Note that each tone specified in the tone profile may be turned on/off individually in a single tone cadence. |
| Caller ID | Defines the signal type and parameters (frequency and level) used when generating CID as well as the signaling protocol used hen performing a CID sequence. The CID profile is used for both Type I and Type II CID. |
| Calibration | Defines the device and line error correction values required to meet Data Sheet Specifications. Applications can either run calibration in the field deployed unit, or apply a previously captured Calibration Profile (one per line) to the system. Note that the Calibration Profile is the only Profile that is NOT generated by the Profile Wizard. |

## 2.3 PROFILE TABLES

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

**Figure 2–1 Profile Tables**




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

The Application refers to an individual profile within a profile table by passing a *profile table index* into VP-API-II functions. Profile table indices are simply C macros in the form of `VP_PTABLE_INDEXx` where $1 \leq x \leq 11$ . `VP_PTABLE_NULL` is a special value indicating that no profile argument is specified.

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

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

When passed by reference, many profiles need to be retained in Application memory while being accessed by the VP-API-II. The column “Retain in Application Space (Y/N?)” in Table 2–2 indicates whether the specified profile when passed by reference must be retained in Application memory (if ’Y’) or if the entire profile content is copied into the VP-API-II and/or ZL880/miSLIC silicon (if ’N’). Note that all profiles passed into the Profile Table and accessed by index must be retained in Application memory.

**Table 2–2 Profile Table Capacity**

| Profile Type | Number of Profiles | Retain in Application Space (Y/N?) |
|---|---|---|
| Device Profile | 1 | N |
| AC Profile | 2 | N |
| DC Profile | 2 | N |
| Ringing Profile | 2 | N |
| Ringing Cadence Profile | 4 | Y |
| Tone Profile | 10 | Y |
| Metering Profile | 2 | N |
| Caller ID Profile | 2 | Y |
| Tone Cadence Profile | 11 | Y |
| Calibration Profile | Not in Profile Table | N |

## 2.4 PROFILE FUNCTIONS

See Table 1–2, *Profile Type to Function Mapping*, on page 3 for a list of the functions that use Profiles (and which profiles they use).

Note the last function `VpInitProfile()` at the bottom of the table. The table points out that this function has limited use in ZL880/miSLIC Applications. Section 2.5 will discuss this further.

## 2.5 APPLICATION NOTE ABOUT ZL880/MISLIC PROFILE TABLES

`VpInitProfile()` has limited use in a ZL880/miSLIC Application. It is provided in the ZL880/miSLIC VP-API-II to be compatible with other device families supported by the VP-API-II. When loading a profile table entry by calling `VpInitProfile()` the VP-API-II simply saves the profile pointer passed into the device object. When the Application later calls a function using the profile table entry, the VP-API-II refers to the profile pointer stored in the device object. This means all profiles pointed to by the profile table entries **MUST BE** maintained by the Application while the profile is being accessed. Given this, there is no memory savings by using the profile tables in a ZL880/miSLIC Application and it should be just as easy to pass profiles into the VP-API-II by reference. When passed by reference, not all of the profiles need to be maintained in Application memory. See Table 2–2 for details.

CHAPTER

# 3 OPTIONS

## 3.1 OVERVIEW

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

| | |
|---|---|
| **DESCRIPTION** | This is a summary description of the option. |
| **DEFAULT** | This field contains the default setting for the option. |

This chapter discusses the individual option types that are accessed through the `VpSetOption()`, on page 107 and `VpGetOption()`, on page 125 functions.

## 3.2 OPTION SUMMARY

There are two types of VP-API-II Options (of type `VpOptionIdType`):

- Line Specific -> Start with `VP_OPTION_ID_`
  - If passed with a line context in `VpSetOption()` then affects only the line.
  - If passed with a device context in `VpSetOption()` then affects all lines of the device.
- Device Specific -> Start with `VP_DEVICE_OPTION_ID_`
  - If passed with a line context in `VpSetOption()` then returns `VP_STATUS_INVALID_ARG`..
  - If passed with a device context in `VpSetOption()` then affects the device.

Table 3–2 lists the Line and Device options supported by the ZL880/miSLIC VP-API-II. Each option is described in detail later in this chapter. Note that most options are set to a default value after `VpInitDevice()` and `VpInitDevice()`. If a default value is not specifically stated, the user should set it in the Application by calling `VpSetOption()`.

Almost all options can be retrieved using either `VpGetOption()` or `VpGetOptionImmediate()`, but there are some major differences between these two methods as described in

**Table 3–1 `VpGetOption()`/`VpGetOptionImmediate()` differences:**

| `VpGetOption()` | `VpGetOptionImmediate()` |
|---|---|
| Data is ready as indicated by `VpGetEvent()` (`event.hasResults` member = `TRUE`). | No events are generated. |
| Data is retrieved by calling `VpGetOption()`. | Data is retrieved when `VpGetOptionImmediate()` is called, provided in the "void *pResults" value passed. |

This means the Application can choose to use either `VpGetOption()` or `VpGetOptionImmediate()` for retrieving option values. Therefore, unless otherwise noted these function names will be used interchangeably throughout the remainder of this document. It will be assumed that given the content in Table 3–1 the reader will now how to retrieve Option data using either method.

**Table 3–2 ZL880/miSLIC Supported Options (`VpOptionIdType`)**

| Options | Device/Line | Page |
|---|---|---|
| VP_DEVICE_OPTION_ID_CRITICAL_FLT | device | 25 |
| VP_OPTION_ID_SWITCHER_CTRL | line | 26 |
| VP_OPTION_ID_DEBUG_SELECT | all | 26 |
| VP_OPTION_ID_PULSE_MODE | line | 26 |
| VP_DEVICE_OPTION_ID_PULSE | device | 27 |
| VP_DEVICE_OPTION_ID_PULSE2 | device | 29 |
| VP_OPTION_ID_TIMESLOT | line | 29 |
| VP_OPTION_ID_CODEC | line | 30 |
| VP_OPTION_ID_PCM_TXRX_CNTRL | line | 31 |
| VP_OPTION_ID_RING_CNTRL | line | 31 |
| VP_OPTION_ID_LOOPBACK | line | 32 |
| VP_OPTION_ID_LINE_STATE | line | 32 |
| VP_OPTION_ID_EVENT_MASK | line | 33 |
| VP_DEVICE_OPTION_ID_DEVICE_IO | device | 34 |
| VP_OPTION_ID_LINE_IO_CFG | line | 36 |
| VP_OPTION_ID_ABS_GAIN | line | 37 |
| VP_OPTION_ID_DCFEED_PARAMS | line | 39 |
| VP_OPTION_ID_RINGING_PARAMS | line | 40 |
| VP_OPTION_ID_GND_FLT_PROTECTION | line | 41 |
| VP_DEVICE_OPTION_ID_ADAPTIVE_RINGING | device | 44 |
| VP_OPTION_ID_DTMF_MODE | line | 47 |
| VP_OPTION_ID_HIGHPASS_FILTER | line | 48 |
| VP_OPTION_ID_DTMF_PARAMS | line | 48 |
| VP_DEVICE_OPTION_ID_FSYNC_RATE | device | 49 |
| VP_DEVICE_OPTION_ID_RING_PHASE_SYNC | device | 49 |
| VP_OPTION_ID_RINGTRIP_CONFIRM | line | 50 |
| VP_DEVICE_OPTION_ID_ACCESS_CTRL | device | 50 |
| VP_DEVICE_OPTION_ID_SPI_ERROR_CTRL | device | 51 |

## 3.3 OPTION DESCRIPTIONS

### 3.3.1 VP_DEVICE_OPTION_ID_CRITICAL_FLT

**DESCRIPTION**

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

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

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

***Notes:***

1. *The ZL880/miSLIC currently does not detect AC Faults and would therefore not set the line to* `VP_LINE_DISCONNECT` *for the setting:* `acFltDiscEn = TRUE`*. However, Applications should set the value of* `acFltDiscEn` *as desired for future compatibility.*
2. *The ZL880/miSLIC currently does not detect DC Faults and Ground Key simultaneously. The selection of how to treat ring-to-ground and ring-to-battery currents (whether DC Fault or Ground Key) is made in the ZL880/miSLIC DC Profile. Regardless of the DC Profile setting, Applications should set the value of* `dcFltDiscEn` *as desired for future compatibility (i.e., when a version of the VP-API-II is provided that will detect DC Faults and Ground Key simultaneously).*

**DEFAULT**

`VpOptionCriticalFltType::acFltDiscEn= TRUE;` (AC Faults not currently detected)
`VpOptionCriticalFltType::dcFltDiscEn= TRUE;` (Detection based on DC Profile)
`VpOptionCriticalFltType::thermFltDiscEn= TRUE;`

### 3.3.2 VP_OPTION_ID_SWITCHER_CTRL

**DESCRIPTION**

This option determines whether or not a line is automatically forced into `VP_LINE_DISABLED` state when a critical supply fault is detected. The supply critical fault option parameter is passed through the `bool` value shown below.

```
bool autoShutdownEn;
```

Supply faults that cause this condition will be reported by **`VP_DEV_EVID_BAT_FLT`**. Further details below.

For Tracker Configurations:
- Switchers are specific to the line.
- A line will be placed into `VP_LINE_DISABLED` if the value `autoShutdownEn = TRUE` for the line/switcher that generates a fault (SWY for Channel 0, SWZ for Channel 1).
- The line can be reinitialized using **`VpInitLine()`** or **`VpInitDevice()`**.

For ABS Configurations:
- Switchers are common across all lines of the device.
- If EITHER Switcher Y or Switcher Z generates a fault, and EITHER line has this option set `autoShutdownEn = TRUE`, BOTH lines will be set to `VP_LINE_DISABLED`.
- The line can be reinitialized using **`VpInitDevice()`**.

**DEFAULT**

```
bool autoShutdownEn = FALSE;
```

### 3.3.3 VP_OPTION_ID_DEBUG_SELECT

**DESCRIPTION**

This option enables and disables the debug capabilities of the VP-API-II. Across different products and VP-API-II releases, the output data may or may not be interpreted by the customer. Therefore, if enabling this option it will be necessary to provide a log capture mechanism that can provide the debug output data into Microsemi Issue Tracker (part of the Software Delivery System).

```
uint32 debugSelect;
```

This option can only be set using **`VpSetOption()`**. It cannot be retrieved using **`VpGetOption()`** as with other options.

See Chapter 13, on page 137 for list of debug features.

**DEFAULT**

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

### 3.3.4 VP_OPTION_ID_PULSE_MODE

**DESCRIPTION**

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

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

**DEFAULT**

`VP_OPTION_PULSE_DECODE_OFF`

## 3.3.5 VP_DEVICE_OPTION_ID_PULSE

**DESCRIPTION**

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

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

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

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

#ifdef VP_ENABLE_OFFHOOK_MIN
    uint16 offHookMin;        /* Minimum off-Hook time */
#endif

} VpOptionPulseType;
```

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

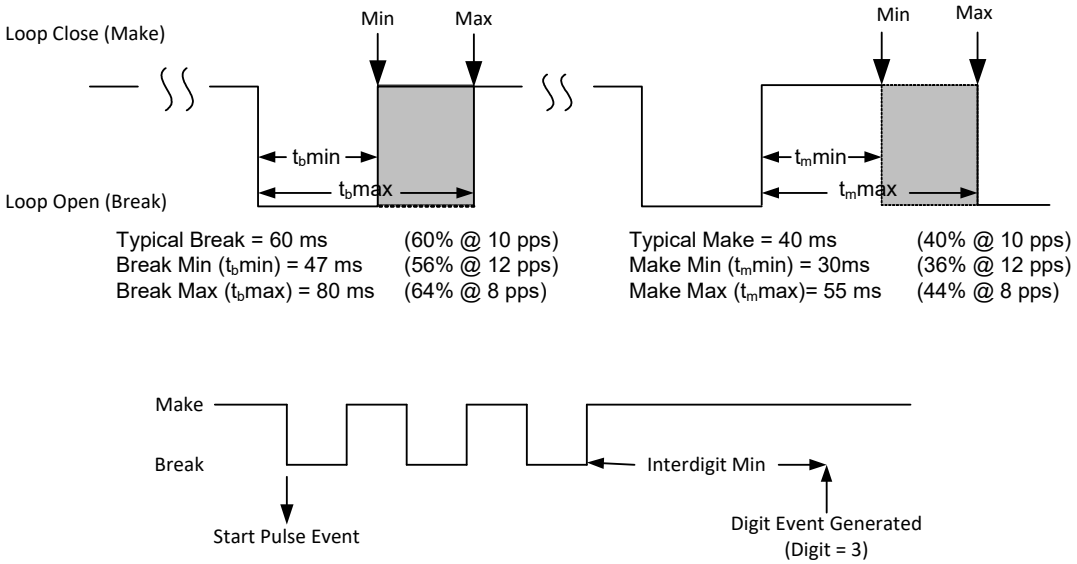
1. `breakMin < breakMax < flashMin < flashMax < onHookMin`
2. `makeMin < makeMax < interDigitMin`

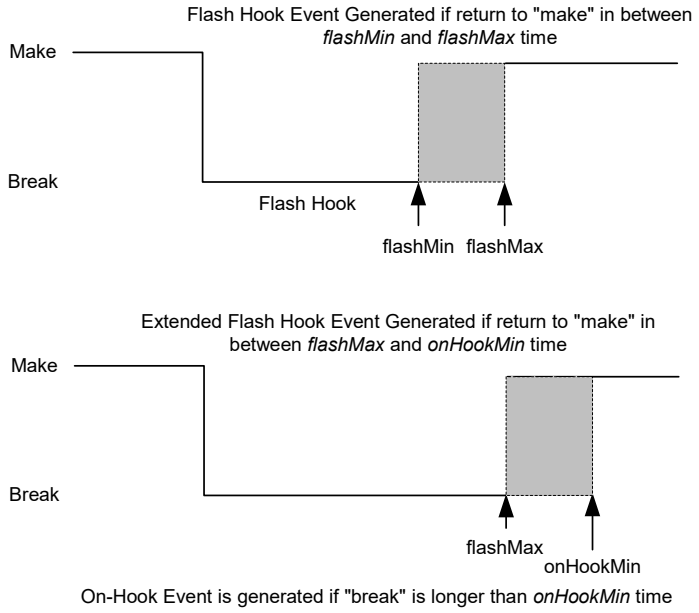
Figure 3–1 shows an example of a typical setting for this option. It also shows the timing relationship between the dialed digit and the events generated.

***onHookMin***

*Parameter* `onHookMin` *is compiled out of the VP-API-II by default. To enable* `onHookMin`*, the value* `EXTENDED_FLASH_HOOK` *must be changed to* `#define` *in file* `vp_api_cfg.h`*. If compiled in the behavior of the VP-API-II will be as shown in* Figure 3–1.

**Figure 3–1 Typical VP Option Pulse Timing Diagrams**

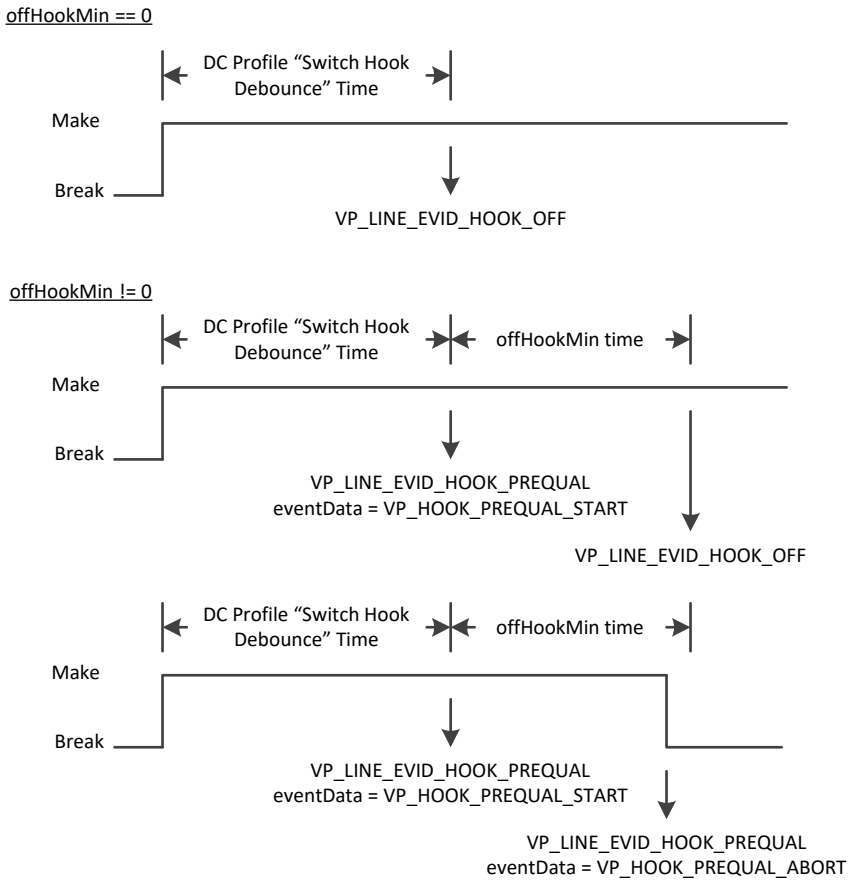





***offHookMin***

*Parameter* `offHookMin` *is compiled out of the VP-API-II by default. To enable* `offHookMin`*, the value* `VP_ENABLE_OFFHOOK_MIN` *must be changed to* `#define` *in file* `vp_api_cfg.h`*. If compiled in the behavior of the VP-API-II for* `offHookMin` *will be as shown in* Figure 3–2.

**Figure 3–2 `offHookMin` Behavior**



**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 */
VpOptionPulseType::offHookMin = 0; /* VP_LINE_EVID_HOOK_PREQUAL is disabled */
```

## 3.3.6 VP_DEVICE_OPTION_ID_PULSE2

**DESCRIPTION**

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

This option provides the ability to uniquely detect dial pulses with different set of `breakMin/breakMax` and `makeMin/makeMax` requirements from those set using **VP_DEVICE_OPTION_ID_PULSE**. When using this option, the values `interDigitMin`, `flashMin/flashMax`, `onHookMin` and `offHookMin` must be identical to those set in **VP_DEVICE_OPTION_ID_PULSE**.

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

**DEFAULT**

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

## 3.3.7 VP_OPTION_ID_TIMESLOT

**DESCRIPTION**

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

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

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

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

***Notes:***

The Application should assign timeslots before setting the line to a state that will enable transmission on the PCM highway. *See* **VpSetLineState()**, on page 104 *for more information on which line states activate the PCM highway.*

**DEFAULT**

Channel 0: tx = rx = 0; Channel 1: tx = rx = 1;

## 3.3.8 VP_OPTION_ID_CODEC

**DESCRIPTION**

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

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

***Timeslot Management For CODEC Modes:***

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

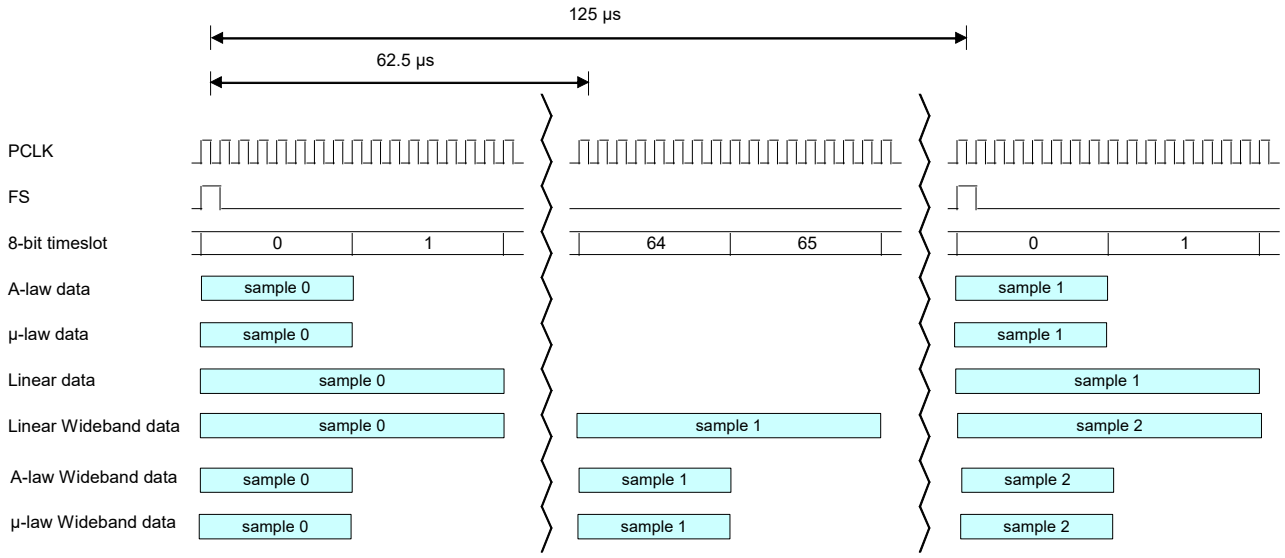
`VP_OPTION_ALAW` *and* `VP_OPTION_MLAW`*: When using 8-bit compressed mode only a single timeslot is required.*

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

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

`VP_OPTION_ALAW_WIDEBAND` *and* `VP_OPTION_MLAW_WIDEBAND`*: When using 8-bit compressed Wideband Mode (sampled at 16kHz - twice the frame rate), 2 timeslots are required. The first corresponds to the first compressed data sample in the frame, the second corresponds to the second compressed data sample in the frame. Since the sample rate is 2x the frame rate, the samples are separated by 1/2 a frame.*

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




**DEFAULT**

`VP_OPTION_ALAW`

## 3.3.9 VP_OPTION_ID_PCM_TXRX_CNTRL

**DESCRIPTION**

This line-specific option enables or disables the PCM transmit and receive paths. It is used to override the PCM Path settings described in **VpSetLineState()** for each state. The option setting is defined by the enumeration VpOptionPcmTxRxCntrlType below:

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

VP_OPTION_PCM_BOTH, VP_OPTION_PCM_RX_ONLY and VP_OPTION_PCM_TX_ONLY applies only to line states: VP_LINE_TALK, VP_LINE_TALK_POLREV, VP_LINE_OHT and VP_LINE_OHT_POLREV.

VP_OPTION_PCM_ALWAYS_ON and VP_OPTION_PCM_OFF applies to all line states.

When set to VP_OPTION_PCM_BOTH the PCM settings will be exactly as specified in **VpSetLineState()**.

**DEFAULT**

VP_OPTION_PCM_BOTH

## 3.3.10 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;

typedef enum {
    VP_OPTION_ZC_M4B,
    VP_OPTION_ZC_B4M,
    VP_OPTION_ZC_NONE
} VpOptionZeroCrossType;
```

The zeroCross parameter controls whether ringing is disabled near a zero cross point or not. Since Ringing in the ZL880/miSLIC silicon is internal, settings VP_OPTION_ZC_M4B and VP_OPTION_ZC_B4M do the same thing - disable ringing near a zero cross point. The setting VP_OPTION_ZC_NONE will disable Ringing immediately.

The ringExitDbncDur variable sets the minimum time to ignore hook activity when exiting ringing, specified in units of 125us. The ring-exit debounce period starts when the line state is changed by the Application not (necessarily) when ringing is removed from the line, which may be up to 1/2 period from the time the Application stops ringing. This debounce time helps filter false hook events during transitions from the *ringing-on* state to the *ringing-off* state caused by the physical characteristics of the line.

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

***Notes:***
*The default ring trip exit state (VP_LINE_TALK) will enable the PCM Highway. This may not be desirable in Applications that do not control the PCM Highway another way. Depending on the system design, if the PCM Highway is opened immediately upon ring trip, the subscriber may continue to hear ringing. If this is possible, the desired ring trip state is VP_LINE_ACTIVE. VP_LINE_TALK should then be set only when a clean PCM highway connection is available, and ringing has stopped being applied at the far end.*

**DEFAULT**

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

### 3.3.11 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_TIMESLOT_WITH_DAC /* Loopback with DAC enabled */
```

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

`VP_OPTION_LB_TIMESLOT_WITH_DAC` performs a timeslot loopback while also allowing signals in the receive PCM path to play on tip and ring.

***Notes:***

*When performing a timeslot loopback test (i.e., set* `VP_OPTION_LB_TIMESLOT` *mode and send data into the PCM receive path while monitoring the PCM transmit path) the line must be in a state where the A/D and D/A converters, and the PCM Highway is enabled. For states that meet these conditions refer to* **`VpSetLineState()`**.

**DEFAULT**

`VP_OPTION_LB_OFF`

### 3.3.12 VP_OPTION_ID_LINE_STATE

**DESCRIPTION**

This option defines the polarity reversal method (abrupt or smooth) used when changing feed polarity. These settings are passed through the `VpOptionLineStateType` structure shown below.

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

The smooth polarity reversal setting (`battRev = FALSE`) on the ZL880/miSLIC silicon is not slow enough to be used in most Polarity Reversal Metering Systems. This option is provided as a way to minimize transient noise.

When the `battRev` variable is set to `FALSE`, it forces a smooth voltage change when the line state is changed from any feed state to any other feed state of opposite polarity.

The ZL880/miSLIC device only supports `bat` = `VP_OPTION_BAT_AUTO`. For future compatibility however, this value is checked by the VP-API-II and will return an error code if `bat` != `VP_OPTION_BAT_AUTO`. Applications should make sure `bat` = `VP_OPTION_BAT_AUTO` when setting this option for the ZL880/miSLIC VP-API-II.

**DEFAULT**

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

## 3.3.13 VP_OPTION_ID_EVENT_MASK

**DESCRIPTION**

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

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

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

The composite masks for each event category are created by *bitwise OR'ing* the event ID constants of the individual events within that category. When building the composite event mask, the Application should first perform a `memset()` on the instantiated `VpOptionEventMaskType` to `0xFF` as shown below:

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

Applications without convenient access to the standard `memset()` function can instead use the VP-API-II provided `VpMemSet()` function as follows:

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

This will ensure future compatibility. Then, the Application should unmask all desired events using the event ID constants belonging to that event category. The following enumeration types define the event ID constants in the source code, which includes events and event categories that the ZL880/miSLIC VP-API-II will not generate:

- `VpFaultEventType`
- `VpSignalingEventType`
- `VpResponseEventType`
- `VpTestEventType`
- `VpProcessEventType`
- `VpFxoEventType`

The event types supported by the ZL880/miSLIC VP-API-II are described in Chapter 8. Some event are device-specific, and some events are line-specific.

Note that calling **`VpSetOption()`** with this option affects event masks for the device and may affect line event masks for all lines of the device. Setting *line-specific* event masks will contain *device-specific* event masks since these are part of the same struct provided to **`VpSetOption()`**. The last call to **`VpSetOption()`** with this option will determine the *device-specific* event masks. To avoid confusion, the Application should always set *device-specific* event masks for individual lines to the same value across all lines of the device. 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 107 for more information on how device-specific and line-specific options are applied based on the values of `pDevCtx` and `pLineCtx`.

***Notes:***

1. *Events that are non-maskable will always read back '0' even when set to '1'.*
2. *Event bit locations that are unspecified or not properly handled by the Application should always be set to '1'. This will prevent future VP-API-II updates from creating compatibility issues with an existing Application.*

**DEFAULT**

All events are masked except for non-maskable events.

## 3.3.14 VP_DEVICE_OPTION_ID_DEVICE_IO

**DESCRIPTION**

This option configures the GPIO pins of the device. It provides the exact same functionality as `VP_OPTION_ID_LINE_IO_CFG` except the I/O pins are accessed at the device level (i.e., all I/O pins accessed at once) rather than on a per-line basis (i.e., requires two function calls to access all I/O pins).

**Customers may find option `VP_OPTION_ID_LINE_IO_CFG` easier to use than this option if the GPIO pins in the Application are logically associated with lines.**

***I/O2 Voltage Monitor Note:***

*In the Device Profile Dialog of Profile Wizard P2.3.0 and later, I/O2 can be configured as: {"Analog Voltage Sense Input", "Digital", or "Digital with Interrupt"}. In the "Analog Voltage Sense Input" mode, the option `VP_DEVICE_OPTION_ID_DEVICE_IO` will not affect the configuration of this pin and instead will return `VP_STATUS_DEDICATED_PINS`. All other pins being configured in the same call will be allowed.*

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

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

The bit-mask `directionPins_31_0` contains a bit for each I/O pin that determines whether a single pin is configured as an input (0) or output (1) using the `VpDeviceIoDirectionType enum` below:

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

The relationship of `directionPins_31_0` to silicon I/O pins is:

| $D_{31}$- $D_8$ | $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|---|
| x | x | x | x | x | $I/O2_1$ | $I/O1_1$ | $I/O2_0$ | $I/O1_0$ |

where:

x = ignored
I/O2n = ignored if input is configured as voltage input per the Device Profile

The bit-mask `outputTypePins_31_0` contains a bit for each I/O pin that determines whether a single output pin is configured as a CMOS output (`VP_OUTPUT_DRIVEN_PIN = 0`) or an open-drain output (`VP_OUTPUT_OPEN_PIN` = 1) using the `VpDeviceOutputPinType enum` below:

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

The relationship of `outputTypePins_31_0` to silicon I/O pins is:

| $D_{31}$- $D_8$ | $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|---|
| x | x | x | x | x | $I/O2_1$ | $I/O1_1$ | $I/O2_0$ | $I/O1_0$ |

where:

x = ignored
I/O2n = ignored if input is configured as voltage input per the Device Profile

Only pins configured as output (see `directionPins_31_0`) will be affected by `outputTypePins_31_0` setting.

**ZL880/miSLIC Application Examples using VP_DEVICE_OPTION_ID_DEVICE_IO:**

Example: Setting channel 0, I/O1 = Output-Driven, Channel 1, I/O2 = Output-Open, all other pins set to Input:

```
directionPins_31_0 = 9;
```

1 0 0 1
- ch0, I/O1 = Output
- ch0, I/O2 = Input
- ch1, I/O1 = Input
- ch1, I/O2 = Output

```
outputTypePins_31_0 = 8;
```

1 x x 0
- ch0, I/O1 = Driven
- ch0, I/O2 = Input (output type ignored)
- ch1, I/O1 = Input (output type ignored)
- ch1, I/O2 = Open

**DEFAULT**

$I/O1_{ch}$ = Input

$I/O2_{ch}$ = Per Device Profile [Analog Voltage Sense 'OR' Digital I/O]. If configured as 'Digital I/O', then Default = Input

## 3.3.15 VP_OPTION_ID_LINE_IO_CFG

**DESCRIPTION**

This option configures the GPIO pins of the device. It provides the exact same functionality as `VP_DEVICE_OPTION_ID_DEVICE_IO` except the I/O pins are accessed at the line level (i.e., requires two function calls to access all I/O pins) rather than on a per-device basis (i.e., one function call to access all I/O pins).

**Customers may find this option easier to use than `VP_DEVICE_OPTION_ID_DEVICE_IO` if the GPIO pins are logically associated with the lines in the Application.**

***I/O2 Voltage Monitor Note:***

*In the Device Profile Dialog of Profile Wizard, I/O2 can be configured as an Analog Voltage Sense Input. In this mode the option `VP_DEVICE_OPTION_ID_DEVICE_IO` will not affect the configuration of this pin, and instead will return `VP_STATUS_DEDICATED_PINS`. All other pins being configured in the same call will be allowed.*

The arguments to this option are passed through the **`VpOptionLineIoConfigType`** `struct` shown below.

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

The bit-mask `direction` contains a bit for each I/O pin that determines whether a single pin is configured as an input (0) or output (1) using the `VpDeviceIoDirectionType enum` below:

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

The ZL880/miSLIC I/O pins to `direction` mapping is:

| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| x | x | x | x | x | x | $I/O2_{ch}$ | $I/O1_{ch}$ |

where:

x = ignored

ch (subscript) = I/O channel number associated with the line context passed with this option.

The bit-mask `outputType` contains a bit for each I/O line that determines whether a single output pin is configured as a CMOS output (`VP_OUTPUT_DRIVEN_PIN = 0`) or an open-drain output (`VP_OUTPUT_OPEN_PIN` = 1) using the `VpDeviceOutputPinType enum` below:

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

The relationship of `outputType` to silicon I/O lines is:

| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| x | x | x | x | x | x | $I/O2_{ch}$ | $I/O1_{ch}$ |

where:

x = ignored

ch (subscript) = I/O channel number associated with the line context passed with this option.

$I/O2_{ch}$ = ignored if input is configured as voltage input per the Device Profile. Only pins configured as output (see `direction`) will be affected by `outputType` setting.

**DEFAULT**

$I/O1_{ch}$ = Input

$I/O2_{ch}$ = Per Device Profile [Analog Voltage Sense 'OR' Digital I/O]. If configured as 'Digital I/O', then Default = Input

## 3.3.16 VP_OPTION_ID_ABS_GAIN

**DESCRIPTION**

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

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

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

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

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

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

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

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

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

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

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

### Supported Ranges and Step Sizes

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

**Table 3–3 ZL880/miSLIC Supported ABS Gain Ranges:**

| gain_AToD | gain_DToA | Step Size |
|---|---|---|
| -6dB to +6dB | -12dB to 0dB | 0.5dB |

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

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

### Using VpGetOption()

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

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

**Setting the Gain to Known Values**

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

gain.gain_AToD = 0    /* 0dB  */
gain.gain_DToA = -60  /* -6dB */
gain blocks have NOT been modified since last being set with this optionId.
The VP-API-II can determine the absolute gain if using Microsemi provided
AC Coefficients.
```

**Setting the Gain to Unknown Values**

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

gain.gain_AToD = VP_ABS_GAIN_UNKNOWN
gain.gain_DToA = VP_ABS_GAIN_UNKNOWN
gain blocks have been modified (with AC_600 Profile) since last being set
with this optionId. The VP-API-II cannot determine the absolute gain.
```

### Example gain_AToD and gain_DToA values:

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

***Notes:***

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

**DEFAULT** VP_ABS_GAIN_UNKNOWN

## 3.3.17 VP_OPTION_ID_DCFEED_PARAMS

**DESCRIPTION**

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

The DC Feed Parameters are passed through the **VpOptionDcFeedParamsType** struct shown below:

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

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

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

| **validMask bit Name** | **Field Affected** |
|---|---|
| VP_OPTION_CFG_VOC | voc |
| VP_OPTION_CFG_ILA | ila |
| VP_OPTION_CFG_HOOK_THRESHOLD | hookThreshold |
| VP_OPTION_CFG_LP_HOOK_THRESHOLD | lpHookThreshold |
| VP_OPTION_CFG_GKEY_THRESHOLD | gkeyThreshold |
| VP_OPTION_CFG_BATT_FLOOR | battFloor |

When using **VpGetOption()**, the validMask result tells which fields are valid for the particular line type (e.g. VP_OPTION_CFG_LP_HOOK_THRESHOLD will not be valid if using termination type VP_TERM_FXS_GENERIC).

**DEFAULT**

The VP-API-II contains a default DC profile (if one is not provided) that is set for:

```
voc = 45V, ila = 20mA, gKeyThreshold = 18mA
hookThreshold = 9mA (Tracker Configurations) or 11mA (ABS Configurations)
lpHookThreshold = 18V (Tracker Configurations) or 22V (ABS Configurations)
battFloor = -25V (Tracker configurations only)
```

## 3.3.18 VP_OPTION_ID_RINGING_PARAMS

**DESCRIPTION**

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

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

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

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

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

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

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

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

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

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

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

**DEFAULT**

The VP-API-II contains a default Ringing Profile (if one is not provided) that is set for:

```
25Hz (Sinewave), 45Vrms AC, 0V DC Bias,
Ring Trip Threshold = 21mA, Ringing Current Limit = 50mA
```

## 3.3.19 VP_OPTION_ID_GND_FLT_PROTECTION

**DESCRIPTION**

This option is used to control an Auto-Ground Fault monitoring feature of the VP-API-II. It is an alternative and safer way to manage the SLIC Line State during certain conditions that can cause Thermal Fault. It is configured by the values defined in structure **VpOptionGndFltProtType** as follows:

```
typedef struct VpOptionGndFltProtType {
    bool enable;
    uint16 confirmTime; /* 10ms units */
    uint16 pollTime;    /* 10ms units */
    uint16 pollNum;     /* Number of times to repeat Ground check */
} VpOptionGndFltProtType;
```

Details of these values is provided below and can be seen applied in Figure 3–3

**bool** enable;
This value either enables (if enable = 'TRUE') or disables (if enable = 'FALSE') the Auto-Ground Fault Protection Algorithm.

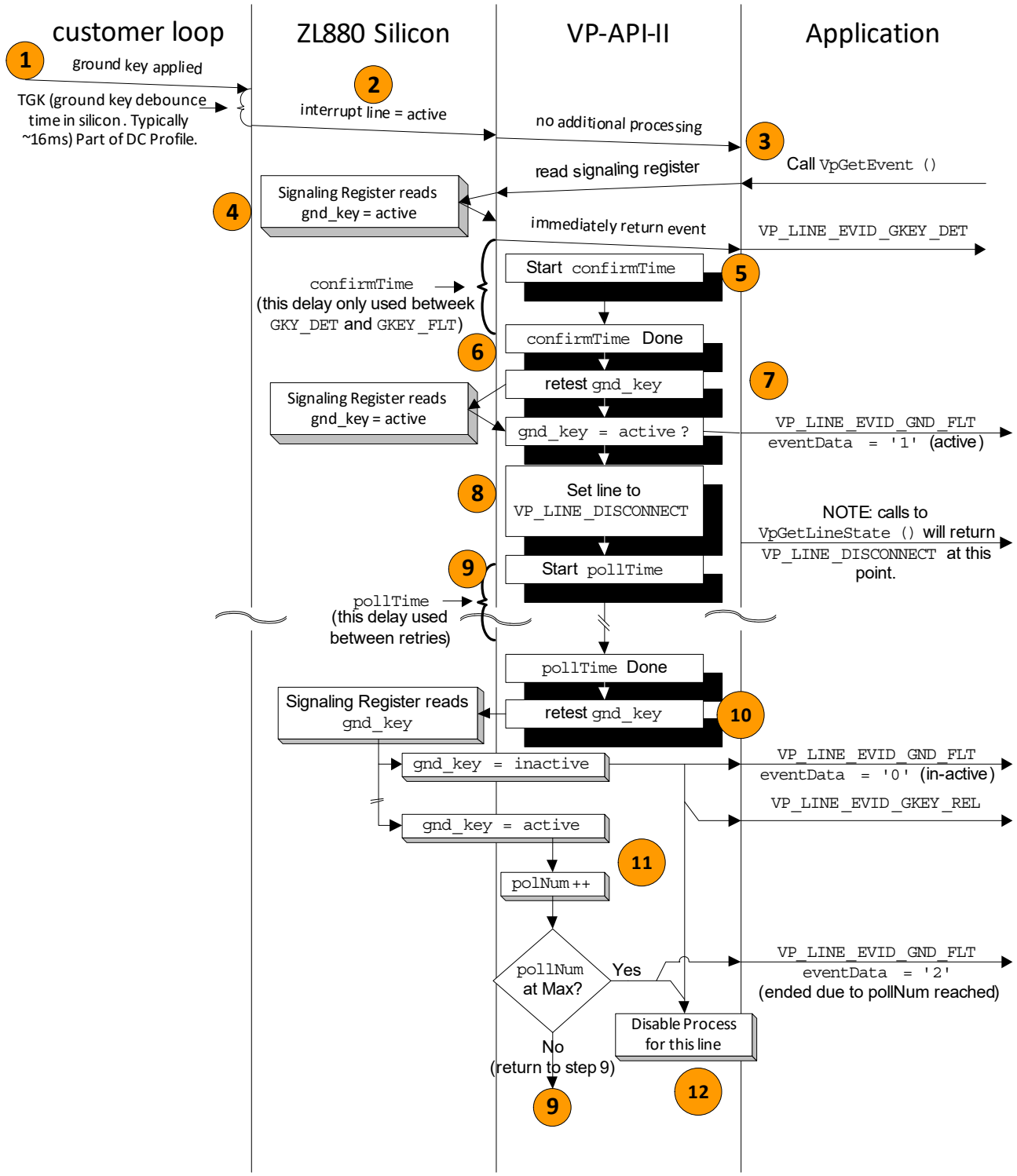
1. enable = 'FALSE': Auto-Ground Fault Protection is disabled, and in some cases the silicon will repeat the following sequence (when **VP_DEVICE_OPTION_ID_CRITICAL_FLT** for thermFltDiscEn set to 'FALSE'):
   a) Enter Thermal Fault condition.
   b) The SLIC will disable it's output drivers.
   c) This allows the SLIC to cool below thermal fault temperatures which will cause the SLIC to set it's output driver back to previous feed states.
   d) If the thermal fault that caused the condition in step #1 was not removed, this cycle will repeat.
2. enable = 'TRUE': Auto-Ground Fault Protection is enabled. The sequence is basically the same as the previous sequence, except instead of using a somewhat harmful Thermal Fault threshold, the cutoff threshold is ground key detected. Figure 3–3 shows the sequence that is applied for this condition.

**uint16** confirmTime;
This value defines the time between the initial indication of Ground Key Detect until a continuous Ground Key Applied will cause the event **VP_LINE_EVID_GND_FLT** to be generated. For POTS/FXS Applications where Ground Key detection is not allowed and therefore may not be processed, this value should be relatively low (e.g., < 100ms). Making this value too high will allow the silicon to instead run the "Thermal Fault->SLIC Disconnect De-Activate" loop sequence mentioned above.

**uint16** pollTime;
This value defines the time between the initial indication of Ground Key Fault as noted by the generation of event **VP_LINE_EVID_GND_FLT** to the time the Algorithm will retest for the Ground Key condition. Note that during this time, it is impossible for the Algorithm to tell if the Ground Key was removed since the SLIC is in a Disconnect Feed condition. On the other hand, execution of this Algorithm does require some amount of MPI traffic and may cause brief noise on the opposite line. These are just some of the considerations to be taken into account when defining this value.

**uint16** pollNum;
This value sets the limit on the number of tries this Algorithm will test the line for Ground Key/Fault to be removed. Note that this count starts after Ground Fault has been detected as shown in Figure 3–3. If pollNum = 0, this algorithm will repeat indefinitely.

**Figure 3–3 Auto-Ground Fault Protection**



**DEFAULT**

**Defaults for non-Shared Buckboost ABS Supply Configurations:**

```
enable = FALSE;      /* Fault behavior per step #1 under "bool enable" */
confirmTime = 100;   /* 10ms per steps = 1000ms (1 second) */
pollTime = 100;      /* 10ms per steps = 1000ms (1 second) */
pollNum = 0;         /* Infinite repeat steps 8 through 11 until
                        "gnd_key = inactive" */
```

***Notes:***

*For non-ABS Shared Supply Configurations, this algorithm is not run by default. Therefore, the values above are the recommended values for non-Shared Buckboost ABS configurations when* `.enable` *is set to "*`TRUE`*". Recommended values for Shared Buckboost ABS are listed below.*

**Defaults for Shared Buckboost ABS Supply configurations:**

```
enable = TRUE;       /* Fault behavior per step #2 under "bool enable" */
confirmTime = 0;     /* Fault is declared immediately after detection of
                        Ground Key using this configuration */
pollTime = 200;      /* 10ms per steps = 2000ms (2 seconds) */
pollNum = 0;         /* Infinite repeat steps 8 through 11 until
                        gnd_key = inactive */
```

## 3.3.20 VP_DEVICE_OPTION_ID_ADAPTIVE_RINGING

**DESCRIPTION**

This option is used to limit the output power provided by the supply. This feature is disabled by default using `#undef VP886_INCLUDE_ADAPTIVE_RINGING` in `vp_api_cfg.h`. To enable this feature, change this line to `#define VP886_INCLUDE_ADAPTIVE_RINGING`. Applications that do not use this feature should keep this value set to `#undef`.

This option is configured by the values in structure **`VpOptionAdaptiveRingingType`** as follows:

```
typedef struct VpOptionAdaptiveRingingType {
   uint32 validMask;  /* Selects parameters in this struct that will be
                         modified */
   uint8 power;          /* Sets the load power switch point */
   uint8 minVoltagePercent; /* Sets the minimum load ringing voltage */
   VpAdaptiveRingingModeType mode;  /* Sets Adaptive Ringing mode used */
} VpOptionAdaptiveRingingType;
```

Where:

```
typedef enum VpAdaptiveRingingModeType {
    VP_ADAPT_RING_SHARED_TRACKER, /* Used for Shared Tracker only */
    VP_ADAPT_RING_SHARED_BB_ABS, /* Used for Shared Buckboost ABS */
    VP_ADAPT_RING_SINGLE_BB_TRACKER /* 1-channel Buckboost tracker */
    VP_ADAPT_RING_FIXED_TRACKER /* For all fixed tracking supplies */
    VP_ADAPT_RING_FULL_TRACKER  /* For non-fixed tracking supplies */
} VpAdaptiveRingingModeType;
```

`uint32 validMask;`
Defines the parameters being changed in current function call. This is an 'OR' operation of the "`validMask` bit name" affecting the "Field Affected" described in Table 3–4 .

**Table 3–4 `VP_DEVICE_OPTION_ID_ADAPTIVE_RINGING` validBitMask names**

| validMask bit name | Value (see Notes) | Field Affected |
|---|---|---|
| VP_ADAPTIVE_RINGING_CFG_POWER | (0x0001L) | power |
| VP_ADAPTIVE_RINGING_CFG_MIN_V_PCT | (0x0002L) | minVoltagePercent |
| VP_ADAPTIVE_RINGING_CFG_MODE | (0x0004L) | mode |

***Notes:***

*The "Values" shown in Table 3–4 may differ between VP-API-II releases. Therefore, the Application must always use the names given in "validMask bit name" column to avoid breaking future compatibility.*

`uint8 power;`
This value is the power in the load where the algorithm will switch from high (nominal) voltage to low voltage (specified by minVoltagePercent). It is represented in 39 mW steps. Power in the load includes the loop and load impedance. For two-channel devices the power per line will be half of this value.

**The adaptive mode is disabled by** *setting* `power = 0xFF`.

`uint8 minVoltagePercent;`
This value sets the minimum ringing voltage the algorithm will apply when enabled under heavy REN load conditions. This is the voltage measured at the SA/SB (i.e., "Sense") inputs, degraded at the load by loop and load impedance.

Setting this value requires some computation because the user must determine a minimum voltage across the load that is acceptable. Then, determine what Ringing Open Circuit Voltage will meet that requirement, and finally determine the percent drop from (Ringing Open Circuit Voltage to the Load Voltage) that is specified in `minVoltagePercent`.

A simplified circuit model with the SLIC in Ringing State and steps required is shown in Figure 3–4.

**Figure 3–4 Ringing SLIC Model and `minVoltagePercent` Computation**

1a. Compute what $VRING_{VOC}$ must be to meet $V_{LOAD}$ requirements.

$$V_{LOAD} = \frac{VRING_{VOC} * Z_{LOAD}}{(Z_{LOAD} + 2 * ZSENSE_{OUT} + 200ohm)}$$

1b. Same equation, converted.

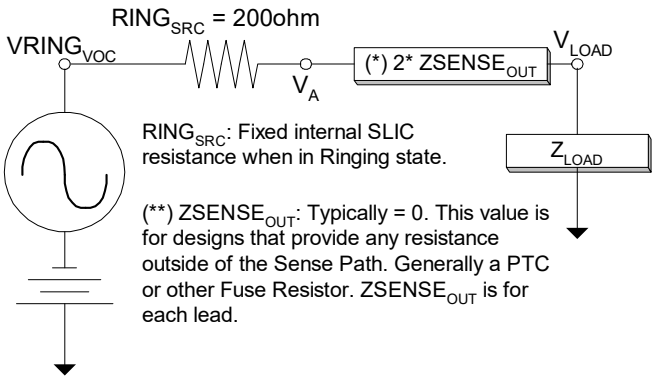
$$\frac{V_{LOAD}}{VRING_{VOC}} = \frac{Z_{LOAD}}{(Z_{LOAD} + 2* ZSENSE_{OUT} + 200ohm)}$$

1c. Invert both sides, multiply by $V_{LOAD}$ to get:

$$VRING_{VOC} = \frac{V_{LOAD} * (Z_{LOAD} + 2* ZSENSE_{OUT} + 200ohm)}{Z_{LOAD}}$$

2.Computing minVoltagePercent is straightforward at this point:

$$minVoltagePercent = \frac{100 * V_{LOAD}}{VRING_{VOC}}$$




$RING_{SRC}$: Fixed internal SLIC resistance when in Ringing state.

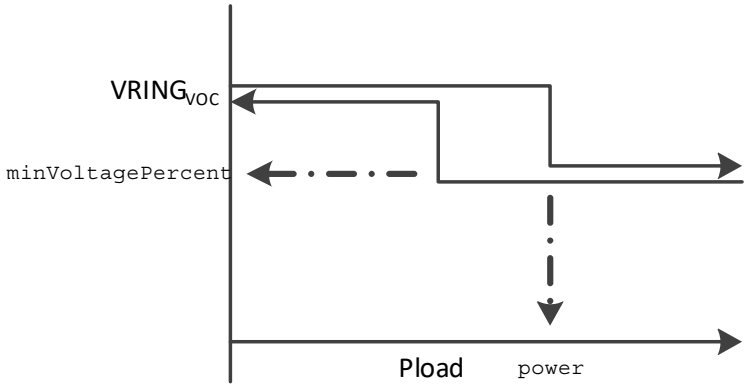
(**) $ZSENSE_{OUT}$: Typically = 0. This value is for designs that provide any resistance outside of the Sense Path. Generally a PTC or other Fuse Resistor. $ZSENSE_{OUT}$ is for each lead.

`VpAdaptiveRingingModeType mode;`

This value selects the configuration mode of the circuit, which affects how precisely the algorithm is implemented. Provided the user selects the correct `mode` to match their configuration, the algorithm will behave as follows:

1. The ringing voltage starts at the low voltage specified by the user input `minVoltagePercent`.
2. If the load power is low (i.e., low REN = high impedance), then the algorithm will switch to the higher (open circuit ringing voltage) specified in the Ringing Profile.
3. *If the load power is high (i.e., high REN = low impedance) the algorithm will remain at the* `minVoltagePercent` *voltage.*
4. As shown in Figure 3–5 the input value `power` is the point as the load power increases (from low to high) that the algorithm will switch from high ringing voltage to the low ringing voltage specified by `minVoltagePercent`. The point to switch back as the load power decreases (from high to low) is different than the `power` value and is automatically computed by the VP-API-II.

**Figure 3–5 Adaptive Ringing Input Arguments**




The possible values for `mode` are:

1. *`mode = VP_ADAPT_RING_SHARED_TRACKER:`*
   – This mode is for Shared Tracking configurations only. Applying to other configurations may cause undesirable behavior.
2. *`mode = VP_ADAPT_RING_SHARED_BB_ABS:`*
   – This mode is for Shared Buckboost ABS configurations only. Applying to other configurations may cause undesirable behavior.
3. *`mode = VP_ADAPT_RING_SINGLE_BB_TRACKER:`*
   – This mode is for Single-channel Buckboost Tracker configurations only. Applying to other configurations may cause undesirable behavior.
4. *`mode = VP_ADAPT_RING_FIXED_TRACKER:`*
   – This mode is for all fixed-ringing tracker configurations.
5. *`mode = VP_ADAPT_RING_FULL_TRACKING:`*
   – This mode is for all tracked-ringing tracker configurations.

**DEFAULT**

```
validMask = 0 /* Must always be set by Application to modify parameters */
power = 0xFF /* Adaptive Ringing is disabled */
minVoltagePercent = 83%
mode = VP_ADAPT_RING_SHARED_TRACKER
```

## 3.3.21 VP_OPTION_ID_DTMF_MODE

**DESCRIPTION**

The DTMF mode option enables or disables DTMF detection for a line.

DTMF detection parameters are passed through the VpOptionDtmfModeType structure shown below.

```
typedef struct {
    VpOptionDtmfModeControlType dtmfControlMode; /* Enable/Disable */
    VpDirectionType direction;                   /* Detection direction */

    /* Not applicable to ZL880/miSLIC devices */
    uint32 dtmfDetectionSetting;
    uint8 dtmfResourcesRemaining;
    uint8 dtmfDetectionEnabled[VP_LINE_FLAG_BYTES];
} VpOptionDtmfModeType;

Enumeration Data Type: VpOptionDtmfModeControlType:
    VP_OPTION_DTMF_DECODE_OFF = 0  /* Disable DTMF Digit Decode */
    VP_OPTION_DTMF_DECODE_ON  = 1  /* Enable DTMF Digit Decode */

Enumeration Data Type: VpDirectionType:
    VP_DIRECTION_DS  /* Upstream */
    VP_DIRECTION_US  /* Downstream */
```

For devices where **VpGetDeviceInfo()** returns `revCode >= 7`, the `direction` parameter chooses between two available detection modes. The downstream detection mode measures tip-ring voltage directly, so will detect both upsteam and downstream. The upstream detection mode uses samples from the voicepath, resulting in attenuated (but not completely eliminated) detection of downstream tones, and improved overall performance. With older devices (`revCode < 7`) the downstream mode is used regardless of the `direction` setting.

Detection thresholds can be configured using **VP_OPTION_ID_DTMF_PARAMS**.

When a DTMF digit is detected, **VpGetEvent()** will return a **VP_LINE_EVID_DTMF_DIG** event.

This option is intended to be used in applications where the control processor does not have access to the voice data and therefore cannot run its own DTMF decode algorithm.

The decode algorithm enabled by this option runs in software, and the samples are taken through additional register reads. This means that it will not offload any work from the host processor, and will require a large amount of SPI traffic to acquire samples. The SPI traffic usage is around 10000-12000 bytes per second for each channel.

The timing requirements for interrupt handling are also tighter when this option is enabled. If a ticked polling method is used, the tick period must be 10ms or less, instead of the normal requirement of 20ms. In interrupt-driven modes, the interrupt handler delay must not exceed 5ms.

If these timing requirements are violated, an overflow condition will occur, accompanied by a warning debug message. During overflow, no detection can occur. If the timing is corrected, the overflow condition will clear, and detection will resume. This means that an occasional delay will not be fatal, but will result in degraded performance.

Sampling and detection will only run in the following line states, and their reverse polarity equivalents:

```
VP_LINE_ACTIVE
VP_LINE_OHT
VP_LINE_TALK
```

In other line states, sampling and detection will be suspended automatically.

**DEFAULT**

```
VpOptionDtmfModeType::dtmfControlMode = VP_OPTION_DTMF_DECODE_OFF;
VpOptionDtmfModeType::direction = VP_DIRECTION_US;
```

## 3.3.22 VP_OPTION_ID_HIGHPASS_FILTER

**DESCRIPTION**

The high-pass filter option allows the 50/60 Hz notch filter in the A->D path to be disabled or enabled. This option is line specific and is passed through a variable of the type `VpHighPassFilterType`, shown below.

Enumeration Data Type: `VpOptionHighPassFilterType`:

```
VP_HIGHPASS_FILTER_DISABLE = 0
VP_HIGHPASS_FILTER_ENABLE = 1
```

**DEFAULT**

`VP_HIGHPASS_FILTER_ENABLE`

## 3.3.23 VP_OPTION_ID_DTMF_PARAMS

**DESCRIPTION**

The DTMF detection parameters option allows the application to configure several of the detection thresholds used when the **VP_OPTION_ID_DTMF_MODE** option is enabled.

The DTMF detection parameters are passed through the **VpOptionDtmfParamsType** struct shown below:

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

typedef struct VpOptionDtmfParamsType {
    uint32 validMask;
    int32 minDetect;
    int32 rowToColLimit;
    int32 colToRowLimit;
} VpOptionDtmfParamsType;
```

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

| **validMask bit Name** | **Field Affected** |
|---|---|
| VP_OPTION_CFG_MIN_DETECT | minDetect |
| VP_OPTION_CFG_ROW_TO_COL_LIMIT | rowToColLimit |
| VP_OPTION_CFG_COL_TO_ROW_LIMIT | colToRowLimit |

When using **VpGetOption()**, the `validMask` result tells which fields are valid.

**minDetect** sets the minimum detection level. If the level of either the row or column tone is below this threshold, a digit will not be reported. Step size is 0.1 dBm (or dBm0, depending on **VP_OPTION_ID_DTMF_MODE**). Range is 0 dBm to -53.5 dBm.

**rowToColLimit** sets the twist threshold for how high the row tone's level can be above the column tone's level. If the ratio of the row level to the column level exceeds this threshold, a digit will not be reported. Step size is 0.1 dB. Range is 0 dB to 70 dB.

**colToRowLimit** sets the twist threshold for how high the column tone's level can be above the row tone's level. If the ratio of the column level to the row level exceeds this threshold, a digit will not be reported. Step size is 0.1 dB. Range is 0 dB to 70 dB.

Row tones are defined as the frequencies from 697 to 941 Hz. Column tones are the frequencies from 1209 to 1633 Hz.

**DEFAULT**

```
VpOptionDtmfParamsType::minDetect = -300 /* -30.0 dBm0 */
VpOptionDtmfParamsType::rowToColLimit = 95 /* 9.5 dB */
VpOptionDtmfParamsType::colToRowLimit = 50 /* 5.0 dB */
```

## 3.3.24 VP_DEVICE_OPTION_ID_FSYNC_RATE

**DESCRIPTION**

The framesync rate option allows the device to be configured to operate on either an 8kHz or 16kHz framesync pulse rate.

Enumeration Data Type: `VpOptionFsyncRateType`:

```
VP_FSYNC_RATE_8KHZ = 0
VP_FSYNC_RATE_16KHZ = 1
```

When using the 16kHz mode, **`VP_OPTION_ID_CODEC`** must be set to a wideband mode.

This option is currently only suppored on single-channel devices (Le9641, Le9651) where **`VpGetDeviceInfo()`** returns `revCode >= 8`.

**DEFAULT**

Automatically detected during **`VpInitDevice()`**

## 3.3.25 VP_DEVICE_OPTION_ID_RING_PHASE_SYNC

**DESCRIPTION**

The ringing phase sync option can be used to align the phases of the ringing signals of two channels in order to reduce peak power draw.

Enumeration Data Type: `VpOptionRingPhaseSyncType`:

```
VP_RING_PHASE_SYNC_DISABLE = 0
VP_RING_PHASE_SYNC_90_OFFSET = 1
```

When this option is set to `VP_RING_PHASE_SYNC_90_OFFSET` the ringing signals will be offset by 90 degrees from each other, so that the peaks on each channel line up with the zeros on the other channel.

When a second channel begins ringing, the ringing on the other channel will be briefly interrupted in order to align the two. The interruption is short enough that most phones should be unaffected.

If this is enabled on a device where **`VpGetDeviceInfo()`** returns revCode < 7, then **`VpSysWait()`** will be used to perform the alignment and must be implemented. On devices with revCode >= 8 there is no need to implement **`VpSysWait()`**.

For this option to work, the ringing frequency must be the same for both channels.

**DEFAULT**

`VP_RING_PHASE_SYNC_DISABLE`

## 3.3.26 VP_OPTION_ID_RINGTRIP_CONFIRM

**DESCRIPTION**

The ring trip confirm option allows the VP-API-II to filter out short ring trip detections that the application would like to ignore.

```
uint16 ringTripConfirm; /* ms units */
```

The value of `ringTripConfirm` is the approximate threshold, in milliseconds, for whether a ring trip will be reported or filtered out. If it is set to 0, the option is disabled.

If a ring trip occurs but the device detects onhook again before the specified time passes, the VP-API-II will report no event, and ringing will continue. If the offhook detection lasts longer than the specified time, the VP-API-II will report a **`VP_LINE_EVID_HOOK_OFF`** event and set the line to the ring trip exit state.

Due to uncertainties in determining exactly when the beginning of the ring trip occurred, there is a grey area of approximately one ringing period, centered around the specified time.

For example, if `ringTripConfirm` is set to 200ms and the ringing period is 40ms (25Hz), then ring trips up to 220ms in duration may be filtered out, and ring trips as low as 180ms in duration may be reported. The threshold should be set accordingly, depending on whether the application requires a "must-reject" or "must-detect" condition.

The minimum recommended non-zero setting for this option depends on the ringing period and the number of cycles for ring trip integration, as chosen in the Ringing Profile. To function reliably, the time should be set no lower than (ringingPeriod * (ringTripCycles + 1)) ms.

**DEFAULT**

```
uint16 ringTripConfirm = 0; /* Disabled */
```

## 3.3.27 VP_DEVICE_OPTION_ID_ACCESS_CTRL

**DESCRIPTION**

The access control option provides a way to disable and re-enable VP-API functions that access the device, which can be useful in dealing with situations where SPI/ZSI communication is temporarily taken down.

Enumeration Data Type: `VpOptionAccessCtrlType`:

```
VP_ACCESS_CTRL_NORMAL = 0
VP_ACCESS_CTRL_BLOCK_DEVICE = 1
```

Setting `VP_ACCESS_CTRL_BLOCK_DEVICE` can be useful if one module in an application is aware of an impending loss of SPI/ZSI communication with the device, but has no easy way of communicating that to other modules which may continue to call VP-API functions. By blocking access with this option, further VP-API calls which would attempt to read or write device registers will instead fail gracefully and return `VP_STATUS_ACCESS_BLOCKED`.

When preparing for a planned downtime of SPI or ZSI communication, this option should be the last VP-API operation performed (if it is needed). It should then be the first thing re-enabled after communication is restored.

***Notes:***

1. *This option will not prevent direct access to the device through `VpMpiCmd()` or low-level SPI functions.*

**DEFAULT**

`VP_ACCESS_CTRL_NORMAL`

## 3.3.28 VP_DEVICE_OPTION_ID_SPI_ERROR_CTRL

**DESCRIPTION**

The SPI error control option allows the application to configure how the VP-API behaves when SPI/ZSI errors are detected.

SPI error control parameters are passed through the `VpOptionSpiErrorCtrlType` structure defined below.

```
typedef struct {
    VpSpiErrorModeType mode;
    uint16 count;
} VpOptionSpiErrorCtrlType;
```

Enumeration Data Type: `VpSpiErrorModeType`:

```
VP_SPI_ERROR_MODE_BLOCK = 0
```

Currently the only **`mode`** is `VP_SPI_ERROR_MODE_BLOCK`. This mode will block device access after detecting a number of SPI errors that meets the **`count`** threshold. When blocked, VP-API functions which would attempt to read or write device registers will instead return VP_STATUS_ERR_SPI.

If the **count** threshold is set to 0, no blocking will occur. Events will still be reported (see **`VP_DEV_EVID_SYSTEM_FLT`**).

Each time this option is configured, access will be unblocked and the running count of errors will be reset to 0. This applies even if the parameters are not changed from the previous configuration.

***Notes:***

1. *The VP-API attempts to detect SPI errors while polling the device for interrupts during VpGetEvent(). It is possible for SPI errors to occur and not be detected during other operations.*

**DEFAULT**

```
VpOptionSpiErrorCtrlType::mode = VP_SPI_ERROR_MODE_BLOCK
VpOptionSpiErrorCtrlType::count = 1
```

CHAPTER

# 4 DESIGN CONSIDERATIONS

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

## 4.1 RETURN CODE HANDLING

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

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

   ***Notes:***

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

3. `VpDeviceIoAccess()` will protect Application changes to I/O2 if this pin was NOT configured as a digital signal in the Device Profile. If the Application attempts to modify I/O2 when it is not set to a digital signal will result in VP-API-II return `VP_STATUS_DEDICATED_PINS`. In this case, I/O1 on each channel may still be modified by the call to `VpDeviceIoAccess()`.

## 4.2 INITIALIZATION OF THE VP-API-II

This section addresses application specific topics regarding initialization of the VP-API-II.

### 4.2.1 Basic Initialization of the VP-API-II

Details aside, the following steps can be followed for all Applications:

1. Call `VpMakeDeviceObject()`
2. Call `VpMakeLineObject()` /* Repeat for each line associated with the device */
3. Call `VpInitDevice()` and wait for the Init Device Complete Event.
4. Unmask all events that will be handled by the Application.
   a) Most events are masked by default, so make sure `VP_EVID_CAL_CMP` is unmasked. This is needed to keep track of when the lines have finished calibration in this method.
   b) See `VpSetOption()` and `VP_OPTION_ID_EVENT_MASK` for details.
5. (optional) Check for FEMF or Hazardous Potential on the line. Requires use of Microsemi Line Test SW.
6. Set all FXS lines to `VP_LINE_STANDBY` or `VP_LINE_OHT`. These are two of the valid line states the FXS line must be in to perform `VpCalLine()`.
7. Call `VpCalLine()` for each line and if `return = VP_STATUS_SUCCESS`, wait for `VP_EVID_CAL_CMP` event (also on each line).
8. Proceed with normal configuration and call control.

The line disconnect time due to the calibration in steps 3. and 7. can be up to 500ms. If undesireable, this disconnect interval can be significantly reduced using one of the "Fast Initialization" methods described in the next section (Section 4.2.2).

### 4.2.2 Fast Initialization (Calibration Bypass)

The initialization time can be significantly reduced by pre-determining the calibration correction factors for the device and lines associated with the device. This is referred to as “Calibration Bypass” and can be accomplished one of two ways:

1. Warm Reboot detection by the VP-API-II such that the device object retains calibration data throughout the initialization sequence.
2. Pre-Load the calibration coefficients into the VP-API-II prior to `VpInitDevice()`. This method requires use of function `VpQueryImmediate()` and `VpCal()`.

#### 4.2.2.1 Method 1: “Warm Reboot”

This method works on the idea that the Application is restarting but is NOT resetting critical data used by the VP-API-II.

Remember from Section 1.8.1 that `VpMakeDeviceObject()` and `VpMakeLineObject()` initialize the device and line objects respectively. These objects are then filled in and modified by the VP-API-II during run-time with calibration and other relevant data.

During `VpInitDevice()` the VP-API-II checks the content of the device object and makes a determination as to whether the content is valid from a previously running condition or reset from `VpMakeDeviceObject()`. If the VP-API-II detects that the device object is from a previously running condition, it sets a flag that will cause the calibration procedures in `VpInitDevice()` and `VpCalLine()` to be bypassed, and instead apply the previously stored calibration data during the corresponding device and line configuration procedures. Note that the Application may call `VpCalLine()` even after previous calibration values have been detected with no harm done. As described in Section 6.2.5, `VpCalLine()` will simply return `VP_STATUS_SUCCESS` and immediately generate the `VP_EVID_CAL_CMP` event.

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

***Notes:***

*In a “Warm Reboot” scenario the VP-API-II only checks the device object for previously initialized data, it does not check the line objects associated with the device. If the device object data is valid, the VP-API-II assumes the line object data is also valid and proceeds. Therefore, if the Application calls `VpMakeDeviceObject()` or `VpMakeLineObject()` it must perform a complete restart.*

#### 4.2.2.2 Method 2: Retrieve/Apply Calibration Coefficients

- Requires: `VpQueryImmediate()` with VP_QUERY_ID_LINE_CAL_COEFF for retrieving the calibration coefficients from the VP-API-II.
- Requires: `VpCal()` with `VP_CAL_APPLY_LINE_COEFF` for applying the calibration coefficients to the VP-API-II.

#### 4.2.2.3 Retrieving Calibration Coefficients from the VP-API-II

1. Call `VpMakeDeviceObject()`
2. Call `VpMakeLineObject()` /* Repeat for each line associated with the device */
3. Call `VpInitDevice()` and wait for Init Device Complete event.
4. Unmask all events that will be handled by the Application.
   a) Make sure `VP_EVID_CAL_CMP` is unmasked since this is needed to keep track of when the lines have finished calibration in this method.

b) See `VpSetOption()` and `VP_OPTION_ID_EVENT_MASK` for details.

5. Set all FXS lines to `VP_LINE_STANDBY` or `VP_LINE_OHT`.
6. Call `VpCalLine()` for each line and wait for the `VP_EVID_CAL_CMP` event.
7. When `VP_EVID_CAL_CMP` event is generated (per line), call `VpQueryImmediate()` for that line with **`queryId = VP_QUERY_ID_LINE_CAL_COEFF`** and **`*pResults = uint8 calProf[256]`** (see Section 10.2.9 for details). Note: **`calProf`** used in "Application of Calibration Coefficients" next.

#### 4.2.2.4 Application of Calibration Coefficients ("Fast Initialization"):

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

   ***Notes:***

   *When `VpCal()` is called prior to `VpInitDevice()` event `VP_EVID_CAL_CMP` will NOT be generated. This is because events can only be generated by devices AFTER they have been initialized.*

4. Call `VpInitDevice()`
5. Wait for Init Device complete event (which will be almost immediate since the system has been pre-loaded with calibration coefficients, thus avoiding the run-time calibration procedures).
6. Unmask all events that will be handled by the Application.

   a) Note that `VP_EVID_CAL_CMP` may NOT be needed in this case because all device and line calibration data has been applied prior to `VpInitDevice()`. For a basic application, there is no need to call functions that generate this event.

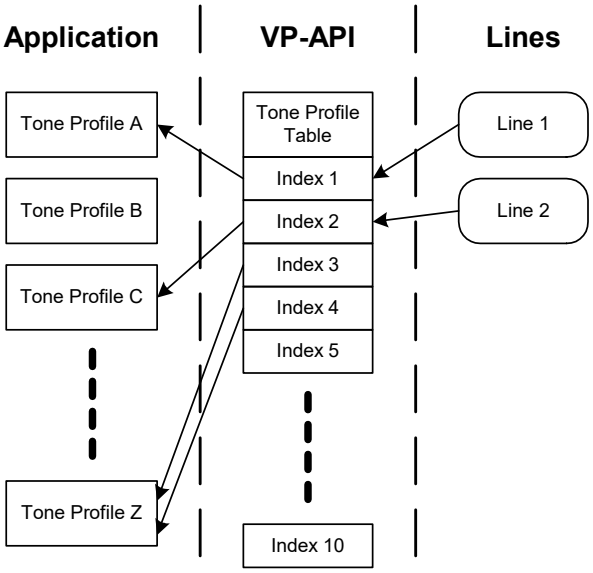
   b) Application call to `VpCalLine()` after the line has been calibrated (as a result of step #3) will act the same as if line calibration had been performed. `VpCalLine()` will return `VP_STATUS_SUCCESS` and the event `VP_EVID_CAL_CMP` will be generated (if unmasked). Except this event will occur immediately (when `VpCalLine()` returns) and none of the calibration processes will be performed for that line.

7. (optional) Check for FEMF or Hazardous Potential on the line. Requires use of Microsemi Line Test SW.
8. Set all FXS lines to `VP_LINE_STANDBY` or `VP_LINE_OHT`.
9. Proceed with normal configuration and call control.

## 4.3 SOFT PROFILE TABLES

The profile tables described in Section 2.3 for ZL880/miSLIC devices are simulated in software (i.e., “soft” profile tables). These *soft profiles tables* are stored in memory allocated by the Application software and referenced by the VP-API-II as necessary. The VP-API-II *does not* store a copy of the profiles internally; it only retains pointers to the profiles. The Application software should take extra care not to delete or modify a soft profile as long as the VP-API-II may be using it. This restriction applies to both soft profile tables and any soft profile that is passed to the VP-API-II by reference (using a profile pointer instead of a profile table index). Figure 4–1 on page 56 illustrates the concept of soft profiles.

**Figure 4–1 Soft Profile Table Example**




This example shows a hypothetical tone profile table setup with the VP-API-II controlling two telephone lines through a ZL880/miSLIC device. Tone profile table Index 1 (`VP_PTABLE_INDEX1`) references the Application’s Tone Profile A, and tone profile table Index 2 (`VP_PTABLE_INDEX2`) references the Application’s Tone Profile C.

In this example, Application Tone Profile A and Tone Profile C should not be modified because Lines 1 and 2 are linked to these profiles. Application Tone Profile B may be modified because the VP-API-II has no knowledge of that profile.

## 4.4 MULTI-TASKING APPLICATIONS

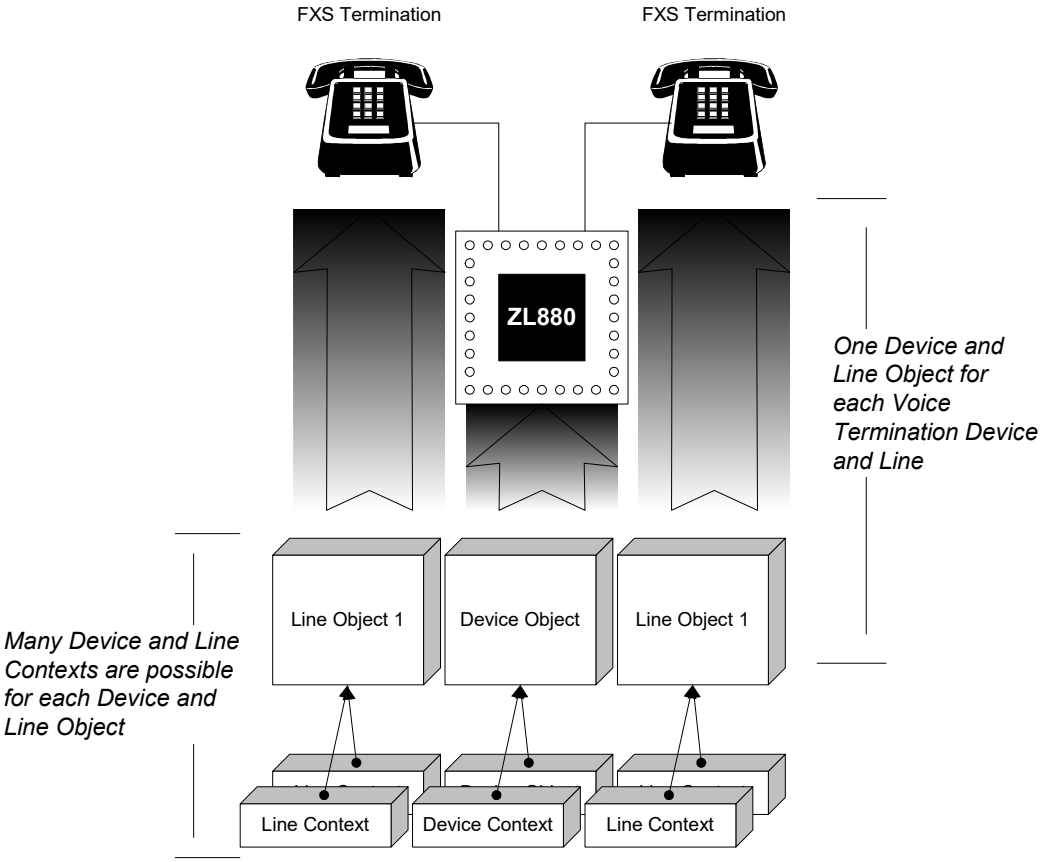
It may be desirable to have multiple tasks controlling various aspects of the voice path. One common example is one handling call control while another managing line testing. In this example, both tasks must share access to the VP-API-II.

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

### 4.4.1 Multiple Device and Line Contexts

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

**Figure 4–2 Multi-Tasking Example**




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

Note that only one task should call `VpGetEvent()` and `VpGetResults()`. If multiple tasks need to receive VP-API-II events, a centralized *event dispatcher* task should be implemented to call

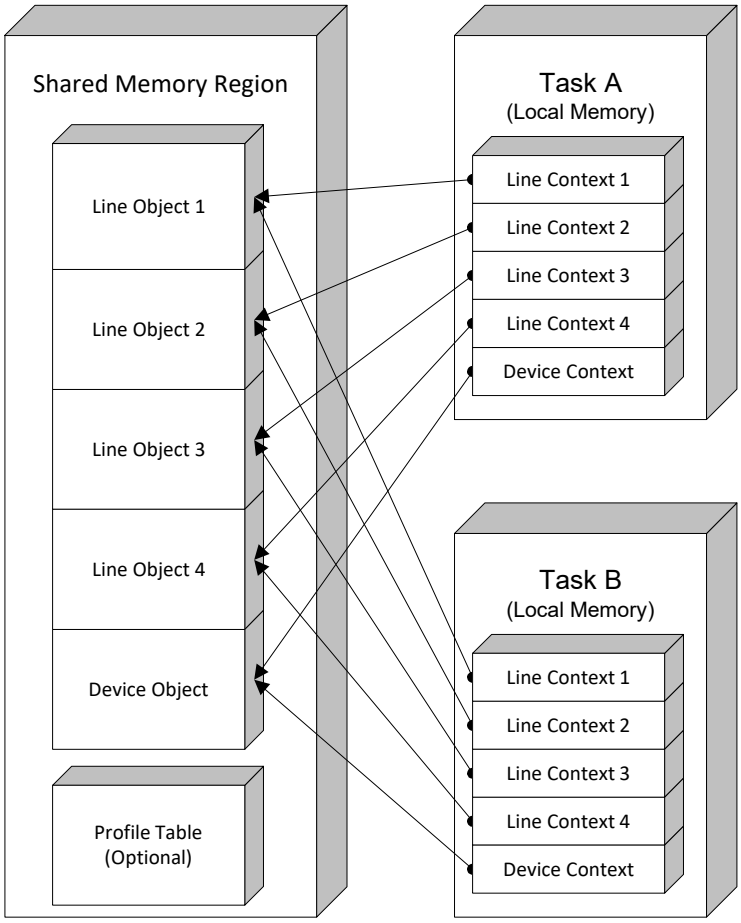
`VpGetEvent()` and `VpGetResults()` and forward the events to the desired tasks. Also, for ZL880/miSLIC devices, only one task should perform the VP-API-II tick function.

### 4.4.2 Protected Memory Requirements

In multi-tasking environments with memory protection, a shared memory region is necessary to share the device and line objects between many tasks. In the example depicted in Figure 4–3 on page 58, a shared memory region is shown with two tasks (A and B) both needing access to the VP-API-II. One task is responsible for creating the shared memory region, and creating the desired device and line objects. All tasks must create device and line contexts for use with VP-API-II function calls.

For systems using “soft” profiles (see *Soft Profile Tables*, on page 56), 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()`.

**Figure 4–3 Protected Memory Example**



## 4.5 CODE SIZE MANAGEMENT

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

Code size can be managed based on certain Application level choices. Table 4–1 shows the reductions possible in the ZL880/miSLIC VP-API-II.

***Notes:***

*Setting some of the Code Size Management values to* ***#undef*** *may result in some source files being compiled down to nothing. This can create an error for some compilers if the reduced file is included in the build process.*

**Table 4–1 Code Size Management Settings (vp_api_cfg.h)**

| Value | Comment | Default |
|---|---|---|
| **VP_REDUCED_API_IF** | If #define removes functions from the VP-API-II Interface. See Section 4.5.3 for details. | #define |
| **VP_DEBUG** | If #define compiles in run-time debug strings used by the VP-API-II. Setting to #undef minimizes code size. | #undef |
| **VP_CSLAC_SEQ_EN** | If #undef, reduces functionality of the VP-API-II. To enable all features of the VP-API-II, this must be set to #define. See Section 4.5.3 for details. | SW Package Specific |
| **VP_CSLAC_RUNTIME_CAL_ENABLED** | Run-time calibration supported. See Section 4.5.4 for details. | #define |
| VP_CSLAC_HIGH_GAIN_SUPPORTED | Set to #define to support line states VP_LINE_HOWLER and VP_LINE_HOWLER_POLREV. | #define |
| VP886_INCLUDE_ADAPTIVE_RINGING | If defined, enables the Adaptive Ringing feature described in **VP_DEVICE_OPTION_ID_ADAPTIVE_RINGING**. See Section 3.3.20 for details. | #undef |
| VP886_INCLUDE_DTMF_DETECT | If defined, enables the DTMF detection feature. See *VP_OPTION_ID_DTMF_MODE*, on page 47 for details. | #undef |

### 4.5.1 VP_REDUCED_API_IF

When set to #define the VP-API-II interface is minimized. Using a combination of other compile-time settings in vp_api_cfg.h, setting this to #define will then result in removal of Functions and Options that would return error code if called by the Application (i.e., Functions and Options that cannot be used by the Application are removed). Applications that contain calls to these (removed) Functions will generate a compiler error.

### 4.5.2 VP_DEBUG

Enabling VP_DEBUG (i.e.,#define VP_DEBUG) will significantly increase code size, and therefore not recommended for code space limited Applications. Setting VP_DEBUG to #undef removes all debug facilities and helps minimize code size. If run-time debug is required, selective debug mechanisms may be compiled in by setting VP_DEBUG to #define and properly defining VP_CC_DEBUG_SELECT. See Chapter 13 for details.

### 4.5.3 VP_CSLAC_SEQ_EN

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

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

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

- VpSetLineTone() - Function will accept tone and tone cadence profiles, but the cadence profile will be ignored. All tone cadence timing must be provided by the Application.
- VpSendSignal() - Not supported.
- Caller ID functions: VpInitCid(), VpSendCid() and VpContinueCid() - Not supported.
- VpInitRing() - Not supported. All ring cadence timing must be provided by the Application.
- Metering functions: VpInitMeter() and VpStartMeter() - Not supported.

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

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

### 4.5.4 VP_CSLAC_RUNTIME_CAL_ENABLED

Setting this compile time value to #undef removes the internal VP-API-II source that performs the calibration algorithms. Since the ZL880/miSLIC silicon requires calibration, the Application would have to provide a pre-determined set of calibration values using VpCal() (**calType = VP_CAL_APPLY_LINE_COEFF**) if run-time calibration is removed.

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

## 4.6 ZL880/MISLIC VP-API-II CONFIGURATION

The ZL880/miSLIC family provides a few configuration settings that can be adjusted to optimize the Application. The values shown in the table below and discussed in the following sections are in vp_api_cfg.h:

**Table 4–2 ZL880/miSLIC VP-API-II Configuration Settings**

| Value | Comment | Default |
|---|---|---|
| VP886_EVENT_QUEUE_SIZE | Defines the size of the ZL880 VP-API-II software event queue. | 6 |
| VP886_USER_TIMERS | This value sets the number of Application timers that can be used with VpGenTimerCtrl() for each ZL880 device. Larger values will result in larger device objects. | 2 |
| VP886_LEGACY_SEQUENCER | Configures the ZL880 Sequencer as described in *Cadence Profiles*, on page 153. | #undef |
| VP886_INCLUDE_MPI_QUICK_TEST | Enables a test of the MPI and HAL implementations during execution of VpInitDevice(). | #define |
| VP886_SIMPLE_POLLED_MODE | These polling/interrupt modes define how the Application will detect events from the VP-API-II. See *VpApiTick() and Polling/Interrupt Modes*, on page 156. for details. | #undef |
| VP886_EFFICIENT_POLLED_MODE | | #define |
| VP886_INTERRUPT_EDGETRIG_MODE | | #undef |
| VP886_INTERRUPT_LEVTRIG_MODE | | #undef |
| VP_ENABLE_OFFHOOK_MIN | If defined, enables the parameter **offHookMin** in options VP_DEVICE_OPTION_ID_PULSE and VP_DEVICE_OPTION_ID_PULSE2, and event VP_LINE_EVID_HOOK_PREQUAL. See Section 3.3.5 for details. | #undef |

### 4.6.1 VP886_EVENT_QUEUE_SIZE

Applications may have to increase this value if receiving **VP_DEV_EVID_EVQ_OFL_FLT** events. See *VP_DEV_EVID_EVQ_OFL_FLT*, on page 87 for details and notes regarding **VP886_EVENT_QUEUE_SIZE**.

### 4.6.2 VP886_USER_TIMERS

This value sets the number of Application timers that can be used with **VpGenTimerCtrl()** for each ZL880/miSLIC device. Larger values will result in larger device objects.

### 4.6.3 VP886_LEGACY_SEQUENCER

Configures the ZL880/miSLIC Sequencer as described in *Cadence Profiles*, on page 153. When set to `#define` the ZL880/miSLIC Cadencer will operate the same way as the VE880 Cadencer (i.e., time compensation is required). When set to `#undef` the ZL880/miSLIC Cadencer will perform each step in-time (i.e., time compensation is not required). See *Cadence Profiles*, on page 153 for details.

CHAPTER

# 5 SYSTEM CONFIGURATION FUNCTIONS

## 5.1 OVERVIEW

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

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

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

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

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

**Required Functions for All Applications**

- `VpMakeDeviceObject()` – This is the MOST important function of the VP-API-II. This function initializes the device object and associates it with the device context. By initializing the device object with the `deviceId`, it links the Application space device context to the physical device. The `deviceType` determines the silicon register set used for controlling the device.
- `VpMakeLineObject()` – This initializes the line object and associates it with the line and device context. By initializing the line object with the `channelId`, it links the Application space line context to the physical line in the silicon (value of `channelId` being limited to the range of the silicon). The `termType` tells the VP-API-II how a line is to be physically managed (i.e., controlled and line status detectors interpreted).

**Additional Functions Required Functions for Multi-Threaded Applications**

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

**Optional/Conditional Functions**

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

## 5.2 OBJECTS AND CONTEXTS

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

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

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

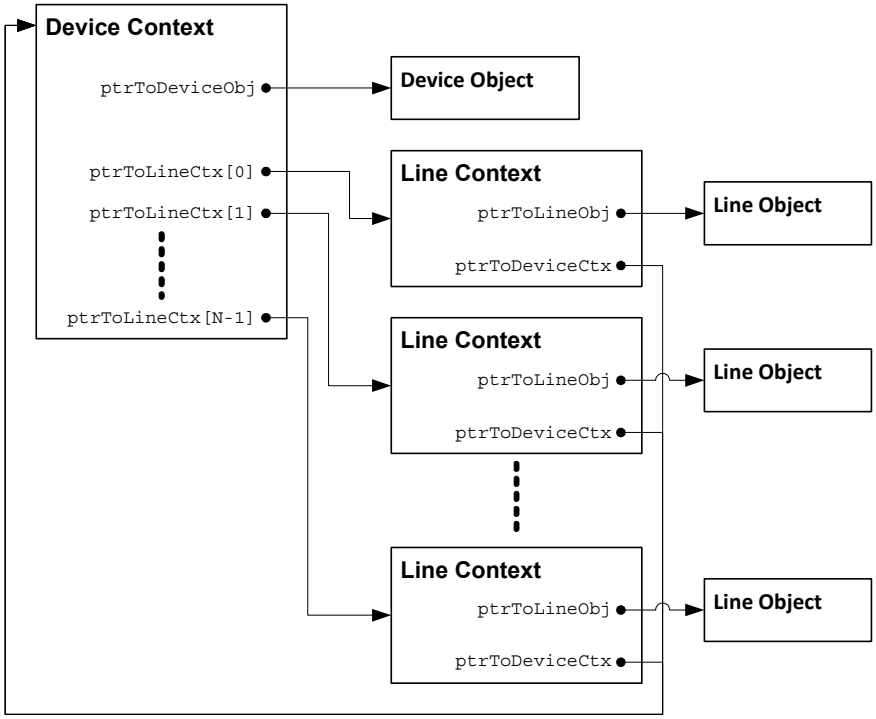
**Table 5–1 Device and Line Objects**

| VpDeviceType | Compile-Time Switch | Device Object | Line Object |
|---|---|---|---|
| VP_DEV_886_SERIES | VP_CC_886_SERIES | Vp886DeviceObjectType | Vp886LineObjectType |
| VP_DEV_887_SERIES | | | |

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

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

**Figure 5–1 Device and Line Objects and Contexts**




To support both static and dynamic memory models, the VP-API-II itself does not allocate any storage for any type of object or context. The Application must allocate storage for these structures, then execute VP-API-II functions to initialize the device and line objects and their respective contexts. It is therefore 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 Application then must call **VpFreeLineCtx()** (in case of freeing memory previously allocated to a line context or the line object it's pointing to).

## 5.3 FUNCTION DESCRIPTIONS

### 5.3.1 VpMakeDeviceObject()

**SYNTAX**

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

**DESCRIPTION**

This function does the following:

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

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

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

***Notes:***
*The device object contains critical VP-API-II information. Therefore, after calling this function the device and all lines of the device MUST be restarted.*

**deviceId**

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

**deviceType**

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

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

***Notes:***

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

**RETURNS**

**VP_STATUS_SUCCESS** (otherwise the device is invalid)
See **VP-API-II Return Codes (VpStatusType)** for all other possible return values.

**EVENTS GENERATED**

None

## 5.3.2 VpMakeLineObject()

**SYNTAX**

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

**DESCRIPTION**

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

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

The termination type parameter (`termType`) describes the circuitry associated with the termination. The termination types matched to the devices that support each type are defined in Table 1–4.

The value required for `channelId` for a specific tip and ring port corresponds to the physical channel of the device, so it must equal '0' when referring to the 1st physical line and '1' when referring to the 2nd physical line.

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

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

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

***Notes:***

1. *No line-specific VP-API-II functions should be called before invoking this function.*
2. *The device context pointed to by* `pDevCtx` *must be initialized with* **`VpMakeDeviceObject()`** *or* **`VpMakeDeviceCtx()`** *before calling this function.*
3. *The type of line object allocated by the Application must be compatible with the device type of the given device context. The VP-API-II has no way to check this condition, and the Application could fail if there is a mismatch. See Table 5–1 for a list of device types and matching line object types.*
4. *This function must be called before executing* **`VpInitDevice()`**. *See VpInitDevice(), on page 69 for more information.*

**FUNCTION RETURNS**

**`VP_STATUS_SUCCESS`** (otherwise the line is invalid)
See **`VP-API-II Return Codes (VpStatusType)`** for all other possible return values.

**EVENTS GENERATED**

None

### 5.3.3 VpMakeDeviceCtx()

**SYNTAX**

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

**DESCRIPTION**

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

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

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

This function returns VP_STATUS_INVALID_ARG if pDevCtx or pDevObj is VP_NULL.

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

**RETURNS**

VP-API-II Return Codes (VpStatusType)

**EVENTS GENERATED**

None

### 5.3.4 VpMakeLineCtx()

**SYNTAX**

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

**DESCRIPTION**

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

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

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

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

**RETURNS**

VP-API-II Return Codes (VpStatusType)

**EVENTS GENERATED**

None

### 5.3.5 VpFreeLineCtx()

**SYNTAX**

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

**DESCRIPTION**

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

In case more than one line context is associated with one line object (see *Multi-Tasking Applications*, on page 57), this function must be called for all such line contexts before freeing up the memory allocated to the line object.

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

**RETURNS**

**VP-API-II Return Codes (VpStatusType)**

**EVENTS GENERATED**

None

### 5.3.6 VpMapLineId()

**SYNTAX**

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

**DESCRIPTION**

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

**RETURNS**

**VP-API-II Return Codes (VpStatusType)**

**EVENTS GENERATED**

None

CHAPTER

# 6 INITIALIZATION FUNCTIONS

## 6.1 OVERVIEW

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

- `VpInitDevice()` – Initializes all lines of a device and applies the specified profiles to those lines. This function is required by all Applications.
- `VpInitLine()` – Initializes an individual line and applies the specified profiles to that line.
- `VpConfigLine()` – Sets the AC, DC, and Ring Profiles for an individual line.
- `VpCal()` – Calibrates the line to external parameters and provides methods for managing system calibration coefficients.
- `VpCalLine()` – Calibrates internal line level parameters.
- `VpInitRing()` – Sets the ringing cadence parameters and Caller ID profile for the line.
- `VpInitCid()` – Initializes the CID buffer prior to Ringing associated Caller ID.
- `VpInitMeter()` – Configures the metering signal generator of the line.
- `VpInitProfile()` – Initializes the device's profile tables.

## 6.2 FUNCTION DESCRIPTIONS

### 6.2.1 VpInitDevice()

**SYNTAX**

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

**DESCRIPTION**

This function initializes all lines controlled by the VTD associated with the device context. This includes performing the recommended power-up sequence, performing calibration tasks, and setting the device and all lines of the device to their default option values. Not that even though this function performs steps that are contained by other VP-API-II functions, when complete it only generates `VP_DEV_EVID_DEV_INIT_CMP` event. It will NOT generate other events corresponding to the equivalent VP-API-II functions being performed (e.g., `VP_LINE_EVID_LINE_INIT_CMP`, `VP_EVID_CAL_CMP`). Also, calling other functions prior to calling `VpInitDevice()` will generally affect the normal behavior of those functions, such as preventing those functions from generating their corresponding events. Refer to Section 4.2 for `Initialization of the VP-API-II` Design Considerations including `Method 1: "Warm Reboot"`, `Fast Initialization (Calibration Bypass)`, concerning warm and cold-bootup initialization, factory/field calibration, and ABS Master, Slave and Single Configurations.

**Requirements:**

- `VpInitDevice()` must be called after creating the device via `VpMakeDeviceObject()`. It is highly recommended also to create all lines via `VpMakeLineObject()` before calling this function, but not required. Lines created after calling `VpInitDevice()` must be initialized using `VpInitLine()`.
- The `pDevProfile` parameter takes a pointer to a Device Profile and is **required**. The device profile provides basic information needed to reliably communicate with and initialize the device. This function returns an error code if `pDevProfile` does not point to a valid profile.
- Proper operation of the FXS lines requires valid set of `pAcProfile`, `pDcProfile`, and `pRingProfile` arguments. If not provided with this function call, these values can be provided using either `VpInitLine()` or `VpConfigLine()`.
- The FXO arguments `pFxoAcProfile` and `pFxoCfgProfile` are ignored and can be set to `VP_NULL` since the ZL880/miSLIC silicon does not support FXO termination types.
- The profile types accepted by this function are described in *Profile Types*, on page 19. For each of these profiles, the Application can supply either a pointer to a valid profile or a profile table index. Valid profiles must be loaded into the profile table before a profile table index can be used. See *Profiles*, on page 19.
- The profiles provided to this function can be removed from Application space once this function returns.
- Once this function is called, except for `VpGetEvent()` the Application must prevent access to the affected device until the `VP_DEV_EVID_DEV_INIT_CMP` event is generated.
- Once the device and all lines of the device have been calibrated, it is not necessary to re-calibrate the same device and lines after calling `VpInitDevice()`.

`VP886_INCLUDE_MPI_QUICK_TEST:`
The beginning of `VpInitDevice()` will perform a test of the MPI interface and Hardware Abstraction Layer implementation (i.e., `VpMpiCmd()`). If an error is detected, this function will return `VP_STATUS_ERR_SPI`. Enabling `VP_DEBUG_ERROR` output will provide details about the error that can be used for debugging. Once a design is proven to work, this test can be compiled out by undefining `VP886_INCLUDE_MPI_QUICK_TEST` in `vp_api_cfg.h`.

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

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

***Notes:***

1. *Only lines that have previously been created and associated with the device using `VpMakeLineObject()` will be initialized at the time this function is called.*
2. *If calling this function on a device with currently active lines, it is recommended that the Application first put all lines of the device into the `VP_LINE_DISCONNECT` state.*

**RETURNS** `VP-API-II Return Codes (VpStatusType)`

**EVENTS GENERATED** *VP_DEV_EVID_DEV_INIT_CMP*, on page 93

## 6.2.2 VpInitLine()

**SYNTAX**

```
VpStatusType
VpInitLine(
 VpLineCtxType *pLineCtx,                /* I/O: Pointer to line context */
 VpProfilePtrType pAcProfile,            /* Input: Pointer to AC profile */
 VpProfilePtrType
   pDcFeedOrFxoCfgProfile,               /* Input: Ptr to DC feed profile */
 VpProfilePtrType
   pRingProfile)                         /* Input: Pointer to ringing profile */
```

**DESCRIPTION**

This function resets all line parameters and options except **VP_OPTION_ID_EVENT_MASK** to their default values. **VP_OPTION_ID_EVENT_MASK** is not reset because it also affects device event masking.

This function loads the provided profiles (if any) to the line, sets the line to **VP_LINE_DISCONNECT** state and sets the relay state according to the list below.

| FXS Line Termination Type | Relay State |
|---|---|
| VP_TERM_FXS_GENERIC | VP_RELAY_NORMAL |
| VP_TERM_FXS_LOW_PWR | VP_RELAY_NORMAL |

**Requirements:**

- The profiles passed to this function may be removed from Application space once this function returns.
- Once this function is called, the Application must prevent access to the line affected until the **VP_LINE_EVID_LINE_INIT_CMP** event is generated.
- Once a line has been calibrated, it is not necessary to re-calibrate the same line after calling **VpInitLine()**.

***Notes:***

1. *If calling this function on a currently active line, it is recommended that the Application first put the line into the* **VP_LINE_DISCONNECT** *state.*
2. *Most Applications will not need to call this function explicitly. If all line contexts of the device are associated with the device context prior to calling* **VpInitDevice()**, then **VpInitDevice()** will initialize the device and call (internally) **VpInitLine()** on all lines of the device.

**RETURNS**

**VP-API-II Return Codes (VpStatusType)**

**EVENTS GENERATED**

*VP_LINE_EVID_LINE_INIT_CMP*, on page 93

## 6.2.3 VpConfigLine()

**SYNTAX**

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

**DESCRIPTION**

This function allows the Application to reconfigure the lines AC, DC, and Ringing parameters by passing appropriate profiles to the function's arguments. It does not reset the line or change the line state as the case is with `VpInitLine()` and `VpInitDevice()`, but merely loads the given profiles to the line.

**Requirements:**

- Only parameters specified by a valid profile pointer are affected by this function. Parameters specified by a `VP_PTABLE_NULL` or `VP_NULL` profile pointer are ignored leaving the line's configuration unchanged for the given parameter.
- The profiles provided to this function can be removed from Application space once this function returns.
- Once a line has been calibrated, it is not necessary to re-calibrate the same line after calling `VpConfigLine()`.

***Notes:***

1. *When changing the Ringing and/or DC parameters, the order of these parameters being changed is not guaranteed. Therefore, to avoid false ring trip or other hook detect issues, the line should be set to* **`VP_LINE_DISCONNECT`** *state prior to reconfiguring the line.*
2. *Most Applications will not need to call this function since the line can be configured during initialization by passing the necessary profiles to* `VpInitDevice()` or `VpInitLine()`. This function provides the benefit of being able to re-configure a line without causing a device or line reset.
3. *Applying a new AC profile will reset any gain changes that have been made with* `VpSetRelGain()` *or* `VP_OPTION_ID_ABS_GAIN`.

**RETURNS**

`VP-API-II Return Codes (VpStatusType)`

**EVENTS GENERATED**

None

## 6.2.4 VpCal()

**SYNTAX**

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

**DESCRIPTION**

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

```
Enumeration Data Type: VpCalType:
    VP_CAL_GET_LINE_COEFF    /* Retrieves calibration values */
    VP_CAL_APPLY_LINE_COEFF  /* Applies calibration values */
```

`VP_CAL_GET_LINE_COEFF`: This routine retrieves the per-line and device calibration values associated with the line context passed. The device and lines on the device must have been calibrated using **`VpInitDevice()`** and **`VpCalLine()`** or loaded with calibration values using option `VP_CAL_APPLY_LINE_COEFF` prior to running this routine. Otherwise this function will return **`VP_STATUS_LINE_NOT_CONFIG`**.

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

`VP_CAL_APPLY_LINE_COEFF`: This routine loads per-line and device calibration values associated with the line context passed. If called before **`VpInitDevice()`** will cause the VP-API-II to skip the run-time calibration performed normally as part of **`VpInitDevice()`** and **`VpCalLine()`** (if called for this line). Calling **`VpCalLine()`** for this line after calibration coefficients have been provided to the line will have no negative affect. It will simply result in the VP-API-II returning **`VP_STATUS_SUCCESS`** from **`VpCalLine()`** and generating **`VP_EVID_CAL_CMP`** event.

Value of `inputArgs` for `calType = VP_CAL_APPLY_LINE_COEFF` is formatted one of two valid ways:

1. *Pointer to a `uint8` string formatted with calibration coefficients per `VP_CAL_GET_LINE_COEFF`. This will apply the pointed to coefficients to the line/device immediately.*
2. *NULL Pointer. This will reset the calibration coefficients for the line and device, allowing the run-time calibration methods in **`VpInitDevice()`** and **`VpCalLine()`** to be repeated.* This feature is most utilized in Manufacturing Test environments, but can also be used in a Field Application.

***Notes:***

*When **`VpCal()`** is called prior to **`VpInitDevice()`** event **`VP_EVID_CAL_CMP`** will not be generated . This is because VP-API-II devices cannot generate events prior to initialization.*

**RETURNS**

**`VP-API-II Return Codes (VpStatusType)`**

**EVENTS GENERATED**

*VP_EVID_CAL_CMP*, on page 93 (unless used prior to calling **`VpInitDevice()`**)

## 6.2.5 VpCalLine()

**SYNTAX**

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

**DESCRIPTION**

**Calibration is required on all lines** in order to meet Data Sheet Specifications, and to avoid hook and ring trip issues. This requirement can be met in one of two ways:

1. *Calling* `VpCalLine()` *for the line - This will perform run0time calibration on the line and apply the computed correction factors to the line upon completion. If successful* `VpCalLine()` *will generate the* `VP_EVID_CAL_CMP` *event.When this event is generated, the Application can read the calibration values from the VP-API-II using* `VpQueryImmediate()`.
2. *Apply previously determined calibration factors (such as those computed using the methods described in step #1) to the line using* `VpCal()`.

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

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

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

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

1. *Calling* `VpCal()` *- When this function is called the VP-API-II will overwrite previous calibration factors with the new calibration factors provided. Note there are two modes to apply calibration factors: new values and reset (if providing NULL). For details refer to* `VpCal()` *in* Section 6.2.4.
2. *Calling* `VpMakeDeviceObject()` *or* `VpMakeLineObject()` *- This causes all data in the device/line object to be reset, including most of the calibration data. The VP-API-II cannot restore the object data from the silicon content alone. The Application must restart.*
3. *Hardware Reset - Most silicon content is reset, so regardless of calibration this condition cannot be restored by the VP-API-II. The Application must restart.*

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

***Notes:***

1. `VpCalLine()` is not performed automatically by the VP-API-II. If the Application is not using `VpCal()` to apply calibration correction factors, then the Application **must call** this function to calibrate the line.
2. *The procedures performed in* `VpCalLine()` *may cause Tip./Ring and Vbat (Tracker Configurations) transients. This may cause noise on other lines if proper layout procedures are not taken.*
3. No other API-II function for the line should be called while the line is calibrating.

**RETURNS**

`VP-API-II Return Codes (VpStatusType)`

**EVENTS GENERATED**

`VP_EVID_CAL_CMP`

## 6.2.6 VpInitRing()

**SYNTAX**

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

**DESCRIPTION**

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

The profiles may be specified by pointer or profile tables as described in *Profile Tables*, on page 20.

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

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

If Type-I Caller ID is enabled for a line, the Application must first call **VpInitCid()** to copy Caller ID data into the Caller ID message buffer before putting the line into the Ringing state. See *VpInitCid()*, on page 76 for more information. Note that although **VpInitCid()** must be called each time prior to generating CID, **VpInitRing()** need only be called once for a given ringing and (Type-I) CID profile. Once initialized, these cadences are applied each time the line is put into the ringing state or until otherwise changed (i.e., additional calls to this function or **VpInitDevice()**/**VpInitLine()**).

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

***Notes:***
*Type-I Caller ID referred to in this function has the CLI (Caller Line Identity) frame transmitted as part of the caller alert sequence (ringing signal plus Caller ID sequence and data). This type of CLI is only transmitted while the line is on-hook. Applications with Caller ID that is NOT associated with Ringing (e.g., Type-II Caller ID) should refer to function* *VpSendCid()*, on page 112.

**RETURNS**

**VP-API-II Return Codes (VpStatusType)**

**EVENTS GENERATED**

None

## 6.2.7 VpInitCid()

**SYNTAX**

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

**DESCRIPTION**

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

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

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

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

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

**RETURNS**

**VP-API-II Return Codes (VpStatusType)**

**EVENTS GENERATED**

*VP_LINE_EVID_CID_DATA*, on page 100

## 6.2.8 VpInitMeter()

**SYNTAX**

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

**DESCRIPTION**

This function initializes the metering parameters for the specified line using the values contained in the Metering Profile. It must 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 20 for more information.

If `pMeterProfile` is `VP_PTABLE_NULL` or `VP_NULL` nothing happens and this function simply returns **VP_STATUS_SUCCESS**.

**RETURNS**

**VP-API-II Return Codes (VpStatusType)**

**EVENTS GENERATED**

None

## 6.2.9 VpInitProfile()

**SYNTAX**

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

**DESCRIPTION**

This function initializes an entry in the device object profile table. The VP-API-II implements *soft* profile tables for the ZL880/miSLIC devices wherein this function simply saves the profile pointer (`pProfile`) for later use. When subsequent VP-API-II function calls reference this profile by its index, the VP-API-II looks-up the saved pointer and reads the profile from Application memory. See *Profile Tables*, on page 20 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 */
```

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

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

***Notes:***

*The Application must not call this function to modify a profile that is currently being used by one or more lines.*

**RETURNS**

**`VP-API-II Return Codes (VpStatusType)`**

**EVENTS GENERATED**

None

CHAPTER

# 7 INTERRUPT AND EVENT HANDLING

## 7.1 OVERVIEW

This chapter discusses how to manage ZL880/miSLIC device and VP-API-II level interrupts.

The functions described in this section are summarized below:

- `VpFlushEvents()` – Flushes all outstanding events.
- `VpGetEvent()` – Returns events corresponding to a device.
- `VpGetResults()` – Reads the data associated with an event.

## 7.2 HANDLING INTERRUPTS FROM ZL880/MISLIC DEVICES

To provide all VP-API-II services, interrupts from the ZL880/miSLIC silicon must be serviced (i.e., `VpGetEvent()` must be called) within 20ms after an interrupt occurs. This can be accomplished in several ways:

1. Call `VpGetEvent()` at a regular rate. This ensures that it will always be called shortly after an interrupt, even without checking if there actually was an interrupt. This is simple to implement but wastes some processing time (the time to call `VpGetEvent()` when there has not been on interrupt since the last call).
2. Check the interrupt pin at a regular rate and call `VpGetEvent()` only if the interrupt pin is active. This provides the same responsiveness as #1 above but with improved processing efficiency.
3. Call `VpGetEvent()` immediately in response to each interrupt. This can provide the best responsiveness and the best accuracy for VP-API-II timed operations. It is the most efficient mode while the device is idle, but can potentially be the most processor intensive when many interrupts occur quickly. If choosing this method, calling `VpGetEvent()` should be done in a deferred interrupt handler rather than directly in an interrupt service routine.

To support compatibility with CSLAC Applications, the ZL880/miSLIC VP-API-II also supports `VpApiTick()` and the associated polling/interrupt handling modes that are familiar to CSLAC customers. These details can be found in *VpApiTick() and Polling/Interrupt Modes*, on page 156.

### 7.2.1 Edge-triggered interrupt handling

Applications which trigger on edges of the interrupt signal, rather than checking the level, may require additional attention.

If a new interrupt is generated at the same time as a previous interrupt is being serviced at the device level, it is possible that the interrupt will appear to remain active, with no detectable edge generated. If this is not dealt with properly, an edge-triggered application will become unresponsive to further device events because there will never be a new interrupt edge to detect.

There are several ways for edge-triggered applications to avoid this problem, including:

1. Define the `VP886_INTERRUPT_EDGETRIG_MODE` flag in vp_api_cfg.h. When this flag is set, the VP-API will perform additional accesses to the device within `VpGetEvent()` to ensure that the interrupt is cleared for long enough to detect an edge.
2. After calling `VpGetEvent()` until it returns `FALSE`, the application can check the status of the interrupt. If it is still active, or if no rising edge has been detected, return to the start of the interrupt handler to call `VpGetEvent()` again.

#### 7.2.1.1 Multiple devices sharing one interrupt line

If multiple SLAC devices share the same interrupt line, solution **1** described above will not be sufficient. Even if the VP-API can be sure that each device's interrupt is cleared when `VpGetEvent()` has finished for that device, it cannot handle the case where an already-serviced device gets a new interrupt while servicing another device. To work properly with multiple devices sharing one interrupt line, the application must check the interrupt status at the end of the handler, as in solution **2** described above.

## 7.3 FUNCTION DESCRIPTIONS

The following sections describe the functions for managing events and event results. Note that since these functions manage existing events, they do not generate new events.

### 7.3.1 VpFlushEvents()

**SYNTAX**

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

**DESCRIPTION**

This function empties the VP-API-II event queue. Pending results (if any) will also be cleared when calling this function. The Application will not have to call `VpGetResults()` to clear the results buffer.

**RETURNS**

`VP-API-II Return Codes (VpStatusType)`

### 7.3.2 VpGetEvent()

**SYNTAX**

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

**DESCRIPTION**

The `VpGetEvent()` function reports a single VTD event.

Note that since `VpGetEvent()` only reports a single event and multiple events can occur at the same time, the Application must call `VpGetEvent()` "immediately" after a previous call that returned `TRUE`. This is to ensure that all events are properly retrieved from the VP-API-II. If events are not retrieved in a timely manner, the VP-API-II will generate the `VP_DEV_EVID_EVQ_OFL_FLT` event.

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

```
typedef struct {
    VpStatusType status;          /* Function return status */
    uint8 channelId;              /* Channel which caused the event */
    VpLineCtxType *pLineCtx;      /* Line that caused the event */
    VpLineIdType lineId;          /* Application set lineId for the line */
    VpDeviceIdType deviceId;      /* Id of the device that caused the event */
    VpDevCtxType *pDevCtx;        /* Device that caused the event */
    VpEventCategoryType
      eventCategory;              /* Event category */
    uint16 eventId;               /* Event ID (within event category) */
    uint16 parmHandle;            /* Event parameter or Application handle */
    uint16 eventData;             /* Data associated with the event */
    bool hasResults;              /* Indicates if event has extra results */
} VpEventType;
```

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

- status == **VP_STATUS_SUCCESS** and **VpGetEvent()** returned TRUE
  An event was retrieved from the VTD, event data valid.
- status == **VP_STATUS_SUCCESS** and **VpGetEvent()** returned FALSE
  No event was retrieved from the VTD, ignore event data.

status != **VP_STATUS_SUCCESS**
**VpGetEvent()** encountered an error. Ignore event data and function return value. See *VP-API-II Function Return Type*, on page 17 for a complete list of error codes.

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

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

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

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

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

Chapter 8, on page 83 describes the eventId for each value of eventCategory, parmHandle, eventData and extended results (hasResults == TRUE) for each VP-API-II event.

***Notes:***

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

**RETURNS**

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

## 7.3.3 VpGetResults()

**SYNTAX**

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

**DESCRIPTION**

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

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

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

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

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

| Function | Event ID | Result Type |
|---|---|---|
| **`VpGetLoopCond()`** | **`VP_LINE_EVID_RD_LOOP`** | **`VpLoopCondResultsType`** |
| **`VpGetOption()`** | **`VP_LINE_EVID_RD_OPTION`** | See Section 10.2.6. |
| **`VpTestLine()`** | **`VP_LINE_EVID_TEST_CMP`** | **`VpTestResultType`** |
| **`VpDeviceIoAccess()`** | **`VP_DEV_EVID_IO_ACCESS_CMP`** | **`VpDeviceIoAccessDataType`** |
| **`VpLineIoAccess()`** | **`VP_LINE_EVID_LINE_IO_RD_CMP`** | **`VpLineIoAccessType`** |
| **`VpCal()`** (see note 2) | **`VP_EVID_CAL_CMP`** | **`uint8p`** |

***Notes:***

1. *When reading options using* **`VpGetOption()`**, *the Application can use the* `eventData` *member of the event structure to determine the option type (**`VpOptionIdType`**) that was read.*
2. *The length of the* `uint8` *buffer to provide to* **`VpGetResults()`** *for event* **`VP_EVID_CAL_CMP`** *(generated as a result of* **`VpCal()`** *with* `calType` = `VP_CAL_GET_LINE_COEFF`*) will be specified by* `eventData` *in the event structure.*
3. *Event* **`VP_LINE_EVID_TEST_CMP`** *will only occur when running Microsemi Line Test API Software. In which case, the Application will not need to call* **`VpGetResults()`**. *It only needs to pass this event to the Line Test API.*

**RETURNS**

**`VP-API-II Return Codes (VpStatusType)`**

CHAPTER

# 8 EVENTS

## 8.1 OVERVIEW

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

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

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

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

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

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

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

**RESULTS** This parameter is identified in the event structure by member: `hasResults`.

Not all events have data associated with them, but those that do fall into two categories:

- Size of the Event Data is <= 16-bit, or
- Size of the Event Data is > 16-bit.

Events that have results larger than 16-bit are indicated by element in the event structure `hasResults = TRUE`. Conversely, events with no results or results <= 16-bit are indicated by `hasResults = FALSE`. Note that the same event may be generated in conditions where it can have results and NOT have results. So the Application must check `hasResults` flag for events that may set `hasResults = TRUE`.

In cases where `hasResults = TRUE` the VP-API-II uses the concept of a *mailbox* to pass this data back to the Application. After detecting `hasResults = TRUE`, the Application must then call **`VpGetResults()`** to retrieve the event data from the mailbox. The type of the result data is described in this section for each event. In cases where the results are a variable length (e.g., **`VP_EVID_CAL_CMP`** when generated for `calType` = `VP_CAL_GET_LINE_COEFF`) the length of the Event Data is specified in the event structure member: `eventData`.

The `VpEventType` definition is provided in function **`VpGetEvent()`**, on page 80.

## 8.2 EVENT SUMMARY

Table 8–1 on page 84 lists all events that the ZL880/miSLIC VP-API-II can generate. The events are organized into categories, and these categories are defined in the software by the `VpEventCategoryType` enumeration indicated in the event structure by element `eventCategory`.

Events are generally either device-specific or 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_`. Only event **`VP_EVID_CAL_CMP`** may occur as a result of device calibration or line calibration.

**Table 8–1 List of VP-API-II Events**

| Event ID | Description | Page |
|---|---|---|
| **Fault Events (eventCategory = VP_EVCAT_FAULT)** | | |
| VP_DEV_EVID_BAT_FLT | Battery fault is detected or is no longer detected. | 86 |
| VP_DEV_EVID_CLK_FLT | Clock fault is detected or is no longer detected. | 86 |
| VP_LINE_EVID_THERM_FLT | Thermal fault is detected or is no longer detected. | 86 |
| VP_LINE_EVID_DC_FLT | DC fault is detected or is no longer detected. | 87 |
| VP_LINE_EVID_AC_FLT | AC fault is detected or is no longer detected. | 87 |
| VP_DEV_EVID_EVQ_OFL_FLT | Event Queue Overflow detected. | 87 |
| VP_LINE_EVID_GND_FLT | Ground Fault is detected or no longer detected. | 88 |
| VP_DEV_EVID_SYSTEM_FLT | System-level fault is detected. | 88 |
| **Signaling Events (eventCategory = VP_EVCAT_SIGNALING)** | | |
| VP_LINE_EVID_HOOK_OFF | The behavior of this event depends on whether pulse-digit decoding is enabled or disabled. | 89 |
| VP_LINE_EVID_HOOK_ON | The behavior of this event depends on whether pulse-digit decoding is enabled or disabled. | 89 |
| VP_LINE_EVID_GKEY_DET | Ring-to-Ground current detected exceeding the user set threshold. | 90 |
| VP_LINE_EVID_GKEY_REL | Ring-to-Ground current less than the user set threshold. | 90 |
| VP_LINE_EVID_FLASH | Hook-Flash was detected. Only occurs if pulse-digit decoding is enabled. | 90 |
| VP_LINE_EVID_STARTPULSE | Initial on-hook indication (i.e., start of dial pulse sequence, flash-hook, or "permanent" on-hook). Occurs only when pulse-digit decoding is enabled. | 90 |
| VP_LINE_EVID_DTMF_DIG | Result of calling **`VpDtmfDigitDetected()`**. | 92 |
| VP_LINE_EVID_PULSE_DIG | Dial Pulse Digit detected. Only occurs if pulse-digit decoding is enabled. | 91 |
| VP_DEV_EVID_TS_ROLLOVER | Occurs every 32.768 seconds. | 91 |
| VP_LINE_EVID_EXTD_FLASH | Hook-Flash exceeding flash-max time was detected. Only occurs if pulse-digit decoding is enabled and `flashHookMax < (onHookMin -1)`. | 91 |

| | | |
|---|---|---|
| VP_LINE_EVID_HOOK_PREQUAL | Occurs when off-hook is detected if pulse-digit decoding is enabled and offHookMin is > 0. See VP_DEVICE_OPTION_ID_PULSE, on page 27 for details. | 92 |
| **Response Events (eventCategory = VP_EVCAT_RESPONSE)** | | |
| VP_LINE_EVID_RD_OPTION | Result of calling VpGetOption(). | 94 |
| VP_EVID_CAL_CMP | VpCalLine() or VpCal() performed successfully. | 93 |
| VP_LINE_EVID_RD_LOOP | Result of calling VpGetLoopCond(). | 95 |
| VP_LINE_EVID_GAIN_CMP | Result of calling VpSetRelGain(). | 96 |
| VP_DEV_EVID_DEV_INIT_CMP | Result of calling VpInitDevice(). | 93 |
| VP_LINE_EVID_LINE_INIT_CMP | Result of calling VpInitLine(). | 93 |
| VP_DEV_EVID_IO_ACCESS_CMP | Result of calling VpDeviceIoAccess(). | 96 |
| VP_LINE_EVID_LINE_IO_RD_CMP | Result of calling VpLineIoAccess() with direction = VP_IO_READ. | 97 |
| VP_LINE_EVID_LINE_IO_WR_CMP | Result of calling VpLineIoAccess() with direction = VP_IO_WRITE. | 97 |
| VP_LINE_EVID_QUERY_CMP | Result of calling VpQuery(). | |
| **Test Events (eventCategory = VP_EVCAT_TEST)** | | |
| VP_LINE_EVID_TEST_CMP | Occurs while running a Line Test API function. | 99 |
| **Process Events (eventCategory = VP_EVCAT_PROCESS)** | | |
| VP_LINE_EVID_MTR_CMP | Metering specified in VpStartMeter() completed successfully. | 100 |
| VP_LINE_EVID_MTR_ABORT | Metering specified in VpStartMeter() aborted prior to completion. | 100 |
| VP_LINE_EVID_CID_DATA | CID Message Buffer activity: {"more data" or "done"} | 100 |
| VP_LINE_EVID_RING_CAD | Indicates start and stop of ringing interval for cadenced ringing. | 101 |
| VP_LINE_EVID_SIGNAL_CMP | Successful completion of signal passed to function VpSendSignal(). | 101 |
| VP_LINE_EVID_TONE_CAD | Successful completion of Tone Cadence passed to VpSetLineTone(). | 102 |
| VP_LINE_EVID_GEN_TIMER | Termination of a user timer provided to VpGenTimerCtrl() | 102 |
| VP_LINE_EVID_USER | Interrupt generated by a non-dedicated I/O line. Event interpretation is Application specific. | 102 |

## 8.3 FAULT EVENTS

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

### 8.3.1 VP_DEV_EVID_BAT_FLT

**DESCRIPTION** This event occurs when a battery fault is detected or is no longer detected.

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

**EVENT DATA** (`eventData`) Event data indicates the source of the battery fault and can be any of the values shown below.

```
Enumeration Data Type: VpBatFltEventDataType:
    /* Generic battery fault codes */
    VP_BAT_FLT_NONE = 0x00,
    VP_BAT_FLT_BAT2 = 0x01,
    VP_BAT_FLT_BAT1 = 0x02,
    VP_BAT_FLT_BAT3 = 0x04,

    /* Fault codes for ZL880/miSLIC */
    VP_BAT_FLT_SWY_OV = 0x02,    /* Switcher Y Over-Voltage */
    VP_BAT_FLT_SWZ_OV = 0x01,    /* Switcher Z Over-Voltage */
    VP_BAT_FLT_CP_UV = 0x04,     /* Charge Pump or VDDSW Under-Voltage */
    VP_BAT_FLT_SWY_OC = 0x08,    /* Switcher Y Over-Current */
    VP_BAT_FLT_SWZ_OC = 0x10,    /* Switcher Z Over-Current */
```

Note that there is overlap between the generic values and the ZL880/miSLIC values. `VP_BAT_FLT_BAT1` is equivalent to `VP_BAT_FLT_SWY_OV`, `VP_BAT_FLT_BAT2` is equivalent to `VP_BAT_FLT_SWZ_OV`, and `VP_BAT_FLT_BAT3` is equivalent to `VP_BAT_FLT_CP_UV`. The ZL880/miSLIC fault code values provide more specific names for the existing generic codes and add support for the over-current fault indications.

**RESULTS** (`hasResults`) `FALSE`

### 8.3.2 VP_DEV_EVID_CLK_FLT

**DESCRIPTION** This event occurs when a clock fault is detected or is no longer detected.

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

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

**RESULTS** (`hasResults`) `FALSE`

### 8.3.3 VP_LINE_EVID_THERM_FLT

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

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

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

**RESULTS** (`hasResults`) `FALSE`

### 8.3.4 VP_LINE_EVID_DC_FLT

**DESCRIPTION**

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

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

Timestamp

**EVENT DATA** (`eventData`)

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

**RESULTS** (`hasResults`)

`FALSE`

### 8.3.5 VP_LINE_EVID_AC_FLT

**DESCRIPTION**

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

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

Timestamp

**EVENT DATA** (`eventData`)

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

**RESULTS** (`hasResults`)

`FALSE`

### 8.3.6 VP_DEV_EVID_EVQ_OFL_FLT

**DESCRIPTION**

This event occurs when the software event queue overflows which results from the host microprocessor failing to retrieve events in a timely manner. In addition to "high bulk call activity" (such that more hook related events are being generated than can be handled), some non-obvious ways the event queue can overflow are:

1. Communication error with the device. If the signaling register reads back as `0xFFFFFFFF` the event queue will likely be overloaded with signaling and fault events.
2. Application calls too many functions that generate immediate events. For example, attempting to add 100 timers with **`VpGenTimerCtrl()`** when only 2 are available will flood the queue with `RESRC_NA` events.
3. Could indicate a VP-API-II design issue.

If a larger ZL880/miSLIC software event queue is needed, this can be accomplished by changing the value of **`VP886_EVENT_QUEUE_SIZE`** found in `vp_api_cfg.h`. Note that this will increase the size of `Vp886DeviceObjectType`.

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

Timestamp

**EVENT DATA** (`eventData`)

N/A

**RESULTS** (`hasResults`)

`FALSE`

### 8.3.7 VP_LINE_EVID_GND_FLT

**DESCRIPTION**

This event occurs when a Ground Fault is detected or is no longer detected. The line may be configured to automatically recheck when the Ground Fault is no longer present without keeping the line in an Active state using **VP_OPTION_ID_GND_FLT_PROTECTION**. See Section 3.3.19 for details.

**T.S. OR HANDLE** (parmHandle)

Timestamp

**EVENT DATA** (eventData)

Event data bit indicates the condition of the "Auto-Ground Fault Protection" algorithm (see Figure 3–3 for details). Possible values are:

- Fault condition is cleared: '0'
- Fault condition is detected: '1'
- Timed out: '2'
  - This means the "Auto-Ground Fault Protection" algorithm terminated because pollNum was reached **AND**
  - The Ground Fault is still active (see Figure 3–3 step 11 and step 12 for details).

**RESULTS** (hasResults)

FALSE

### 8.3.8 VP_DEV_EVID_SYSTEM_FLT

**DESCRIPTION**

This event indicates that a system-level fault was detected.

**T.S. OR HANDLE** (parmHandle)

N/A

**EVENT DATA** (eventData)

The eventData field encodes which faults were detected.

```
Enumeration Data Type: VpSysFltEventDataType:
    VP_SYS_FLT_SPI_ERROR = 0x0001
```

VP_SYS_FLT_SPI_ERROR indicates that an attempt to access a device register failed. This could result in unpredictable behavior. We recommend re-initializing the device at the soonest convenience after a SPI error is detected.

The VP-API attempts to detect SPI errors while polling for interrupts during **VpGetEvent()**. It is possible for SPI errors to occur and not be detected during other operations. Refer to **VP_DEVICE_OPTION_ID_SPI_ERROR_CTRL** for details about how the VP-API responds to SPI errors.

**RESULTS** (hasResults)

FALSE

## 8.4 SIGNALING EVENTS

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

### 8.4.1 VP_LINE_EVID_HOOK_OFF

**DESCRIPTION**

The behavior of this event depends on whether pulse-digit decoding is enabled or disabled. See **VP_DEVICE_OPTION_ID_PULSE, on page 27** and **VP_DEVICE_OPTION_ID_PULSE2, on page 29** for details.

If pulse-digit decoding is enabled, this event occurs when the VTD/VP-API-II determines that the line is off-hook beyond the pulse-digit make period. Therefore, the Application would not observe this event during pulse digit dialing with digit decoding enabled. In this mode, this event indicates the same status as `VP_INPUT_HOOK` retrieved using **VpGetLineStatus()** or **VpGetDeviceStatus()**. `VP_INPUT_RAW_HOOK` will change at the dial pulsed rate.

If pulse-digit decoding is disabled, this event occurs every time the VTD/VP-API-II detects the off-hook condition. Therefore the Application will observe this event during pulse digit dialing with digit decoding disabled. In this mode, this event indicates the same status as `VP_INPUT_HOOK` and `VP_INPUT_RAW_HOOK` retrieved using **VpGetLineStatus()** or **VpGetDeviceStatus()** which will change at the dial pulsed rate.

Note that whether dial pulse detect is enabled or disabled, `VP_INPUT_RAW_HOOK` will always reflect the actual line condition unfiltered by the dial pulse detect state machine.

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

Timestamp

**EVENT DATA** (`eventData`)

N/A

**RESULTS** (`hasResults`)

`FALSE`

### 8.4.2 VP_LINE_EVID_HOOK_ON

**DESCRIPTION**

The behavior of this event depends on whether pulse-digit decoding is enabled or disabled. See **VP_DEVICE_OPTION_ID_PULSE, on page 27** and **VP_DEVICE_OPTION_ID_PULSE2, on page 29** for details.

If pulse-digit decoding is enabled, this event occurs when the VTD/VP-API-II determines that the line is on-hook beyond the pulse-digit break period and hook flash period. In other words, this event does not occur during pulse dialing or a hook-switch flash. The exception is when an invalid pulse train is detected and an on-hook occurs while monitoring the pulse train. In that case, only an on-hook event is generated rather than an invalid digit.

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

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

Timestamp

**EVENT DATA** (`eventData`)

N/A

**RESULTS** (`hasResults`)

`FALSE`

### 8.4.3 VP_LINE_EVID_GKEY_DET

| | |
|---|---|
| **DESCRIPTION** | This event occurs when a ring-to-ground current is detected that exceeds the programmed threshold. Depending on the specific device and other configuration parameters, this even may also occur when a ring-to-battery current is detected. |
| **T.S. OR HANDLE** (parmHandle) | Timestamp |
| **EVENT DATA** (eventData) | N/A |
| **RESULTS** (hasResults) | FALSE |

### 8.4.4 VP_LINE_EVID_GKEY_REL

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

### 8.4.5 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 27 and **VP_DEVICE_OPTION_ID_PULSE2**, on page 29 for details. |
| **T.S. OR HANDLE** (parmHandle) | Timestamp |
| **EVENT DATA** (eventData) | N/A |
| **RESULTS** (hasResults) | FALSE |

### 8.4.6 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 27 and **VP_DEVICE_OPTION_ID_PULSE2**, on page 29 for details. |
| **T.S. OR HANDLE** (parmHandle) | Timestamp |
| **EVENT DATA** (eventData) | N/A |
| **RESULTS** (hasResults) | FALSE |

## 8.4.7 VP_LINE_EVID_EXTD_FLASH

**DESCRIPTION**

This event can only be generated if `EXTENDED_FLASH_HOOK` is set to `#define` in `vp_api_cfg.h`.

This event occurs when an extended flash hook has been detected. This is useful for determining detection of "new call" request (flash duration longer than "hook flash" but less than "on hook"). This event only occurs if pulse-digit decoding is enabled via **`VpSetOption()`**. See **`VP_DEVICE_OPTION_ID_PULSE`**, on page 27 and **`VP_DEVICE_OPTION_ID_PULSE2`**, on page 29 for details.

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

Timestamp

**EVENT DATA** (`eventData`)

N/A

**RESULTS** (`hasResults`)

`FALSE`

## 8.4.8 VP_LINE_EVID_PULSE_DIG

**DESCRIPTION**

This event occurs when a pulse digit is detected. This event only occurs if pulse-digit decoding is enabled via **`VpSetOption()`**. See **`VP_DEVICE_OPTION_ID_PULSE`**, on page 27 and **`VP_DEVICE_OPTION_ID_PULSE2`**, on page 29 for details.

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

'0' if the digit detected meets the parameters specified by **`VP_DEVICE_OPTION_ID_PULSE`**,'1' if the digit detected meets the parameters specified by **`VP_DEVICE_OPTION_ID_PULSE2`**.

**EVENT DATA** (`eventData`)

Event data bits 3 to 0 contain the received digit information, which can be decoded by comparing with the `VpDigitType` enumeration constants. Event data will be `VP_DIG_NONE` if while monitoring the pulse train any digit fails to meet the `breakMin`, `breakMax`, `makeMin`, `makeMax`, or if an on-hook occurs within the `interDigitMin` time specified by **`VP_DEVICE_OPTION_ID_PULSE`** and **`VP_DEVICE_OPTION_ID_PULSE2`**

| D15-D4 | D3-D0 | |
|---|---|---|
| RSVD | PULSE_DIG | |
| | | |
| RSVD = ignore | | |
| PULSE_DIG = digit Detected (of VpDigitType) | | |

**RESULTS** (`hasResults`)

`FALSE`

## 8.4.9 VP_DEV_EVID_TS_ROLLOVER

**DESCRIPTION**

This event occurs every 32.768 seconds.

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

Timestamp

**EVENT DATA** (`eventData`)

N/A

**RESULTS** (`hasResults`)

`FALSE`

## 8.4.10 VP_LINE_EVID_DTMF_DIG

**DESCRIPTION**

This event occurs at the beginning and end of DTMF digit detection if the **VP_OPTION_ID_DTMF_MODE** option is enabled.

Calling **VpDtmfDigitDetected()** will also cause this event to occur. Refer to **VpDtmfDigitDetected(), on page 114** for more information.

**T.S. OR HANDLE** (parmHandle)

Timestamp

**EVENT DATA** (eventData)

Event data bits D3 to D0 contain the received digit information, which can be decoded by comparing with the VpDigitType (see **VpSetLineTone(), on page 109**) enumeration constants. Bit D4 indicates whether this event corresponds to the start (1) or the end (0) of the DTMF digit. Bits D15 through D8 encode the average detection level. The remaining bits D7 to D5 are reserved and should be ignored.

**Table 8–2 DTMF Digit eventData Interpretation**

| D15-D8 | D7-D5 | D4 | D3-D0 |
|---|---|---|---|
| level | reserved | make/break | digit |
| uint8 (see below) | - | VpDigitSenseType | VpDigitType |

The **level** reported in D15-D8 is an unsigned 8-bit integer representing the average level of the detected digit. It is encoded with a step size of 0.25 dB and an offset of -53.75 dB. The range is from +10.0 dB to -53.5 dB. A binary value of 0 means that the level was not measured. In downstream detection mode, the result unit is dBm. In upstream mode, it is dBm0.

Example code to extract the dBm or dBm0 value from eventData:

```
double dBm = ((pEvent->eventData & 0xFF00) >> 8) * 0.25 - 53.75;
```

**RESULTS** (hasResults)

FALSE

## 8.4.11 VP_LINE_EVID_HOOK_PREQUAL

**DESCRIPTION**

This event occurs (with eventData = VP_HOOK_PREQUAL_START) when an off-hook is detected on a line that has VP_OPTION_ID_PULSE_MODE enabled and has been configured with VP_DEVICE_OPTION_ID_PULSE parameters where offHookMin is set > 0. It is normally followed by a VP_LINE_EVID_HOOK_OFF event after the hook qualification interval (offHookMin) expires.

If the phone goes back on-hook before the hook qualification interval expires, another VP_LINE_EVID_HOOK_PREQUAL event occurs, with eventData = VP_HOOK_PREQUAL_ABORT. In this case, no VP_LINE_EVID_HOOK_ON event will be reported.

**T.S. OR HANDLE** (parmHandle)

Timestamp

**EVENT DATA** (eventData)

Event Data indicates whether the phone went off-hook or on-hook.

```
Enumeration Data Type: VpHookPrequalEventDataType:
    VP_HOOK_PREQUAL_START = 0x0000    /* Off-Hook Detected */
    VP_HOOK_PREQUAL_ABORT = 0x0001    /* On-Hook Detected */
```

**RESULTS** (hasResults)

FALSE

## 8.5 RESPONSE EVENTS

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

### 8.5.1 VP_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 69 for information on that function.

This event is non-maskable.

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

Timestamp

**EVENT DATA** (`eventData`)

The `eventData` indicates if the device initialization procedure was completed successfully (`eventData = 0`) or if a failure was detected (`eventData != 0`). When one or more failures are detected, the `eventData` value will be a bit-wise 'OR'ing of the following:

```
VP_DEV_INIT_CMP_SUCCESS     0x0000  /* Success */
VP_DEV_INIT_CMP_FAIL        0x0001  /* Set if any failure occurs */
VP_DEV_INIT_CMP_CFAIL       0x0002  /* Clock Fault Observed */
VP_DEV_INIT_CMP_LINE_FAIL   0x0004  /* Line initialization failed */
VP_DEV_INIT_CMP_CAL_FAIL    0x0008  /* Error during calibration */
VP_DEV_INIT_CMP_VREF_FAIL   0x0010  /* Vref failure */
VP_DEV_INIT_CMP_CP_FAIL     0x0020  /* Charge Pump Failure */
VP_DEV_INIT_CMP_SW_FAIL     0x0040  /* Switcher failure */
VP_DEV_INIT_CMP_CH1_SD      0x0080  /* Channel 1 Shutdown */
VP_DEV_INIT_CMP_CH2_SD      0x0100  /* Channel 2 Shutdown */
VP_DEV_INIT_CMP_CH1_OC      0x0080  /* Channel 1 Over-Current */
VP_DEV_INIT_CMP_CH2_OC      0x0100  /* Channel 2 Over-Current */
VP_DEV_INIT_CMP_CH1_OV      0x0200  /* Channel 1 Over-Voltage */
VP_DEV_INIT_CMP_CH2_OV      0x0400  /* Channel 2 Over-Voltage */
```

**RESULTS** (`hasResults`)

`FALSE`

### 8.5.2 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 71 for information on that function.

This event is non-maskable.

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

Timestamp

**EVENT DATA** (`eventData`)

N/A

**RESULTS** (`hasResults`)

`FALSE`

### 8.5.3 VP_EVID_CAL_CMP

**DESCRIPTION**

This event occurs as a result of calling **`VpCalLine()`** or **`VpCal()`** and indicates that the requested calibration procedure completed successfully. Note that disabling (masking) this event blocks this event for all of the **`VpCalLine()`** and **`VpCal()`** functions.

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

Timestamp

**EVENT DATA** (`eventData`)

N/A

**RESULTS** (`hasResults`)

`TRUE` if **`VpCal()`** is called with `VP_CAL_GET_LINE_COEFF`. `FALSE` otherwise.
See Section 6.2.4 for details.

## 8.5.4 VP_LINE_EVID_RD_OPTION

**DESCRIPTION**

This event occurs as a result of calling `VpGetOption()` and indicates that the VP-API-II has retrieved the requested option setting from the VTD.

This event is non-maskable.

**T.S. OR HANDLE** (parmHandle)

Handle

**EVENT DATA** (eventData)

Event data is of `VpOptionIdType` type and indicates which option was read from the VTD, allowing the Application to correctly interpret the associated results data. See `Option Summary`, on page 23 for the complete list of VP-API-II options.

**RESULTS** (hasResults)

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 3 describes the result type for each VP-API-II option.

## 8.5.5 VP_LINE_EVID_RD_LOOP

**DESCRIPTION**

The event occurs as a result of calling `VpGetLoopCond()` and indicates that the loop condition results are available.

This event is non-maskable.

**T.S. OR HANDLE** (parmHandle)

Handle

**EVENT DATA** (eventData)

N/A

**RESULTS** (hasResults)

The results associated with this event are passed through the `VpLoopCondResultsType` structure:

```
typedef struct {
    int16 rloop;                /* Measured loop resistance */
    int16 ilg;                  /* Longitudinal (common mode) current */
    int16 imt;                  /* 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;
```

The Application can convert the integer results in this structure to real-world values using the “Full Scale” parameters found in Table 8–3. Note that these ranges assume the Application is using the recommended external components.

**Table 8–3 Loop Condition Results Conversion ZL880/miSLIC**

| Loop Condition | ZL880/miSLIC (Full Scale) |
|---|---|
| Loop Resistance | +/-11.67 kΩ |
| Longitudinal Current | +/-59.5 mA |
| Metallic Current | +/-59.5 mA |
| Metallic Voltage | +/-240 V |
| Battery *1 and 2* Voltage | +/-240 V |

Note that values `mspl` and `dcFeedReg` are not used. These will be reported as 0.

These results are based on a single instantaneous reading; no filtering is performed. Values may fluctuate depending on system and line conditions.

`VpLoopCondResultsType` returns the measured battery voltages using the generic battery names `vbat1`, `vbat2`, and `vbat3`. Table 8–4 decodes these generic battery names to device-specific battery names.

**Table 8–4 Battery Name Interpretation**

| Device Type | vbat1 | vbat2 | vbat3 |
|---|---|---|---|
| ZL880/miSLIC Tracker | SWY (ch 0) | SWZ (ch 1) | N/A |
| ZL880/miSLIC ABS | VBL | VBH | N/A |

The `selectedBat` result is of type `VpBatteryType`, and can have a value of `VP_BATTERY_1` or `VP_BATTERY_2`. For tracker configurations, it will always be `VP_BATTERY_1` for channelId 0 and `VP_BATTERY_2` for channelId 1. For ABS configurations, each channel can indicate `VP_BATTERY_1` when using low battery or `VP_BATTERY_2` when using high battery.

## 8.5.6 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 110 for information on that function.

This event is non-maskable.

**T.S. OR HANDLE** (parmHandle)

Handle

**EVENT DATA** (eventData)

N/A

**RESULTS** (hasResults)

Gain adjustment results are returned through the VpRelGainResultsType structure:

```
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.

## 8.5.7 VP_DEV_EVID_IO_ACCESS_CMP

**DESCRIPTION**

This event occurs as a result of calling **VpDeviceIoAccess()** and indicates that the requested I/O access is done.

This event is non-maskable.

**T.S. OR HANDLE** (parmHandle)

Timestamp

**EVENT DATA** (eventData)

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

**RESULTS** (hasResults)

FALSE if used with VP_DEVICE_IO_WRITE.
TRUE if used with VP_DEVICE_IO_READ.

If this event corresponds to a VP_DEVICE_IO_READ operation, the Application should call **VpGetResults()** with a pointer to a VpDeviceIoAccessDataType struct. **VpGetResults()** copies the I/O read results into the deviceIOData_31_0 variable of VpDeviceIoAccessDataType. **VpDeviceIoAccess()**, on page 115 describes how to map the I/O read results in these variables to physical I/O pin logic states.

### 8.5.8 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 116 for information on this function.

This event is non-maskable.

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

Handle

**EVENT DATA** (`eventData`)

N/A

**RESULTS** (`hasResults`)

The Application should call `VpGetResults()` with a pointer to a `VpLineIoAccessType` struct to receive the results associated with this event. See `VpLineIoAccess()`, on page 116 for a description of this struct.

### 8.5.9 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 116 for information on this function.

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

Handle

**EVENT DATA** (`eventData`)

N/A

**RESULTS** (`hasResults`)

`FALSE`

## 8.5.10 VP_LINE_EVID_QUERY_CMP

**DESCRIPTION**

This event occurs as a result of calling **VpQuery()** and indicates that the requested information is available.

**T.S. OR HANDLE** (parmHandle)

Handle

**EVENT DATA** (eventData)

queryId value

**RESULTS** (hasResults)

TRUE

The application should call **VpGetResults()** with a pointer to an instance of VpQueryResultsType, which is a union of all possible query result types.

**Table 8–5 Query Results**

| queryId Value | VpQueryResultsType member | Type |
|---|---|---|
| VP_QUERY_ID_TEMPERATURE | temp | int16 |
| VP_QUERY_ID_SW_DUTY_CYCLE | swDutyCycle | VpSwDutyCycleType |
| VP_QUERY_ID_DEV_TRAFFIC | trafficBytes | uint32 |
| VP_QUERY_ID_LINE_CAL_COEFF | calProf | uint8[256] |

VP_QUERY_ID_TEMPERATURE

The result **temp** provides an uncalibrated measurement of the SLAC device temperature.

```
temperature(°C) = (temp / 32768) * 287 + 153
```

VP_QUERY_ID_SW_DUTY_CYCLE

The **swDutyCycle** result structure provides information about the cycle count and on-time of the switching power supply. The measurement period for these results is a minimum of 7ms, but could be longer due to tick times and interrupt handling latency.

```
typedef struct {
    uint16 onTime;
    uint16 cycleCount;
    uint16 avgDutyCycle;
    bool stateChanged;
} VpSwDutyCycleType;
```

- onTime - The amount of switcher on-time during the measurement period, in 6.144MHz clock ticks. This measurement saturates at 65535.
- cycleCount - The number of switcher cycles which occurred during the measurement period. This measurement saturates at 65535.
- avgDutyCycle - The average switcher on-time percentage during the measurement period, in units of 1/100th of a percent. For example, a 25% average duty cycle would be represented as 2500. If either the onTime or cycleCount is saturated, this result will not be accurate.
- stateChanged - If TRUE, indicates that a line state change occurred during the measurement period, which may make the results unreliable.

For more information about how to interpret these results, please contact a Microsemi applications engineer.

VP_QUERY_ID_DEV_TRAFFIC

See **VpQueryImmediate()**, on page 127

VP_QUERY_ID_LINE_CAL_COEFF

See **VpQueryImmediate()**, on page 127

## 8.6 TEST EVENTS

### 8.6.1 VP_LINE_EVID_TEST_CMP

| | |
|---|---|
| **DESCRIPTION** | The test events occur as a result of the Application calling a Line Test API function. The Application only needs to pass this event to the Line Test API.<br><br>This event is non-maskable. |
| **T.S. OR HANDLE** (parmHandle) | N/A |
| **EVENT DATA** (eventData) | N/A |
| **RESULTS** (hasResults) | Managed by the Line Test Software packages. The Application needs only to handle results generated by the Line Test API. |

## 8.7 PROCESS EVENTS

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

### 8.7.1 VP_LINE_EVID_MTR_CMP

**DESCRIPTION** This event occurs as a result of calling **`VpStartMeter()`** and indicates that the metering signal was generated as requested.

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

**EVENT DATA** (`eventData`) N/A

**RESULTS** (`hasResults`) `FALSE`

### 8.7.2 VP_LINE_EVID_MTR_ABORT

**DESCRIPTION** This event occurs as a result of calling **`VpStartMeter()`** and indicates that the metering signal was aborted before completion.

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

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

**RESULTS** (`hasResults`) `FALSE`

### 8.7.3 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()`**.

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

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

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

The `VP_CID_DATA_NEED_MORE_DATA` event occurs when the VP-API-II can accept 16 bytes of CID message data. This occurs approximately when the VP-API-II is processing it's last 16-bytes of data, giving the Application up to (16*10/1200 = 133.3ms) to respond.

The `VP_CID_DATA_TX_DONE` event occurs when the CID sequence is complete. It occurs for both normal (i.e., when all CID message data has been sent) and "abnormal" termination of CID. Abnormal termination of CID occurs for the following:

- AC, DC or Thermal fault is detected.
- Clock Fault is detected or calling `VpFreeRun(freeRunMode = VP_FREE_RUN_START)`.
- Calling any VP-API-II function that changes the line state (e.g., VpSetLineState(), VpSendSignal()).
- Ground Key is detected.
- Any Hook activity (transition on-hook to off-hook, or off-hook to on-hook) that is not masked by a "Masked-Hook Interval" in the Caller ID Profile.
- The digit(s) in a Caller ID “Detect Interval” were not reported (using **`VpDtmfDigitDetected()`**) by the end of the specified detect interval.
- Start of a new Caller ID sequence by calling **`VpSendCid()`**.
- An unknown instruction is found in the Caller ID Profile. Note that this should only occur if the Caller ID Profile has been manually created or edited, or if a new feature/command exists in the Caller ID Profile that is not supported by the current version of the VP-API-II.

**RESULTS**
`(hasResults)`

`FALSE`

## 8.7.4 VP_LINE_EVID_RING_CAD

**DESCRIPTION**

This event occurs when the VP-API-II/VTD performs automatic ringing cadencing. It notifies the Application of ringing-on and ringing-off state transitions as the VP-API-II/VTD executes the specified cadence. This event does not occur for any line that does not have a ringing cadence applied to it. See **`VpInitRing()`**, on page 75 for information on applying a ringing cadence to a line. This event also does not occur when the ringing cadence is halted because of a state change or ring-trip.

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

Timestamp

**EVENT DATA**
`(eventData)`

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

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

**RESULTS**
`(hasResults)`

`FALSE`

## 8.7.5 VP_LINE_EVID_SIGNAL_CMP

**DESCRIPTION**

This event occurs as a result of calling **`VpSendSignal()`** and indicates that the requested signal was sent.

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

Timestamp

**EVENT DATA**
`(eventData)`

Event data contains a variable of `VpSendSignalType` type, which indicates the type of signal that was sent. This enumeration type is described with **`VpSendSignal()`**, on page 111.

**RESULTS**
`(hasResults)`

`FALSE`

## 8.7.6 VP_LINE_EVID_TONE_CAD

**DESCRIPTION** This event occurs when the VP-API-II performs tone cadencing. It notifies the Application of completion of the cadence. Completion of the cadence occurs when a cadence reaches an "always on", "always off", or end of cadence.

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

**EVENT DATA** (eventData) N/A

**RESULTS** (hasResults) FALSE

## 8.7.7 VP_LINE_EVID_GEN_TIMER

**DESCRIPTION** This event is generated in a response to **VpGenTimerCtrl()**.

**T.S. OR HANDLE** (parmHandle) Handle

**EVENT DATA** (eventData) Contains one of the following values:

```
VP_GEN_TIMER_STATUS_CMP      /* Timer expired successfully */
VP_GEN_TIMER_STATUS_CANCELED /* Timer canceled successfully */
VP_GEN_TIMER_STATUS_RESRC_NA /* Tried to start a new timer, but failed due
                              * to all timer resources in use */
VP_GEN_TIMER_STATUS_UNKNOWN  /* Tried to cancel a timer, but specified an
                              * unknown handle/line combination. */
```

**RESULTS** (hasResults) FALSE

## 8.7.8 VP_LINE_EVID_USER

**DESCRIPTION** This is a line specific event that is generated when I/O2 changes, provided that I/O2 is configured as a digital interrupt signal in the Device Profile. Otherwise, the ZL880/miSLIC VP-API-II will not generate this event.

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

**EVENT DATA** (eventData) The eventData will contain the current state of I/O2: Possible values for eventData are:

```
eventData = 0x1;    /* I/O2 is logic '1' */
eventData = 0x0;    /* I/O2 is logic '0' */
```

**RESULTS** (hasResults) FALSE

CHAPTER

# 9 CONTROL FUNCTIONS

## 9.1 OVERVIEW

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

- `VpSetLineState()` – Sets a line to the requested state.
- `VpSetRelayState()` – Sets the line relay configuration.
- `VpSetOption()` – Sets various device and line specific options.
- `VpSetLineTone()` – Generates a call progress tone on the line. The tone provided may be cadenced or steady using this function.
- `VpSetRelGain()` – Sets the relative transmit or receive gain for a line.
- `VpStartMeter()` – Starts metering on the line.
- `VpSendSignal()` – Generates a DC signature signal on the line (e.g., Message Waiting Pulse, Forward Disconnect).
- `VpSendCid()` – Immediately starts a Caller ID sequence on the line. This function is most often used for Type II Caller ID (non-Ringing associated Caller ID).
- `VpContinueCid()` – Refreshes the Caller ID buffer for the line during message transmission.
- `VpDtmfDigitDetected()` – Reports a DTMF digit detected outside the scope of the VP-API-II. Used for implementing Type-II Caller ID.
- `VpGenTimerCtrl()` - Provides control of the general purpose timers.
- `VpDeviceIoAccess()` – Controls device input/output pins.
- `VpLineIoAccess()` – Controls input/output pins of a specific line.
- `VpFreeRun()` – Puts the device/lines into and takes it out of PLL Free Run mode.
- `VpBatteryBackupMode()` – Adjusts power supply timing parameters to maintain functionality in a battery backup mode.
- `VpShutdownDevice()` – Shuts down the device.

## 9.2 FUNCTION DESCRIPTIONS

### 9.2.1 VpSetLineState()

**SYNTAX**

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

**DESCRIPTION**

This function sets the line to the requested line state. ZL880/miSLIC supported line states are listed below.

| Enumeration Type VpLineStateType: | | | |
|---|---|---|---|
| | PCM Highway | CODEC | DC Feed |
| VP_LINE_STANDBY | Disabled | Disabled | Normal Voltage Feed.<br>Low power on-hook state. |
| VP_LINE_STANDBY_POLREV | Disabled | Disabled | Reverse Voltage Feed<br>Low power on-hook state. |
| VP_LINE_TIP_OPEN | Disabled | Disabled | Battery on Ring Lead<br>Tip in High Impedance State<br>Ground Start or Ground Key Idle State. |
| VP_LINE_RING_OPEN | Disabled | Disabled | Battery on Tip Lead<br>Ring in High Impedance State |
| VP_LINE_ACTIVE | Disabled | Active | Normal current feed off-hook state. Can be used for sending Call Progress Tones. |
| VP_LINE_ACTIVE_POLREV | Disabled | Active | Reverse current feed off-hook state. Can be used for sending Call Progress Tones. |
| VP_LINE_TALK | Enabled | Active | Normal current feed off-hook state typically used for voice conversation. |
| VP_LINE_TALK_POLREV | Enabled | Active | Reverse current feed off-hook state typically used for voice conversation. |
| VP_LINE_OHT | Enabled | Active | Normal current feed on-hook state typically used for Type I Caller ID or other on-hook transmission. |
| VP_LINE_OHT_POLREV | Enabled | Active | Reverse current feed on-hook state typically used for Type I Caller ID or other on-hook transmission. |
| VP_LINE_DISCONNECT | Disabled | Disabled | Tip and Ring in high impedance state. Battery is enabled. |
| VP_LINE_DISABLED | Disabled | Disabled | Tip and Ring in high impedance state. Battery is disabled. Must call VpInitDevice() to recover. |
| VP_LINE_RINGING | Disabled (*) | Disabled (**) | Signal defined by last Ringing Profile provided. DC Bias is (Tip - Ring). |
| VP_LINE_RINGING_POLREV | Disabled (*) | Disabled (**) | Signal defined by last Ringing Profile provided. DC Bias is (Ring - Tip). |
| VP_LINE_HOWLER | Disabled | Active | Normal polarity off-hook feed state. D-A gain increase by 11.5 compared to 0dBr. Used for generating Howler Tones. |
| VP_LINE_HOWLER_POLREV | Disabled | Active | Reverse polarity off-hook feed state. D-A gain increase by 11.5 compared to 0dBr. Used for generating Howler Tones. |

**(*) PCM Highway and CODEC can be enabled in Ringing Cadence by using a VP-API-II line state in the silent interval that does enables the CODEC and/or PCM Highway. This can also be accomplished using an appropriate Caller ID Profile applied during Ringing.**

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

In addition to the Ringing Profile and Ringing Cadence Profile, parameters affecting ring entry and exit are managed using **VP_OPTION_ID_RING_CNTRL**, on page 31.

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

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

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

***Notes:***

1. **VpInitDevice()** *and* **VpInitLine()** *places all FXS lines in* VP_LINE_DISCONNECT *state. The Application must call this function after initialization to enable service to the line.*
2. *A new line state requested by the Application is considered a higher priority than the current state, which includes cadences requiring the D/A converter. Active cadences that conflict with the new line state will be terminated. This includes, but not limited to, Caller ID, Ringing Cadence, Metering, and signals generated by* **VpSendSignal()**. *Tone cadences will continue provided that the new state is one where the D/A converter is enabled.*
3. *If metering is active the current meter pulse will be terminated immediately. Applications that prefer to stop metering after the current meter pulse should call* **VpStartMeter()** *with* numMeters *set to 0 and wait for the* **VP_LINE_EVID_MTR_ABORT** *event prior to calling this function.*
4. *The application may only set the line to VP_LINE_DISABLED on Tracker devices*

**RETURNS**

**VP-API-II Return Codes (VpStatusType)**

**EVENTS GENERATED**

If the line supports ringing cadence, then the following events may be generated as result of setting the line to VP_LINE_RINGING or VP_LINE_RINGING_POLREV:

```
VP_LINE_EVID_RING_CAD  /* Indicates Ringing On/Of instances */
VP_LINE_EVID_CID_DATA  /* Occurs during Caller ID Data Transmission */
```

Some events are generated also when running an existing cadence that is aborted due to calling a conflicting line state (i.e., a line state that prevents the running sequence from continuing). These conditions are mentioned in the ***Notes*** section and specific functions and events that are affected.

## 9.2.2 VpSetRelayState()

**SYNTAX**

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

**DESCRIPTION**

This function configures the VTD-controlled relays. 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 66 through the line termination type). The supported relay states are listed below.

```
Enumeration Data Type: VpRelayControlType:
    VP_RELAY_NORMAL
    VP_RELAY_BRIDGED_TEST
```

The `VP_RELAY_NORMAL` state allows for normal VTD control of the relays. The actual state of a relay in this case may depend on the line state set by the **VpSetLineState()** or fault conditions detected. Selection of any other relay state overrides the automatic VTD relay control.

The `VP_RELAY_BRIDGED_TEST` state provides an internal test termination on Tip-Ring. Note that although the termination appears in parallel with Tip/Ring leaving Tip/Ring connected, the internal termination is a very low resistance virtually eliminating the Tip/Ring connection.

The appendix **Relay Configurations**, on page 159 describes simple connection diagrams for the relay states for various line termination types supported by the ZL880/miSLIC VP-API-II.

**RETURNS**

**VP-API-II Return Codes (VpStatusType)**

**EVENTS GENERATED**

None

## 9.2.3 VpSetOption()

**SYNTAX**

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

**DESCRIPTION**

This function sets an option to the specific value for one or more lines. The `option` argument determines which option is modified. Options may be line-specific or device-specific. Refer to Chapter 3, on page 23 for a complete list and definition of all VP-API-II options. For convenience, included here is the link to: **`VpGetOption()`**

***Notes:***

**`VP_OPTION_ID_DEBUG_SELECT`** *is the only option that can be set with a line context, a device context or neither. See* Chapter 13 *for details on VP-API-II Debug configuration and mechanisms.*

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

**Table 9–1 `VpSetOption()` Behavior**

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

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

The arguments required for each option are different. Therefore, a `void` pointer (`pValue`) is provided as the option input parameter to the function. This argument must point to an initialized instance of the input structure related to the target option. For example, if the device critical fault options are being set, then `pValue` must point to an initialized instance of `VpOptionCriticalFltType`. Chapter 3, on page 23 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()**, on page 69. 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 except for **VP_OPTION_ID_EVENT_MASK** 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**

**VP-API-II Return Codes (VpStatusType)**

**EVENTS GENERATED**

None.

## 9.2.4 VpSetLineTone()

**SYNTAX**

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

**DESCRIPTION**

This function starts a call progress or DTMF tone with an optional cadence on the line. The generated tone is defined by `pToneProfile` and `pDtmfControl`. If both of these are set to `VP_NULL` then any currently active tone is stopped.

The Cadence Profile (`pCadProfile`) specifies the on/off times of the tones. If `pCadProfile` is `VP_PTABLE_NULL` or `VP_NULL` then the tones are played continuously.

The argument `pDtmfControl` controls DTMF tone generation, but is a lower priority than Tone Generation (`pToneProfile`). DTMF generation will only occur if `pToneProfile` is `VP_PTABLE_NULL` or `VP_NULL` and `pDtmfControl != VP_NULL`. The DTMF control structure is defined below.

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

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

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

**Generating Special Howler Tones:**

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

1. *UK: BTNR 1080 Version 15 and Draft 960-G*
2. *AUS: Australia ACIF S002:2001*
3. *NTT: NTT Edition 5*

Selection of a "Special Howler Tone" is done in the Tone Cadence Profile generated by Profile Wizard. When generating a "Special Howler Tone", the Tone Profile (`pToneProfile`) is ignored but cannot be `VP_PTABLE_NULL` or `VP_NULL`.

***Notes:***

1. *All tone generation is supported in the D-A direction only.*
2. *The total level increase for NTT Howler Tone will be over the duration of [14 to 15] seconds.*
3. *When used with line states `VP_LINE_HOWLER` or `VP_LINE_HOWLER_POLREV` the maximum output level of the special howler tones is: UK (14.5dBm), NTT (14.5dBm), AUS (10.0dBm) when measured into a 600ohm load. When used with talk line states, the maximum output level is +3dBm using a standard AC Profile and setting the maximum receive (D-A) gain of 0dBr.*

**RETURNS**

**VP-API-II Return Codes (VpStatusType)**

**EVENTS GENERATED**

**VP_LINE_EVID_TONE_CAD**, on page 102

## 9.2.5 VpSetRelGain()

**SYNTAX**

```
VpStatusType
VpSetRelGain(
 VpLineCtxType *pLineCtx,     /* I/O: Pointer to line context */
 uint16 txLevel,              /* Input: Adjustment relative to default A-D level */
 uint16 rxLevel,              /* Input: Adjustment relative to default D-A level */
 uint16 handle)               /* Input: Handle returned with event */
```

**DESCRIPTION**

**This function should not be used in most Applications. When gain setting is required, the option VP_OPTION_ID_ABS_GAIN is usually preferred.**

This function adjusts the transmit and receive relative gain for the specified line. Proper use of this function requires coefficients provided by Microsemi.

The gain adjustment is defined as a voltage gain (multiplier) made relative to the gain levels set in the AC Profile applied to the line. Setting the txLevel or rxLevel to 1.0 sets the respective path to the default gain from the AC Profile.

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

Generic Equation:

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

converting for level results in the equation:

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

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

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

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

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

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

The VP_LINE_EVID_GAIN_CMP event occurs once the VTD has applied the new gain settings. The results of this function are described in Section 8.5.6 with the definition of this event.

**RETURNS**

VP-API-II Return Codes (VpStatusType)

**EVENTS GENERATED**

VP_LINE_EVID_GAIN_CMP, on page 96

## 9.2.6 VpSendSignal()

**SYNTAX**

```
VpStatusType
VpSendSignal(
 VpLineCtxType *pLineCtx,          /* I/O: Pointer to line context */
 VpSendSignalType signalType,      /* Input: Specifies the type of signal */
 void *pSignalData)                /* Input: 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 */
    VP_SENDSIG_FWD_DISCONNECT     /* Generate a Forward Disconnect */
    VP_SENDSIG_POLREV_PULSE      /* Generate a Polarity Reversal */
    VP_SENDSIG_TIP_OPEN_PULSE     /* Generate a Tip Open Pulse */
```

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

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

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

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

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

**RETURNS**

**`VP-API-II Return Codes (VpStatusType)`**

**EVENTS GENERATED**

**`VP_LINE_EVID_SIGNAL_CMP`**, on page 101

## 9.2.7 VpSendCid()

**SYNTAX**

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

**DESCRIPTION**

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

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

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

The `pCidData` argument should point to a buffer containing the Caller ID message. Refer to **VpInitCid()**, on page 76 for move information on handling the Caller ID data.

**RETURNS**

**VP-API-II Return Codes (VpStatusType)**

**EVENTS GENERATED**

**VP_LINE_EVID_CID_DATA**, on page 100

## 9.2.8 VpContinueCid()

**SYNTAX**

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

**DESCRIPTION**

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

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

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

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

**RETURNS**

**VP-API-II Return Codes (VpStatusType)**

**EVENTS GENERATED**

**VP_LINE_EVID_CID_DATA**, on page 100

## 9.2.9 VpStartMeter()

**SYNTAX**

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

**DESCRIPTION**

This function manages metering pulses on the line. 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 76. 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**

**VP-API-II Return Codes (VpStatusType)**

**EVENTS GENERATED**

**VP_LINE_EVID_MTR_CMP**, on page 100
**VP_LINE_EVID_MTR_ABORT**, on page 100

## 9.2.10 VpDtmfDigitDetected()

**SYNTAX**

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

**DESCRIPTION**

This function is used to support VP-API-II Type-II Caller ID.

The Type II Caller ID Sequence (in many regions) requires detection of a DTMF digit within a certain interval as an acknowledgement that the CPE is ready to receive the Caller ID message. This interval is referred to in the Caller ID Profile as the "*Detect Interval*". Since the ZL880/miSLIC silicon itself does not detect DTMF, it must therefore rely on external DTMF detection resources in order to support the "*Detect Interval*" required for Type II Caller ID.

The ZL880/miSLIC VP-API-II will efficiently use the external DTMF detection resources by calling **VpSysDtmfDetEnable()** at the start of the "*Detect Interval*" and **VpSysDtmfDetDisable()** at the end of the "*Detect Interval*". See **VpSysDtmfDetEnable() and VpSysDtmfDetDisable()**, on page 131 for further information about these functions.

During the "*Detect Interval*", the Application should call **VpDtmfDigitDetected()** at the leading edge and trailing edge of a DTMF digit detection. At the end of the "*Detect Interval*" the VP-API-II in the Caller ID sequence will determine if the DTMF digits detected during the interval (if any) match the digits specified in the Caller ID profile. If a match is found, Caller ID will proceed. If not, Caller ID will abort.

The `digit` argument indicates which DTMF digit was detected and is of the **VpDigitType** type described in **VpSetLineTone()**, on page 109.

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 */
```

Note that calling this function results in the **VP_LINE_EVID_DTMF_DIG** event being generated. This is primarily done to be compatible with Microsemi devices (using VP-API-II) that do support DTMF detection.

**RETURNS**

**VP-API-II Return Codes (VpStatusType)**

**EVENTS GENERATED**

**VP_LINE_EVID_DTMF_DIG**, on page 92

## 9.2.11 VpDeviceIoAccess()

**SYNTAX**

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

**DESCRIPTION**

This function accesses the device input/output (I/O) pins of the VTD. Refer to VP_DEVICE_OPTION_ID_DEVICE_IO, on page 34 for 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. It can take one of the following values:

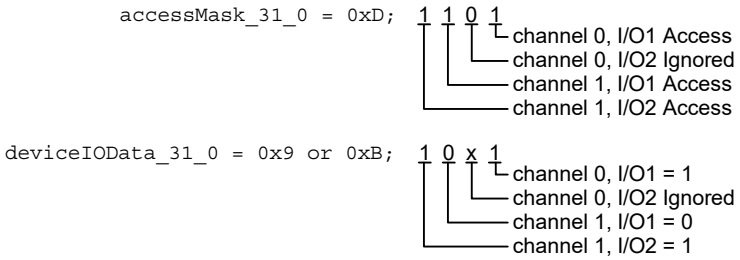
```
Enumeration Data Type: VpDeviceIoAccessType:
    VP_DEVICE_IO_WRITE           /* Perform device I/O write access */
    VP_DEVICE_IO_READ            /* Perform device I/O read access */
```

The ZL880/miSLIC silicon only supports 2 channels, each with 2 I/O lines. Therefore, the ZL880/miSLIC VP-API-II will ignore accessMask_63_32 and deviceIOData_63_32. It will also ignore all except the lower 4-bits each of accessMask_31_0 and deviceIOData_31_0.

Each of the lower 4-bits in accessMask_31_0 determines whether an individual pin is accessed (1) or ignored (0). During an I/O write operation, the state of any pin with its accessMask bit set to 0 remains unchanged. For the ZL880/miSLIC silicon, accessMask[2N] sets the direction for I/O Pin 1 belonging to channel N, and accessMask[2N+1] sets the direction for I/O Pin 2 belonging to channel N.

Each of the lower 4-bits in deviceIOData_31_0 determines the state of an individual output pin when a write operation is performed. The deviceIOData field is mapped to I/O pins in the same manner as the accessMask field described above. Note that deviceIOData is ignored if accessType is VP_DEVICE_IO_READ.

Example: Setting channel 0 I/O1 =1, channel 0 I/O2 = ignored, channel 1 I/O1 = 0, channel 1 I/O2 = 1




**RETURNS**

VP-API-II Return Codes (VpStatusType)

**EVENTS GENERATED**

VP_DEV_EVID_IO_ACCESS_CMP, on page 96

## 9.2.12 VpLineIoAccess()

**SYNTAX**

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

**DESCRIPTION**

This function accesses some or all of the GPIO pins associated with a particular line. Refer to VP_OPTION_ID_LINE_IO_CFG, on page 36 for information on I/O pin configuration and restrictions.

This function takes a pointer to the VpLineIoAccessType structure defined below:

```
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 one of the following values:

```
Enumeration Data Type: VpIoDirectionBits:
    VP_IO_WRITE
    VP_IO_READ
```

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_LINE_IO_RD_CMP event only 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.

**RETURNS**

VP-API-II Return Codes (VpStatusType)

**EVENTS GENERATED**

VP_LINE_EVID_LINE_IO_RD_CMP, on page 97
VP_LINE_EVID_LINE_IO_WR_CMP, on page 97

## 9.2.13 VpGenTimerCtrl()

**SYNTAX**

```
VpStatusType
VpGenTimerCtrl(
 VpLineCtxType *pLineCtx,          /* I/O: Pointer to line context */
 VpGenTimerCtrlType timerCtrl,     /* Input: Details below */
 uint32 duration,                  /* Input: Time specified in ms (if
                                   starting) */
 uint16 handle)                    /* Input: Handle value returned with
                                   event */
```

**DESCRIPTION**

This function provides control of the general-purpose timers managed by the ZL880/miSLIC VP-API-II. Timers can be started and stopped (cancelled) using the **VpGenTimerCtrlType** enumeration below:

```
typedef enum {
    VP_GEN_TIMER_START
    VP_GEN_TIMER_CANCEL
} VpGenTimerCtrlType;
```

If canceling a timer you must pass the same `pLineCtx` and `handle` parameters as when you started the timer.

The event **VP_LINE_EVID_GEN_TIMER** is generated when a timer provided to this function is terminated. If using multiple timers on the same line, the Application can determine the timer that expired by assigning a unique 16-bit value to `handle` when this function is called and comparing that to the value of `parmHandle` provided with the **VP_LINE_EVID_GEN_TIMER** event.

The ZL880/miSLIC VP-API-II provides 2 general-purpose timers per-device that are managed by this function. This number can be modified by changing the value of `VP886_USER_TIMERS` (found in `vp_api_cfg.h`). Increasing this value will increase the size of `Vp886DeviceObjectType`. Setting this value to (0) will remove this function from the ZL880/miSLIC VP-API-II.

**RETURNS**

**VP_STATUS_SUCCESS**: timer started or cancelled per request.
**VP_STATUS_DEVICE_BUSY**: request for a new timer when all existing timers are in use.
**VP_STATUS_FUNC_NOT_SUPPORTED**: `VP886_USER_TIMERS` is set to 0.
**VP_STATUS_INVALID_ARG**: error in the arguments provided.

**EVENTS GENERATED**

**VP_LINE_EVID_GEN_TIMER**, on page 102

## 9.2.14 VpFreeRun()

**SYNTAX**

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

**DESCRIPTION**

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

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

PLL Free Run occurs when a Clock Fault is detected. A Clock Fault is detected either by missing or incorrect PCLK, or missing or incorrect FS. When in Clock Fault mode, the silicon switches it's PLL to run on an without a reference clock (but continues to run) and eventually slows to ~1/5th the normal rate. If allowed to run at the slower rate, the switcher and line response will be affected.

To minimize customer impact in the event of a Clock Fault, the VP-API-II will automatically enter PLL Free Run Mode when a Clock Fault is detected. If a PCLK fault can be anticipated, the Application should call this function with `freeRunMode = VP_FREE_RUN_START` prior to PCLK removal. Note that there is a significant difference between the VP-API-II automatically entering Free Run mode compared to being told to enter Free Run mode by the Application:

1. *If automatically entering Free Run mode by the VP-API-II (when Clock Fault is detected), the VP-API-II will exit Free Run mode when:*
   a) *The Clock Fault is removed.*
   b) *The Application calls* **VpInitDevice()**.
   c) *The Application overrides the VP-API-II Free Run Mode by first calling this function with* `freeRunMode = VP_FREE_RUN_START` *then* `freeRunMode = VP_FREE_RUN_STOP` *when the Clock Fault has been removed.*
2. *If put into Free Run mode by the Application (even if done after the VP-API-II has automatically entered Free Run mode), then the VP-API-II will remain in Free Run mode until one of the following occur:*
   a) *The Application calls this function with* `freeRunMode = VP_FREE_RUN_STOP`.
   b) *The Application calls* **VpInitDevice()**.
3. *If put into Free Run mode by the Application, the VP-API-II will mask* **VP_DEV_EVID_CLK_FLT** *and* **VP_DEV_EVID_TS_ROLLOVER** *events to avoid interrupting sleep modes.*

**RETURNS**

**VP-API-II Return Codes (VpStatusType)**

**EVENTS GENERATED**

None

*Note that while calling this function does not generate an event, the Clock Fault event described in Section 8.3.2 can be used to determine if the VP-API-II is operating in Free Run mode or not. If not using events, Clock Fault status can be determined at any time using either* **VpGetLineStatus()** or **VpGetDeviceStatus()**.

## 9.2.15 VpBatteryBackupMode()

**SYNTAX**

```
VpStatusType
VpBatteryBackupMode(
 VpDevCtxType *pDevCtx,                    /* I/O: Device context pointer */
 VpBatteryBackupModeType backupMode,       /* Input: Mode flag */
 uint8 vsw)                                /* Input: New power supply input
                                              voltage in 0.2V steps */
```

**DESCRIPTION**

This function should be called if the power supply input voltage needs to change due to entering or exiting a battery backup mode. The switching regulator parameters will be changed to adapt to the new input voltage and maintain functionality.

Enumeration Data Type: **VpBatteryBackupModeType**:
VP_BATT_BACKUP_ENABLE
VP_BATT_BACKUP_DISABLE

The **vsw** argument is the new input voltage, in 0.2V steps. If the new voltage is 5V, the **vsw** value should be 25.

This function will also apply new switcher timing parameters for battery backup mode if they are included in the device profile which was used in **VpInitDevice()**. Please contact a Microsemi applications engineer to design an appropriate device profile which contains these timing parameters.

**RETURNS**

**VP-API-II Return Codes (VpStatusType)**

**EVENTS GENERATED**

None

## 9.2.16 VpShutdownDevice()

**SYNTAX**

```
VpStatusType
VpShutdownDevice(
 VpDevCtxType *pDevCtx)            /* I/O: Pointer to device context */
```

**DESCRIPTION**

This function puts all lines into the VP_LINE_DISABLED state, shuts down all power supplies, stops all timers, and masks all interrupts for the specified device. The device must be reinitialized with **VpInitDevice()** to be used again.

**RETURNS**

**VP-API-II Return Codes (VpStatusType)**

**EVENTS GENERATED**

None

CHAPTER

# 10 STATUS AND QUERY FUNCTIONS

## 10.1 OVERVIEW

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

- `VpGetLoopCond()` – Reads loop and battery conditions for the line.
- `VpGetLineStatus()` – Returns boolean type of status about a particular line.
- `VpGetDeviceStatus()` – Returns the state of a particular status for all lines of the device.
- `VpGetLineInfo()` – Retrieves line-specific information from a device or line context.
- `VpGetDeviceInfo()` – Retrieves device-specific information from a device or line context.
- `VpGetOption()` – Returns the current setting of an option.
- `VpGetOptionImmediate()` – Returns the current setting of an option without waiting for an event.
- `VpGetLineState()` – Reads the current line state.
- `VpQueryImmediate()` – Returns various non-boolean information about a line or device which does not require waiting for an event.
- `VpQuery()` – Returns variuous non-boolean information about a line or device. Requires waiting for an event.

## 10.2 FUNCTION DESCRIPTIONS

### 10.2.1 VpGetLoopCond()

**SYNTAX**

```
VpStatusType
VpGetLoopCond(
  VpLineCtxType *pLineCtx,      /* I/O: Pointer to line context */
  uint16 handle)                /* Input: Handle value returned with event */
```

**DESCRIPTION**

This function reads the current loop and battery conditions for the specified line. Each reported value is taken as a non-filtered single point measurement. So repeated calls to this function can show varying results depending on system and line conditions.

When all measurements have been taken, the `VP_LINE_EVID_RD_LOOP` event occurs with `hasResults = TRUE`. The Application then has to call `VpGetResults()` to retrieve the results. See `VP_LINE_EVID_RD_LOOP`, on page 95 for details.

This function is non-intrusive, and can therefore be used with the line in any state.

**RETURNS**

`VP-API-II Return Codes (VpStatusType)`

**EVENTS GENERATED**

`VP_LINE_EVID_RD_LOOP`, on page 95

## 10.2.2 VpGetLineStatus()

**SYNTAX**

```
VpStatusType
VpGetLineStatus(
 VpLineCtxType *pLineCtx,     /* I/O: Pointer to line context */
 VpInputType input            /* Input: Status type being requested */
 bool *pStatus)               /* Output: 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**:

| Status types | Description |
|---|---|
| VP_INPUT_HOOK | /* Hook Status (ignoring pulse & flash) */ |
| VP_INPUT_RAW_HOOK | /* Hook Status (include pulse & flash) */ |
| VP_INPUT_GKEY | /* Ring-to-Ground current status */ |
| VP_INPUT_THERM_FLT | /* Thermal Fault Status */ |
| VP_INPUT_CLK_FLT | /* Clock Fault Status */ |
| VP_INPUT_AC_FLT | /* AC Fault Status */ |
| VP_INPUT_BAT1_FLT | /* Battery 1 Fault Status */ |
| VP_INPUT_BAT2_FLT | /* Battery 2 Fault Status */ |
| VP_INPUT_BAT3_FLT | /* Battery 3 Fault Status */ |
| VP_INPUT_DC_FLT | /* Longitudinal current from Ring-to-Battery. */ |

Considerations for reported line status:

**VP_INPUT_HOOK, VP_INPUT_RAW_HOOK:**

- Indicates `TRUE` for loop close detection and `FALSE` for loop open detection.
- When VP-API-II pulse detection is disabled (i.e., **VP_OPTION_ID_PULSE_MODE** set to `VP_OPTION_PULSE_DECODE_OFF`) `VP_INPUT_HOOK` and `VP_INPUT_RAW_HOOK` are identical.
- When VP-API-II pulse detection is enabled (i.e., **VP_OPTION_ID_PULSE_MODE** set to `VP_OPTION_PULSE_DECODE_ON`) `VP_INPUT_RAW_HOOK` reflects the instantaneous state of the line's hook detector and will toggle at the dial pulse/flash-hook rate. `VP_INPUT_HOOK` on the other hand reflects the overall state of hook detector and will remain steady during dial pulse/flash-hook activity. `VP_INPUT_HOOK` will change from `FALSE` (on-hook) to `TRUE` (off-hook) at the start of the initial off-hook detection and will continue to report `TRUE` until the line has detected on-hook for the time specified by `flashHookMax` if `EXTENDED_FLASH_HOOK` is set to `#undef`, or `onHookMin` if `EXTENDED_FLASH_HOOK` is set to `#define`.

**VP_INPUT_CLK_FLT**

- Indicates `TRUE` for Clock Fault present, `FALSE` when Clock Fault has been removed.
- When the device is in Clock Fault, it will be running in Free Run mode as described in Section 9.2.1.

***Notes:***

1. *Applications that do not support Ring-to-Ground currents as a normal part of the call processing (e.g., Applications that do not support Ground Start Signaling) should configure the DC Profile "Reporting Behavior" as "DC Fault: Report longitudinal currents as DC Faults.." and monitor `VP_INPUT_DC_FLT`.*
2. *See Table 8–4 to convert generic battery names to device-specific battery names.*

**RETURNS**

**VP-API-II Return Codes (VpStatusType)**

**EVENTS GENERATED**

None

### 10.2.3 VpGetDeviceStatus()

**SYNTAX**

```
VpStatusType
VpGetDeviceStatus(
  VpDevCtxType *pDevCtx,          /* I/O: Pointer to device context */
  VpInputType input,              /* Input: Test the status of this input
                                     type */
  uint32 *pDeviceStatus)          /* Output: Pointer to status results */
```

**DESCRIPTION**

This function returns the status of the requested `input` for both lines the device. Each bit in the result represents the status of `input` for one line. The status result is written to the location pointed to by `pDeviceStatus`. The least significant bit represents line 1, the next successive bit represents line 2. The possible values for `input` are specified in **`VpGetLineStatus()`**. Refer to **`VpGetLineStatus()`**, on page 122 for more details.

**RETURNS**

**`VP-API-II Return Codes (VpStatusType)`**

**EVENTS GENERATED**

None

### 10.2.4 VpGetLineInfo()

**SYNTAX**

```
VpStatusType
VpGetLineInfo(
  VpLineInfoType *pLineInfo)       /* I/O: 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 provided 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**

**`VP-API-II Return Codes (VpStatusType)`**

**EVENTS GENERATED**

None

## 10.2.5 VpGetDeviceInfo()

**SYNTAX**

```
VpStatusType
VpGetDeviceInfo(
  VpDeviceInfoType
*pDeviceInfo)                          /* I/O: Pointer to device info */
```

**DESCRIPTION**

This function returns information about a device. The following structure is defined for use with this function:

```
typedef struct  {
    VpLineCtxType *pLineCtx;     /* Pointer to Line Context */
    VpDeviceIdType deviceId;     /* Device identity */
    VpDevCtxType *pDevCtx;       /* Pointer to device Context */
    VpDeviceType deviceType;     /* Device Type */
    VpFeatureListType featureList;    /* Silicon features available */
    uint8 numLines;         /* Number of lines */
    uint8 revCode;          /* Silicon revision of the device */
    uint16 productCode;     /* Silicon Product Code Number */
    uint32 intProductCode; /* Internal product code */
} VpDeviceInfoType;

typedef struct  {
    VpFeatureType testLoadSwitch;          /* External Test Load */
    VpFeatureType internalTestTermination; /* Internal Test Load */
} VpFeatureListType;

Enumeration Data Type: VpFeatureType:
  VP_AVAILABLE
  VP_NOT_AVAILABLE
```

The ZL880/miSLIC devices only support the (internal) `internalTestTermination`.

This function can be used in the following two ways:

1. *If the pointer to the line context (*`pDeviceInfo->pLineCtx`*) is not* `VP_NULL`*, then this function returns information about the device associated with the given line context. It fills all other elements in the* `VpDeviceInfoType` *struct. The identity of the device is stored in the* `deviceId` *field, a pointer to device context is stored in the* `pDevCtx` *field, the type of device is stored in the* `deviceType` *field, and the number of lines supported by the device is stored in the* `numLines` *field.*
2. *If the pointer to the line context is* `VP_NULL` *and the pointer to the device context (*`pDeviceInfo->pDevCtx`*) is not* `VP_NULL`*, then this function returns other details like device identity, device type, and the number of lines supported by the device. It stores this information in their respective fields. No information is written to the line context. Note that the Application can use this function in this mode to learn the number of lines supported by the device before creating line objects.*

This function returns an error if either `pDeviceInfo` is `VP_NULL` or if both the line context pointer and the device context pointer are `VP_NULL`.

**RETURNS**

**VP-API-II Return Codes (VpStatusType)**

**EVENTS GENERATED**

None

## 10.2.6 VpGetOption()

**SYNTAX**

```
VpStatusType
VpGetOption(
  VpLineCtxType *pLineCtx,      /* I/O: Pointer to line context */
  VpDevCtxType *pDevCtx,        /* I/O: Pointer to the device context */
  VpOptionIdType option,        /* Input: Selects the option to get */
  uint16 handle)                /* Input: Handle value returned with event */
```

**DESCRIPTION**

This function retrieves the current setting of an option applied to the specified device or line. The `option` argument determines which option is read. For a list and description of all VP-API-II options see Chapter 3.

***Notes:***

`VP_OPTION_ID_DEBUG_SELECT` *cannot be retrieved with* `VpGetOption()`. See Chapter 13 for details on debug functions enabled with `VP_OPTION_ID_DEBUG_SELECT`.

`VpGetOption()` starts a process in the VP-API-II/VTD that retrieves the requested data from a device. This function does not wait for the data to become available. Instead, this function returns immediately, and an event occurs at some later time indicating that the requested data is available.

This exact option setting that is retrieved by this function depends on the values of the device context, line context, and option arguments. The table below summarizes this behavior. The "Option" column indicates whether the target option is device-specific or line-specific. The "Device Ctx" and "Line Ctx" columns indicate whether a valid pointer or `VP_NULL` is passed for the `pDevCtx` and `pLineCtx` parameters, respectively.

**Table 10–1 `VpGetOption()` Behavior**

| Option | Device Ctx | Line Ctx | Result |
|---|---|---|---|
| device | VP_NULL | VP_NULL | returns `VP_STATUS_INVALID_ARG` |
| device | VP_NULL | valid | gets option for device that controls the specified line |
| device | valid | VP_NULL | gets option for the specified device |
| device | valid | valid | returns `VP_STATUS_INVALID_ARG` |
| line | VP_NULL | VP_NULL | returns `VP_STATUS_INVALID_ARG` |
| line | VP_NULL | valid | gets option for the specified line |
| line | valid | VP_NULL | returns `VP_STATUS_INVALID_ARG` |
| line | valid | valid | returns `VP_STATUS_INVALID_ARG` |

The `VP_LINE_EVID_RD_OPTION` event is generated as a result of this function call, indicating that the requested option data is available. The `handle` argument to `VpGetOption()` specifies the event handle that is attached to this event. Upon receiving this event, the Application must call `VpGetResults()` with a pointer to the appropriate data structure to retrieve the option settings. Refer to `VP_LINE_EVID_RD_OPTION`, on page 94 for more information on retrieving the option data associated with this event.

**RETURNS**

`VP-API-II Return Codes (VpStatusType)`

**EVENTS GENERATED**

`VP_LINE_EVID_RD_OPTION`, on page 94

## 10.2.7 VpGetOptionImmediate()

**SYNTAX**

```
VpStatusType
VpGetOptionImmediate(
  VpLineCtxType *pLineCtx,          /* I/O: Pointer to line context */
  VpDevCtxType *pDevCtx,            /* I/O: Pointer to the device context */
  VpOptionIdType option,            /* Input: Selects the option to get */
  void *pResults)                   /* Output: Buffer to place results */
```

**DESCRIPTION**

This function is nearly the same as `VpGetOption()`, except this function returns the option data immediately (in the `pResults` buffer provided) and does NOT generate an event.

- `VpGetOption()` behavior is described on page 125.
- List of supported ZL880/miSLIC options is provided in Table 3–2.

***Device Compatibility Note:***

1. *The CSLAC silicon communicates via immediate MPI transactions and could therefore support this function (see Microsemi Customer Support for interest and further information).*
2. *This function however CANNOT be supported on any VCP, VCP2 or VE792 device. It is a limitation of the silicon. These devices require mailbox communication which for the purpose of this function means the device contains data that is not readily available.*

**RETURNS**

`VP-API-II Return Codes (VpStatusType)`

**EVENTS GENERATED**

None

## 10.2.8 VpGetLineState()

**SYNTAX**

```
VpStatusType
VpGetLineState(
  VpLineCtxType *pLineCtx,          /* I/O: Pointer to line context */
  VpLineStateType *pCurrentState)   /* Output: 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 104 describes the line states.

***Notes:***
*If Ringing Cadence is used, the VP-API-II considers the line in the "Ringing" state for the entire duration of Ringing Cadence. Calling* `VpGetLineState()` *during a Ringing Cadence will return the state used to start Ringing Cadence (either* `VP_LINE_RINGING` *or* `VP_LINE_RINGING_POLREV`*). If an off-hook is detected during a Ringing Cadence, the line will be automatically set to the* `ringTripExitSt` *defined by option* `VP_OPTION_ID_RING_CNTRL`*.*

**RETURNS**

`VP-API-II Return Codes (VpStatusType)`

**EVENTS GENERATED**

None

## 10.2.9 VpQueryImmediate()

**SYNTAX**

```
VpStatusType
VpQueryImmediate(
  VpLineCtxType *pLineCtx,      /* I/O: Pointer to line context */
  VpQueryIdType queryId,        /* Input: Information to query */
  void *pResults)               /* Output: Buffer to place results */
```

**DESCRIPTION**

This function provides various information about a particular line or device.

The queryId information that can be requested is defined by the VpQueryIdType enumeration:

```
Enumeration Data Type: VpQueryIdType:
    VP_QUERY_ID_DEV_TRAFFIC
    VP_QUERY_ID_LINE_CAL_COEFF
```

The VpQueryIdType type does include more values, but these are the only ones supported by this function. Others may be supported by **VpQuery()** or on other device families.

The results of the query will be copied into a buffer pointed to by pResults. This buffer must be large enough to hold the result type of the specified query, as detailed below in Table 10–2. For simplicity, the application may use the VP-API-II provided type **VpQueryResultsType** which is defined as a union of all the possible query result types.

**Table 10–2 VpQueryImmediate() pResults Types**

| queryId Value | pResults Type | VpQueryResultsType member |
|---|---|---|
| VP_QUERY_ID_DEV_TRAFFIC | uint32 | trafficBytes |
| VP_QUERY_ID_LINE_CAL_COEFF | uint8[256] | calProf |

VP_QUERY_ID_DEV_TRAFFIC

Returns the number of MPI bytes sent to and received from the device associated with the line since the last VP_QUERY_ID_DEV_TRAFFIC query. The pResults argument must point to a uint32 value when using this queryId value.

The purpose of VP_QUERY_ID_DEV_TRAFFIC is to provide a way for applications to determine if the MPI bus has enough bandwidth to sustain the traffic demands of the system (e.g., running Call Control Application for all devices sharing the same MPI bus). There are two ways it can be used to accomplish this goal:

1. *By using* VP_QUERY_ID_DEV_TRAFFIC *at a specified interval, customers can fully characterize the MPI traffic (i.e., bytes/time) for a running Call Control Application, then compare these results against pre-determined MPI limits. If a host timer is not available (i.e., to provide the "specified interval") the required interval can be created using* **VpGenTimerCtrl()**.
2. VP_QUERY_ID_DEV_TRAFFIC *can be used to measure the MPI traffic requirements of all functions that will be used during normal call control processing. Then using an appropriate statistical model, worst case scenarios can be identified.*

VP_QUERY_ID_LINE_CAL_COEFF
Provides the per-line Calibration Profile. This query writes to a uint8 array of up to 256 bytes with a Calibration Profile that may NOT fill the entire buffer. To find the actual size of the Calibration Profile written to the buffer, read the fourth byte of the array and add 4:

```
profileLength = calProf[3] + 4;
```

***Profile Length Note:***
*The length of all Profiles can be computed using the method described above (profileLength = profile[3] + 4). That's because all Profiles comply with a header definition that includes the first 6 bytes). Replacing the above line with the VP-API-II enum values looks as follows:*

```
profileLength = calProf[VP_PROFILE_LENGTH] + VP_PROFILE_LENGTH + 1;
```

**Application Note (VP_QUERY_ID_LINE_CAL_COEFF):**
This feature was implemented as a more efficient way to retrieve the calibration values than the method previous supported. The method previously supported is NOT being obsoleted, but now customers have a choice:

Previous Method
1. Calling **VpCal()** with calType = VP_CAL_GET_LINE_COEFF.
2. Call **VpGetEvent()** until receive **VP_EVID_CAL_CMP** event.
3. Call **VpGetResults()**.

New Method
1. Call **VpQueryImmediate()** with VP_QUERY_ID_LINE_CAL_COEFF.

**RETURNS** **VP-API-II Return Codes (VpStatusType)**

**EVENTS GENERATED** None

## 10.2.10 VpQuery()

**SYNTAX**

```
VpStatusType
VpQuery(
 VpLineCtxType *pLineCtx,      /* I/O: Pointer to line context */
 VpQueryIdType queryId,        /* Input: Information to query */
 uint16 handle)                /* Input: Handle to return with event */
```

**DESCRIPTION** This function provides various information about a particular line or device.

The queryId information that can be requested is defined by the VpQueryIdType enumeration:

```
Enumeration Data Type: VpQueryIdType:
    VP_QUERY_ID_TEMPERATURE
    VP_QUERY_ID_SW_DUTY_CYCLE
```

**VP_QUERY_ID_TEMPERATURE** provides an uncalibrated measurement of the SLAC device temperature.

**VP_QUERY_ID_SW_DUTY_CYCLE** provides information about the cycle count and on-time of the switching power supply.

This function generates the **VP_LINE_EVID_QUERY_CMP** event with the event.hasResults value set to TRUE (indicating results are pending). The application must then call **VpGetResults()** to receive the data into a struct of VpQueryResultsType. See **VP_LINE_EVID_QUERY_CMP**, on page 98 for a description of this type and the result formats.

**RETURNS** **VP-API-II Return Codes (VpStatusType)**

**EVENTS GENERATED** **VP_LINE_EVID_QUERY_CMP**, on page 98

CHAPTER

# 11 SYSTEM SERVICES

## 11.1 OVERVIEW

The System Services layer provides critical section, timing and interrupt control functions. These functions are system-dependent and must be implemented specifically for each platform on which the VP-API-II is used.

Note that some of the functions described in this section are in support of interrupt management as described in Chapter 7, Interrupt and Event Handling. They are described here because these functions are to be implemented (when needed) by the user.

The following functions are included in the System Services layer (refer to files: `sys_service.h` and `sys_service.c` for examples):

- **`VpSysDebugPrintf()`** – Print mechanism used by VP-API-II Debug features.
- **`VpSysEnterCritical() and VpSysExitCritical()`** – Blocks/Opens entry into a critical section of VP-API-II code through some user-defined method.
- **`VpSysDtmfDetEnable() and VpSysDtmfDetDisable()`** – These functions are used by the VP-API-II to control a DTMF digit decoding resource that is outside the scope of the VP-API-II. This is used for Type-II Caller ID implementation.
- **`VpSysWait()`** – Implements a software delay. Only needed if **`VP_DEVICE_OPTION_ID_RING_PHASE_SYNC`** is enabled on a device with revCode < 7.

## 11.2 FUNCTION DESCRIPTIONS

### 11.2.1 VpSysDebugPrintf()

**SYNTAX**

```
void
VpSysDebugPrintf(
 Variable)                          /* Input: String to print out */
```

**DESCRIPTION**

This function is used by the Debug Functions described in Chapter 13, on page 137. It is a print mechanism used for the purposes of debug output and in the simplest case can be implemented as:

```
#define VpSysDebugPrintf printf     /* User Space implementation */
```

OR

```
#define VpSysDebugPrintf printk     /* Kernel Space implementation */
```

Please note that this function may be called multiple times in displaying a single line of debug output (i.e., the argument will not always end in "\n"). This may be important for systems in which prefixing, suffixing, or other handling of the output stream is done line-by-line.

**RETURNS**

None

**EVENTS GENERATED**

None

## 11.2.2 VpSysEnterCritical() and VpSysExitCritical()

**SYNTAX**

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

**DESCRIPTION**

These functions protect critical sections of VP-API-II code and device access from reentrant execution. The Application developer must implement these functions such that for a given device, no VP-API-II functions can be called from another thread of execution while any thread is within a critical section. This is typically done by disabling appropriate interrupts or "taking" a task-blocking semaphore.

The `deviceId` argument indicates which device resource or object is being accessed during this critical section. In systems with more than one VTD attached to the host microprocessor, the Application can use this information to limit the number of interrupts disabled or tasks blocked by semaphores to only those interrupts or tasks that are related to a specific device. In most implementations the `deviceId` argument can simply be ignored.

The `criticalSecType` argument specifies the type of critical section being entered, and may take one of the following values:

```
Enumeration Data Type: VpCriticalSecType:
    VP_MPI_CRITICAL_SEC
    VP_CODE_CRITICAL_SEC
```

MPI critical sections occur around MPI bus transactions. MPI transactions are timing-sensitive and must not be interrupted.

Code critical sections are used to protect device and line object data from simultaneous access by more than one thread of execution.

In the simplest case, the same protection mechanism can be used for all critical section types. Alternatively, the Application may choose to protect different types of critical sections using different mechanisms. For example, MPI critical sections could be protected by disabling all relevant interrupts, while code critical sections could be protected by semaphores. The decision is left to the Application developer.

**`VpSysEnterCritical()` and `VpSysExitCritical()`** needs to only support single-depth critical sections for ZL880/miSLIC Applications. The ZL880/miSLIC VP-API-II does not call `VpSysEnterCritical()` more than once before calling `VpSysExitCritical()`.

***Notes:***

*If the Application is designed such that all VP-API-II calls are made from only one thread of execution then these functions can be empty.*

**RETURNS**

The nesting level after the call is completed.

**EVENTS GENERATED**

None

### 11.2.3 VpSysDtmfDetEnable() and VpSysDtmfDetDisable()

**SYNTAX**

```
void
VpSysDtmfDetEnable( or
VpSysDtmfDetDisable(
 VpDeviceIdType deviceId,          /* Input: Selects the target device */
 uint8 channelId)                  /* Input: 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 ZL880/miSLIC devices, the VP-API-II/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 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 114* for further information on this function.

**RETURNS**

None

**EVENTS GENERATED**

None

### 11.2.4 VpSysWait()

**SYNTAX**

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

**DESCRIPTION**

The VP-API-II calls this function when it needs to wait a certain amount of time. The `time` argument determines the wait time in terms of 125us frame sync periods.

This function only needs to be implemented if **VP_DEVICE_OPTION_ID_RING_PHASE_SYNC** is enabled on a device where **VpGetDeviceInfo()** returns revCode < 7.

**RETURNS**

None

**EVENTS GENERATED**

None

**DEVICES**

All

**TERMINATIONS**

All

CHAPTER

# 12 HARDWARE ABSTRACTION LAYER

## 12.1 OVERVIEW

The Hardware Abstraction Layer (HAL) defines functions for communicating with a target VTD. These functions hide the details of the platform hardware design from the VP-API-II. The customer must implement these functions as appropriate for their specific platform. Only one HAL function is required for ZL880/miSLIC Applications:

- `VpMpiCmd()` – Implements MPI/ZSI transactions.

`VpMpiCmd()` takes a `deviceId` argument that identifies the target VTD. The `deviceId` argument is of type **`VpDeviceIdType`** defined by the customer which can be any valid 'C' type (see `vp_api_types.h`). The `deviceId` is associated with the device object during configuration (`VpMakeDeviceObject()`) and passed through the VP-API-II to the Application in some of the VP-API-II functions.

Unless otherwise noted, the MPI/ZSI interface will simply be referred to as "MPI". Exceptions are noted in the "Requirements" section of `VpMpiCmd()`.

## 12.2 FUNCTION DESCRIPTIONS

### 12.2.1 VpMpiCmd()

**SYNTAX**

```
void
VpMpiCmd(
 VpDeviceIdType deviceId,        /* Input: Selects the target device */
 uint8 ecVal,                    /* Input: Enable Channel reg. value */
 uint8 cmd,                      /* Input: Command byte */
 uint8 cmdLen)                   /* Input: length of data string */
 uint8 *dataPtr)                 /* I/O: Pointer to data string */
```

**DESCRIPTION**

This function implements block level MPI read/write transactions with the ZL880/miSLIC device. The target device is indicated by the `deviceId` argument. The `ecVal` argument contains the value written to the ZL880/miSLIC device's Enable Channel register (`0x4A`).

The `cmd` argument contains the first MPI command in the block (note: the total data passed may be a series of MPI cmd+data, not just a single MPI cmd+data set). Bit zero of the `cmd` byte specifies whether the entire MPI transaction is a read (1) or a write (0) operation. If the MPI transaction is a write operation, this function writes `cmdLen` number of bytes following the command byte from the location pointed by the `dataPtr` argument to the ZL880/miSLIC device. If a read operation, this function reads and stores `cmdLen` number of bytes from the ZL880/miSLIC device to the location pointed by the `dataPtr` argument.

**REQUIREMENTS**

| Value | Interface | Requirement | Comments |
|---|---|---|---|
| CS Off-Time | MPI | ZL880 >= 0ns | |
| | | VE960 (does not support MPI) = N/A | |
| | ZSI | All Devices = Function of SPI CLK and PCLK. | See Data Sheet for details. |
| NO_OP | MPI | Not Required in MPI transaction. | |
| | ZSI | 1 - NO_OP (0x06) Required after the "Read" Command for every Read transaction. | See the code example provided in the Function Description in Section 12.2.1 |
| Blocking | MPI and ZSI | Required only IF multiple access to the same device is possible by more than one thread at the same time. | |
| | | IF required: All MPI/ZSI commands to the device passed to VpMpiCmd() must be sent before ANY other access to the device can be allowed. | The Application should use functions VpSysEnterCritical() and VpSysExitCritical()or similar methods to meet this requirement. |
| Latency | MPI and ZSI | Once started, a single MPI/ZSI transaction (full series of [MPI/ZSI Command + Data] in a single dataPtr buffer) should be completed in the shortest amount of time possible. Unexpected behavior may occur if commands are delayed. | |

**MPI EXAMPLE**

```
#define EC_WRT_CMD (0x4A)

void
VpMpiCmd(
    VpDeviceIdType deviceId,
    uint8 ecVal, uint8 cmd, uint8 cmdLen, uint8 dataPtr)
{
    uint8 cmdIndex = 0;

    /* Write the EC Value to the EC Register */
    spi_byte_write(deviceId, EC_WRT_CMD);
    spi_byte_write(deviceId, ecVal);

    /* Write the Command Byte */
    spi_byte_write(deviceId, cmd);

    if (cmd & 1) {  /* If cmd LSB = '1', Perform Read Operation */
        for (cmdIndex = 0; cmdIndex < cmdLen; cmdIndex++) {
            dataPtr[cmdIndex] = spi_byte_read(deviceId);
        }
    } else {        /* If cmd LSB = '0', Perform Write Operation */
        for (cmdIndex = 0; cmdIndex < cmdLen; cmdIndex++) {
            spi_byte_write(deviceId, dataPtr[cmdIndex]);
        }
    }
}
```

**ZSI EXAMPLE**

This is easily modified for ZSI devices as follows:

```
    if (cmd & 1) {  /* If cmd LSB = '1', Perform Read Operation */
        /* Previous Command was Read. Need to send the NO_OP first
         * for ZSI interface */
        spi_byte_write(deviceId, 0x06);

        /* Now read in the data */
        for (cmdIndex = 0; cmdIndex < cmdLen; cmdIndex++) {
            dataPtr[cmdIndex] = spi_byte_read(deviceId);
        }
    } else {        /* If cmd LSB = '0', Perform Write Operation */
        for (cmdIndex = 0; cmdIndex < cmdLen; cmdIndex++) {
            spi_byte_write(deviceId, dataPtr[cmdIndex]);
        }
    }
}
```

**APPLICATION NOTE**

In the case where the same **VpMpiCmd()** supports both MPI devices and ZSI devices, the `deviceId` may easily be used to determine which type of interface exists. Remember that the `deviceId` is of user defined type `VpDeviceIdType`, so it is possible to do something simple to identify MPI vs. ZSI. For example: using the MSB to identify ZSI ('1' = ZSI, '0' = MPI) then the `deviceId` selection could be as follows:

Given: `typedef uint8 VpDeviceIdType;`

| deviceId | Interface | CS |
|---|---|---|
| 0x00 | MPI | 0 |
| 0x80 | ZSI | 0 |
| 0x01 | MPI | 1 |
| 0x81 | ZSI | 1 |

and so on...

The VP-API-II makes no assumption about the type of `VpDeviceIdType`. However, if making `VpDeviceIdType` non-compatible with an integer type (e.g., if defined as a struct), then the user may have to modify the macro:

```
#define VP_PRINT_DEVICE_ID(deviceId) VpSysDebugPrintf(" (dev 0x%2.2X)",
(int)deviceId)
```

Found in `vp_api_types.h`.

**RETURNS**

None

CHAPTER

# 13 DEBUG FUNCTIONS

## 13.1 OVERVIEW

The VoicePath API-II includes many different types of debug used to help isolate problems in the Application and/or system. Each type of output can be included/excluded at compile time. Included types can be enabled/disabled at run-time using `VP_OPTION_ID_DEBUG_SELECT` option or at compile-time using `VP_CC_DEBUG_SELECT` (in `vp_api_cfg.h`).

## 13.2 TYPES OF DEBUG OUTPUT

**Table 13–1 Debug Flags Used by ZL880/miSLIC VP-API-II**

| Macro | Description | Purpose |
|---|---|---|
| VP_DBG_ERROR | Error messages | Internal or System error. Usually will coincide with VP-API-II return value other than VP_STATUS_SUCCESS. |
| VP_DBG_WARNING | Warning messages | Reports behavior that is probably not what was intended. |
| VP_DBG_INFO | Info messages | Provides status indication during normal operation. |
| VP_DBG_API_FUNC | VP-API-II function entry/ exit messages | Output when VP-API-II functions are entered and exited. |
| VP_DBG_API_FUNC_INT | Internal function entry/exit messages | Analagous to VP_DBG_API_FUNC, but applies to internal functions. |
| VP_DBG_HOOK | Hook and Ground Key activity | Output for raw and filtered hook and ground key activity. |
| VP_DBG_LINE_STATE | Line State messages | Output when either device or VP-API-II states are changed. |
| VP_DBG_CALIBRATION | Calibration messages. | Calibration related information. Occurs when accessing calibration values and during execution of calibration functions. |
| VP_DBG_TEST_FUNC | Test entry/exit messages. | Output when VP-API-II level test functions are entered/exited. |
| VP_DBG_TEST | Test messages. | Occurs during execution of the VP-API-II test functions. |
| VP_DBG_SEQUENCER | Messages from the internal VP-API-II sequencer. | Output when running a ringing/tone cadence or other operation that uses the VP-API-II sequence engine (e.g., VpSendSignal(), VpStartMeter()) excluding CID operations. |
| VP_DBG_CID | Caller ID messages. | Output for all Caller ID related functions (Note: Does not ouput Ringing Control information used in Type I CID generation). |
| VP_DBG_HAL | Hardware Abstraction Layer messages | Device accesses performed by the VP-API-II |
| VP_DBG_SSL | System Services Layer messages | Displays function calls and result codes of customer-defined SSL functions. |
| VP_DBG_EVENT | Event messages | Event related information. Occurs in functions VpGetEvent() and VpGetResults(). |
| VP_DBG_GAIN | Digital Gain messages | Output for access to the Digital Gain blocks of the SLAC. Occurs in VpSetRelGain() and VpSetOption() for optionId = VP_OPTION_ID_ABS_GAIN. |
| VP_DBG_TIMER | Timer debug output | Output for VP-API-II internal timers. |
| VP_DBG_ADP_RING | Adaptive Ringing output | Output when the Adaptive Ringing Algorithm is running (see VP_DEVICE_OPTION_ID_ADAPTIVE_RINGING). |
| VP_DBG_INTERRUPT | Interrupt debug | Output when an interrupt is detected. Generally this output shows pre-processed "raw" silicon data. |
| VP_DBG_DTMF | DTMF detection | Shows when DTMF sampling starts or stops, and when digit changes are detected. |
| VP_DBG_DTMF_DETAIL | DTMF detection details | Shows internal details for each block of DTMF detection. |

The above macros are defined as bitmasks which can be ORed together to select any combination of debug output types. The macro VP_DBG_ALL is provided for selecting all debug output types.

## 13.3 DEBUG OUTPUT SELECTION AT COMPILE TIME

Debug output strings and the code necessary for displaying them can occupy a significant amount of memory in some Applications. Therefore, the VP-API-II provides the ability to exclude unwanted types of debug output at compile time.

Customers may want to compile in all types of debug output during initial development, but include less debug output in the final Application. Or, if separate "production" and "debug" builds are maintained, a different selection of debug output may be specified in each.

The selection of debug output types at compile time is specified in the header files in the api_lib/includes directory:

The VP_DEBUG option (in vp_api_cfg.h) applies to all types of debug output.

- If VP_DEBUG is undefined, then all debug output is compiled out.
- If VP_DEBUG is defined, then VP_CC_DEBUG_SELECT sets the types of debug output that will be included.

The VP_CC_DEBUG_SELECT macro (in vp_debug.h) can be defined as VP_DBG_ALL to include all types of debug output, or it can be defined as the bitwise OR of one or more of the debug selector values (Table 13–1). Any debug output types not included in the definition will be compiled out.

Debug output excluded at compile time cannot be enabled at runtime.

## 13.4 DEBUG OUTPUT SPECIFICITY

For each type of debug output, the VP-API-II source code may generate line-specific, device-specific, or "non-specific" (not particular to any device or line) messages.

Line-specific messages apply to a single line in the system. For example, passing an invalid Profile argument to **VpConfigLine()** will generate a line-specific debug message.

If the line cannot be identified, or if the error message is logically not specific to a particular line, then a device-specific debug message will be generated.

If the device cannot be identified, a non-specific debug message will be generated. In some instances, a message may be clearly line-specific or device-specific, but the particular function in which it is generated does not have enough information to determine the line or device association. In such cases, the debug output is non-specific.

The run-time selection of line-specific debug output messages is stored in the Line Object. The device-specific selections are stored in the Device Object. The non-specific selections are stored in a global variable (vpDebugSelectMask) defined in vp_debug.c.

## 13.5 DEBUG OUTPUT SELECTION AT RUN TIME

At run time, the **VP_OPTION_ID_DEBUG_SELECT** option can be used to enable/disable debug output. Using this option, each type of debug output can be enabled/disabled per line, per device, or for non-specific output.

The pValue argument of **VpSetOption()** should be a pointer to a uint32 number. The value should be the bitwise OR of one or more debug output selectors (Table 13–1). Other debug output types will be disabled.

The other arguments to VpSetOption() determine the specificity of the debug output that is being enabled/disabled:

**Table 13–2 Valid Parameter Combinations for VP_OPTION_ID_DEBUG_SELECT**

| pLineCtx | pDevCtx | Result |
|---|---|---|
| valid | VP_NULL | Enables/disables line-specific debug output messages for the given line |
| VP_NULL | valid | Enables/disables device-specific debug output messages for the given device |
| valid | valid | return VP_STATUS_INVALID_ARG. |
| VP_NULL | VP_NULL | Enables/disables non-associated debug output messages |

**Important:** The above cases do not overlap in effect. Enabling/disabling the device-specific debug output for a given device will have no effect on the line-specific debug output for lines of that device. Also, enabling/disabling non-specific debug output has no effect on device-specific or line-specific output.

## 13.6 PORTABILITY CONCERNS

### 13.6.1 Static Variable Coherency

It has always been a major design goal of the VP-API-II to give customers complete freedom over memory allocation. To that end, all state variables are maintained in a Line Object or Device Object which can be allocated statically, dynamically ("malloc"), or on the stack.

To allow enabling/disabling debug output at run-time for “non-specific” events, the VP-API-II uses a statically-allocated global variable, of type uint32, called vpDebugSelectMask, located in vp_debug.c. This variable is modified by VpSetOption() using optionId = VP_OPTION_ID_DEBUG_SELECT when both the pLineCtx and pDevCtx arguments are VP_NULL. It is accessed every time a non-specific debug output message is generated to determine if the message will be displayed.

The vpDebugSelectMask variable is internal to the API, and it is not normally necessary to access it externally. However, problems can arise in multiprocessing environments where each process may have its own copy of all static variables. In such cases, it is important to restore this variable each time a new process is started, either by modifying it directly, or by calling VpSetOption().

### 13.6.2 ANSI X3.64 Colors

The VP-API-II uses ANSI X3.64 color codes to display each type of debug message in a different color. These codes work with all popular modern terminal programs. However, there are many cases in which it is desirable to exclude these extra characters from the debug output. To do so, undefine the VP_DEBUG_COLOR option in vp_debug.h.

### 13.6.3 Displaying VpDeviceIdType and VpLineIdType

Line-specific and device-specific debug output messages are prefixed by displaying the corresponding VpLineIdType or VpDeviceIdType struct. Since these are customer-defined types, it may not be appropriate to display them using a printf(“%d”) format specifier. If not, the display method can be changed by modifying the definitions of the VP_PRINT_DEVICE_ID() and VP_PRINT_LINE_ID() macros (located in vp_debug.h).

## 13.7 FUNCTION DESCRIPTIONS

The following Debug Functions are conditionally compiled into the VP-API-II. They are compiled in when debug is enabled (i.e., `#define VP_DBG_ENABLE` is set) 'AND' when error debug strings are included (when `#define VP_CC_DEBUG_SELECT` includes the `VP_DBG_INFO` debug mask).

- **`VpRegisterDump()`** – This function generates output data of silicon register content.
- **`VpObjectDump()`** – This function generates output data of the line or device object.
- **`VP_PRINT_DEVICE_ID()`** – This macro prints the `deviceId` associated with the device.
- **`VP_PRINT_LINE_ID()`** – This macro prints the `lineId` associated with the line.

### 13.7.1 VpRegisterDump()

**SYNTAX**

```
VpStatusType
VpRegisterDump(
 VpDevCtxType *pDevCtx)
```

**DESCRIPTION**

This function generates output data of silicon register content. Note that calling this function may affect normal event handling since it may cause a pending interrupt to be cleared. Communication with the silicon must be established prior to calling this function.

**RETURNS**

**VP-API-II Return Codes (VpStatusType)**

**EVENTS GENERATED**

None

### 13.7.2 VpObjectDump()

**SYNTAX**

```
VpStatusType
VpObjectDump(
 VpLineCtxType *pLineCtx,
 VpDevCtxType *pDevCtx)
```

**DESCRIPTION**

This function generates output data of the line or device object. This function accesses only the software object data and can be called even when communication with the silicon has yet to be established.

**RETURNS**

**VP-API-II Return Codes (VpStatusType)**

**EVENTS GENERATED**

None

### 13.7.3 VP_PRINT_DEVICE_ID()

**SYNTAX**

```
User Defined
VP_PRINT_DEVICE_ID(
 VpDeviceIdType deviceId)
```

**DESCRIPTION**

This macro prints the `deviceId` associated with the device. An example of this macro is in: `\arch\examples\vp_api_types.h` which prints `deviceId` as an `int`. This may not be appropriate given the Application definition of `VpDeviceIdType`, in which case this macro should be modified.

**RETURNS**

```
User Defined
```

**EVENTS GENERATED**

None

### 13.7.4 VP_PRINT_LINE_ID()

**SYNTAX**

```
User Defined
VP_PRINT_LINE_ID(
 VpLineIdType lineId)
```

**DESCRIPTION**

This macro prints the `lineId` associated with the line. An example of this macro is in: `\arch\examples\vp_api_types.h` which prints `lineId` as an `int`. This may not be appropriate given the Application definition of `VpLineIdType`, in which case this macro should be modified.

**RETURNS**

```
User Defined
```

**EVENTS GENERATED**

None

CHAPTER

# A GLOSSARY

| | |
|---|---|
| **API** | Application Programmer's Interface. In generic terms refers to any Application level Software Interface. When used with VP-API-II refers to the specific Microsemi SW Package (Le71SK0002 or Le71SDKAPIL). |
| **Channel** | See Section 1.3 |
| **CODEC** | Coder/Decoder |
| **Device** | See Section 1.3 |
| **DTMF** | Dual-Tone Multi Frequency. Signaling used for dialing and in some Caller ID Applications. |
| **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. |
| **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. |
| **IP** | Internet Protocol. Generally referring to a derived voice line via "cloud" (i.e., IP) network. |
| **ISR** | Interrupt Service Routine |
| **Kernel** | Referring to the portion of the OS that is responsible for basic Application services (e.g., scheduling, timers, etc..) and has direct access to the hardware. As opposed to "User Space" which accesses ONLY the Kernel functions.<br><br>Note that some OS contain only Kernel space (e.g., VxWorks) whereas some contain both Kernel and "User" space (e.g., Linux). Microsemi Software is designed to operate in either Kernel or User space. |
| **Line** | See Section 1.3 |
| **miSLIC** | Microsemi SLIC with part numbers Le9662, Le9672, Le9652, Le9651, Le9642, Le9641. |
| **MPI** | Micro-Processor Interface. The MPI is a command only serial interface used to communicate to the ZL880 devices. MPI is NOT supported on miSLIC devices. Note that some ZL880 devices also support the alternative interface - ZSI. |
| **OS** | Operating System. This includes any software that contains a scheduler, timers, print facilities, and most other features found in standard off-the-shelf Operating Systems such as Linux and VxWorks. |
| **PLL** | Phase-Lock-Loop. For the purposes of the ZL880/miSLIC device, this refers to the logic inside the silicon that translates from the PCLK frequency to the higher frequency used by the internal silicon logic. In absence of a PCLK, the PLL runs in "Free Run" mode. See Section 9.2.1 for details. |
| **Profile** | Profiles encapsulate Application specific data including cadencing, tones, Caller ID parameters, etc. |
| **PSTN** | Public Switched Telephone Network. FXS circuits from the PSTN are also referred to as "land lines" as opposed to a derived voice line via IP network (i.e., "cloud"). Microsemi silicon supports both "land line" and "cloud" networks. |

| | |
|---|---|
| **SLAC™** | Subscribe Line Access Circuit, a Microsemi trademark. |
| **SLIC** | Subscriber Line Interface Circuit |
| **Subscriber Line** | The analog telephone line connecting the subscriber to the PSTN. Subscriber line is synonymous with loop or local loop. |
| **VeriVoice™** | Microsemi trademark applied to Voice Telephony line and manufacturing test software. |
| **VeriVoice™ LT-API** | Microsemi's Line Test API used to perform GR-909 line and circuit tests. |
| **VoicePath™ Profile Wizard** | A Microsoft Windows Application provided by Microsemi that creates and organizes the Profiles used by the VP-API-II. |
| **VP-API-II** | VoicePath™ API-II. Microsemi's 2nd Generation API that provides access to the ZL880/miSLIC silicon. This is the main set of software discussed in this document. Note the difference between the specific VP-API-II and generic definition of API. In some cases these terms are interchangeable, but in most cases used in this document they are not. |
| **VTD** | Voice Termination Device. |
| **VVMT** | VeriVoice Manufacturing Test Package. Software used to verify board build and may be used to generate the required Calibration values necessary to provide correct operation of Microsemi silicon. |
| **ZSI** | Microsemi serial interface containing command and data. This interface requires less pins than the MPI/PCM interface but can only be used to control one device (MPI interface can be used for multiple devices on the same bus). ZL880 silicon supports both MPI and ZSI while miSLIC supports ONLY the ZSI interface. |

CHAPTER

# B FUNCTION INDEX

Table B- 1, “VoicePath™ API II Functions Summary,” on page 146 lists all VP-API-II functions, along with their input types, return type and applicable termination types.

## Table B–1 VoicePath™ API II Functions Summary

| Function Name | Arguments | Return Type | Page |
|---|---|---|---|
| **System Configuration Functions** | | | |
| VpMakeDeviceObject | VpDeviceType deviceType, VpDeviceIdType deviceId, VpDevCtxType *pDevCtx, void *pDevObj | VpStatusType | 65 |
| VpMakeLineObject | VpTermType termType, uint8 channelId, VpLineCtxType *pLineCtx, void *pLineObj, VpDevCtxType *pDevCtx | VpStatusType | 66 |
| VpMakeDeviceCtx | VpDeviceType deviceType, VpDevCtxType *pDevCtx, void *pDevObj | VpStatusType | 67 |
| VpMakeLineCtx | VpLineCtxType *pLineCtx, void *pLineObj, VpDevCtxType *pDevCtx | VpStatusType | 67 |
| VpFreeLineCtx | VpLineCtxType *pLineCtx | VpStatusType | 68 |
| VpMapLineId | VpLineCtxType *pLineCtx, VpLineIdType lineId | VpStatusType | 68 |
| **Initialization Functions** | | | |
| VpInitDevice | VpDevCtxType *pDevCtx, VpProfilePtrType pDevProfile, VpProfilePtrType pAcProfile, VpProfilePtrType pDcProfile, VpProfilePtrType pRingProfile, VpProfilePtrType pFxoAcProfile, VpProfilePtrType pFxoCfgProfile | VpStatusType | 69 |
| VpInitLine | VpLineCtxType *pLineCtx, VpProfilePtrType pAcProfile, VpProfilePtrType pDcFeedOrFxoCfgProfile, VpProfilePtrType pRingProfile | VpStatusType | 71 |
| VpConfigLine | VpLineCtxType *pLineCtx, VpProfilePtrType pAcProfile, VpProfilePtrType pDcFeedOrFxoCfgProfile, VpProfilePtrType pRingProfile | VpStatusType | 72 |
| VpCalLine | VpLineCtxType *pLineCtx | VpStatusType | 74 |
| VpCal | VpLineCtxType *pLineCtx, VpCalType calType, void *inputArgs | VpStatusType | 73 |
| VpInitRing | VpLineCtxType *pLineCtx, VpProfilePtrType pCadProfile, VpProfilePtrType pCidProfile | VpStatusType | 75 |
| VpInitCid | VpLineCtxType *pLineCtx, uint8 length, uint8p pCidData | VpStatusType | 76 |
| VpInitMeter | VpLineCtxType *pLineCtx, VpProfilePtrType pMeterProfile | VpStatusType | 76 |
| VpInitProfile | VpDevCtxType *pDevCtx, VpProfileType type, VpProfilePtrType pProfileIndex, VpProfilePtrType pProfile | VpStatusType | 77 |
| **Interrupt Functions** | | | |
| VpGetEvent | VpDevCtxType *pDevCtx, VpEventType *pEvent | bool | 80 |
| VpGetResults | VpEventType *pEvent, void *pResults | VpStatusType | 82 |

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

| Function Name | Arguments | Return Type | Page |
|---|---|---|---|
| VpFlushEvents | VpDevCtxType *pDevCtx | VpStatusType | 80 |
| **Control Functions** | | | |
| VpSetLineState | VpLineCtxType *pLineCtx, VpLineStateType state | VpStatusType | 104 |
| VpSetLineTone | VpLineCtxType *pLineCtx, VpProfilePtrType pToneProfile, VpProfilePtrType pCadProfile, VpDtmfToneGenType *pDtmfControl | VpStatusType | 109 |
| VpSetRelayState | VpLineCtxType *pLineCtx, VpRelayControlType rState | VpStatusType | 106 |
| VpSetRelGain | VpLineCtxType *pLineCtx, uint16 txLevel, uint16 rxLevel, uint16 handle | VpStatusType | 110 |
| VpSendSignal | VpLineCtxType *pLineCtx, VpSendSignalType signalType, void *pSignalData | VpStatusType | 111 |
| VpSendCid | VpLineCtxType *pLineCtx, uint8 length, VpProfilePtrType pCidProfile, uint8p pCidData | VpStatusType | 112 |
| VpContinueCid | VpLineCtxType *pLineCtx, uint8 length, uint8p pCidData | VpStatusType | 113 |
| VpDtmfDigitDetected | VpLineCtxType *pLineCtx, VpDigitType digit, VpDigitSenseType sense | VpStatusType | 114 |
| VpStartMeter | VpLineCtxType *pLineCtx, uint16 onTime, uint16 offTime, uint16 numMeters | VpStatusType | 111 |
| VpSetOption | VpLineCtxType *pLineCtx, VpDevCtxType *pDevCtx, VpOptionIdType option, void *pValue | VpStatusType | 107 |
| VpDeviceIoAccess | VpDevCtxType *pDevCtx, VpDeviceIoAccessDataType *pDeviceIoData | VpStatusType | 115 |
| VpLineIoAccess | VpLineCtxType *pLineCtx, VpLineIoAccessType *pLineIoAccess, uint16 handle | VpStatusType | 116 |
| VpGenTimerCtrl | VpLineCtxType *pLineCtx, VpGenTimerCtrlType timerCtrl, uint32 duration, uint16 handle | VpStatusType | 117 |
| VpFreeRun | VpDevCtxType *pDevCtx, VpFreeRunModeType freeRunMode | VpStatusType | 118 |
| VpBatteryBackupMode | VpDevCtxType *pDevCtx, VpBatteryBackupModeType backupMode, uint8 vsw | VpStatusType | 119 |
| VpShutdownDevice | VpDevCtxType *pDevCtx | VpStatusType | 119 |
| **Status and Query Functions** | | | |
| VpGetLineStatus | VpLineCtxType *pLineCtx, VpInputType input, bool *pStatus | VpStatusType | 122 |
| VpGetDeviceInfo | VpDeviceInfoType *pDeviceInfo | VpStatusType | 124 |
| VpGetLineInfo | VpLineInfoType *pLineInfo | VpStatusType | 123 |

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

| Function Name | Arguments | Return Type | Page |
|---|---|---|---|
| VpGetDeviceStatus | VpDevCtxType *pDevCtx, VpInputType input, uint32 *pDeviceStatus | VpStatusType | 123 |
| VpGetLoopCond | VpLineCtxType *pLineCtx, uint16 handle | VpStatusType | 121 |
| VpGetOption | VpLineCtxType *pLineCtx, VpDevCtxType *pDevCtx, VpOptionIdType option, uint16 handle | VpStatusType | 125 |
| VpGetOptionImmediate | VpLineCtxType *pLineCtx, VpDevCtxType *pDevCtx, VpOptionIdType option, void *pResults | VpStatusType | 126 |
| VpGetLineState | VpLineCtxType *pLineCtx, VpLineStateType *pCurrentState | VpStatusType | 126 |
| VpQueryImmediate | VpLineCtxType *pLineCtx, VpQueryIdType queryId, VpQueryResultsType *pResults | VpStatusType | 127 |
| VpQuery | VpLineCtxType *pLineCtx, VpQueryIdType queryId, uint16 handle | VpStatusType | 128 |
| **System Services Layer Functions** | | | |
| VpSysEnterCritical | VpDeviceIdType deviceId, VpCriticalSecType criticalSecTyp | uint8 | 130 |
| VpSysExitCritical | VpDeviceIdType deviceId, VpCriticalSecType criticalSecTyp | uint8 | 130 |
| VpSysDtmfDetEnable | VpDeviceIdType deviceId | void | 131 |
| VpSysDtmfDetDisable | VpDeviceIdType deviceId | void | 131 |
| VpSysDebugPrintf | Variable | void | 129 |
| VpSysWait | uint8 time | void | 131 |
| **Hardware Abstraction Layer (HAL) Functions** | | | |
| VpMpiCmd | VpDeviceIdType deviceId, uint8 ecVal, uint8 cmd, uint8 cmdLen, uint8 *dataPtr | void | 134 |
| **Debug Functions** | | | |
| VpRegisterDump | VpDevCtxType *pDevCtx | VpStatusType | 141 |
| VpObjectDump | VpLineCtxType *pLineCtx, VpDevCtxType *pDevCtx | VpStatusType | 141 |
| VP_PRINT_DEVICE_ID | VpDeviceIdType deviceId | User Defined | 141 |
| VP_PRINT_LINE_ID | VpLineIdType lineId | User Defined | 142 |

CHAPTER

# C VE880 TO ZL880/MISLIC SOFTWARE CONVERSION GUIDE

## C.1 INTRODUCTION

This Appendix is intended to assist in converting a VP-API-II based software application written for the VE880 family of devices to work with the ZL880/miSLIC family of devices. The sections in this Appendix are presented in the order than an Application will encounter each step of the conversion process. The sections are presented in the following order:

1. **`Basic VP-API-II Configuration`**: This section discusses the steps necessary to include support for ZL880/miSLIC silicon in the Application and VP-API-II library.
2. **`Creating ZL880/miSLIC Objects`**: This section covers the VP-API-II data enumeration and types necessary for creating the ZL880/miSLIC device and line objects.
3. **`Profile Updates`**: In this section, the reader will see that some VE880 profiles may be used "as-is" in the ZL880/miSLIC Application, while only a few MUST be updated.
4. **`Interface Differences`**: This section addresses differences between the VE880 VP-API-II and ZL880/miSLIC VP-API-II interface.

The first 4 sections are all that is required to convert a VE880 Application to a ZL880/miSLIC Application. The remaining sections address improvements and new features in the ZL880/miSLIC VP-API-II.

5. **`Relaxed Requirements`**: This section addresses some of the improvements made to the ZL880/miSLIC VP-API-II (compared to VE880 VP-API-II). In general, these changes are easy to implement and will immediately provide some System level benefit (i.e., by reducing processor demand).
6. **`New Features`**: This section discusses new features in the ZL880/miSLIC VP-API-II.

## C.2 BASIC VP-API-II CONFIGURATION

This section refers to the configuration options defined in `vp_api_cfg.h`.

IMPORTANT: The flag `VP_CC_886_SERIES` must be defined in order to include Application and VP-API-II library support for ZL880/miSLIC silicon. With `VP_CC_886_SERIES` defined, additional compile-time options will become available. See Section 4.5 and Section 4.6 for a list of VP-API-II ZL880/miSLIC feature level compile-time options. See Section 7.1 for a discussion on ZL880/miSLIC equivalent polling modes.

When determining how to set the ZL880/miSLIC compile-time options, it may be helpful to take the following into account:

- Options available on both VE880 and ZL880/miSLIC will have a prefix of VP880 for VE880 silicon, and similar prefix of VP886 for ZL880/miSLIC silicon. Features/functions affected by these options will behave the same way for these devices for the same setting.
- If a VP880 option does not have a VP886 equivalent, then that feature is either always-on or non-applicable, or not yet implemented for the ZL880/miSLIC VP-API-II. This applies also to code size reduction options (see Section 4.5).
- New compile-time options are available for the ZL880/miSLIC VP-API-II (see Section 4.6).

## C.3 CREATING ZL880/MISLIC OBJECTS

This section addresses conversion of the VE880 arguments passed to **VpMakeDeviceObject()** and **VpMakeLineObject()** to their ZL880/miSLIC equivalent.

### C.3.1 ZL880/miSLIC VP-API-II Device Type

The VpDeviceType value passed to **VpMakeDeviceObject()** tells the VP-API-II what silicon the host controller is directly communicating with. In a VE880 Application this value is always set to VP_DEV_880_SERIES regardless of the silicon in the design. For ZL880/miSLIC Applications, the value of deviceType passed to **VpMakeDeviceObject()** will depend on the silicon used in the design, but must be one of the following:

1. VP_DEV_886_SERIES - use when configuring a ZL88601/2 (ABS) device
2. VP_DEV_887_SERIES - use when configuring a ZL88701/2 (Tracking) device

The reason for using two VpDeviceType values (rather than one) for ZL880/miSLIC Applications will become clear in the section **Profile Updates**, on page 150.

### C.3.2 ZL880/miSLIC VP-API-II Line Termination Types

The list of supported termination types for ZL880/miSLIC are in Table 1–4 on page 6. Note that the VE880 VP-API-II supports a wider range of termination types. So it is important to verify that the argument termType passed to **VpMakeLineObject()** is one from the table mentioned.

### C.3.3 Device and Line Object Types

A VP-API-II application written for the VE880 series instantiates device and line objects of types Vp880DeviceObjectType and Vp880LineObjectType. For a ZL880/miSLIC application, these objects must be changed to types Vp886DeviceObjectType and Vp886LineObjectType, respectively.

## C.4 PROFILE UPDATES

To ease the effort when converting an existing VE880 Application to ZL880/miSLIC, the ZL880/miSLIC VP-API-II was designed to use as much of the VE880 Profiles as possible. In some cases 100% compatibility was achieved, in other cases the behavior is slightly different, and in a few cases compatibility was not achieved. Table 3–1 provides the details where “**Compatible?**” can be one of the following:

- Yes (**Y**): ZL880/miSLIC and VE880 behavior is the same for the specified profile type.
- No (**N**): The ZL880/miSLIC **WILL NOT** work with a VE880 version of this profile type. A new ZL880/miSLIC specific profile must be generated.
- Conditional (**C**): ZL880/miSLIC behavior is slightly different than VE880 behavior for the specified profile type and may actually be preferred by the Application. For these profile types, customers are strongly recommended to review the differences, determine the impact to their application and take whatever corrective action (if any) is needed.

Table 3–1 VE880 to ZL880/miSLIC Profile Compatibility

| Profile Type | Compatible? (Yes, No, Conditional) | Comment |
|---|---|---|
| Device | N | The Device Profile is unique to the device family (e.g., VE880, ZL880/miSLIC) as well as the device type (e.g., ABS, Tracker). For VE880 Device Profiles, there is insufficient information to determine whether the profile was created for VE880-ABS or VE880-Tracker which has caused some issues. This has been corrected in the ZL880/miSLIC Device Profile definition. For more information on this topic, refer to Improved Profile Argument Checking.<br><br>Applications must generate unique Device Profiles for both ZL880/miSLIC-ABS and ZL880/miSLIC-Tracker designs. |
| DC | N | The DC Profile is unique to the device family (e.g., VE880, ZL880/miSLIC) as well as the ZL880/miSLIC device type (e.g., ZL880/miSLIC-ABS, ZL880/miSLIC-Tracker). Note that the DC Profile for VE880 devices is common for both VE880-ABS and VE880-Tracker designs.<br><br>The profile argument checking discussed in Improved Profile Argument Checking has been applied to ZL880/miSLIC DC Profiles (as well as the Device Profiles). Therefore, Applications must generate unique DC Profiles for both ZL880/miSLIC-ABS and ZL880/miSLIC-Tracker designs. |
| Ringing | N | The Ringing Profile is unique only to the device family (e.g., VE880, ZL880/miSLIC). The Ringing Profile for both ZL880/miSLIC-ABS and ZL880/miSLIC-Tracker designs are the same and can therefore use the same Ringing profile (but not a Ringing profile used with VE880 design).<br><br>Applications must generate a new Ringing Profile for ZL880/miSLIC designs, but can use the same (ZL880/miSLIC based) Ringing Profile for both ZL880/miSLIC-ABS and ZL880/miSLIC-Tracker designs. |
| Metering | N | The register definitions are the same for VE880 and ZL880/miSLIC for Metering, but there are Level differences making their profiles incompatible. |
| AC FXS | Y | Applications can use these profiles interchangeably between VE880 and ZL880/miSLIC designs. Note that some of the features enable through these profiles may not be available on both devices in the same VP-API-II release. |
| Caller ID | | |
| Tone | Y | Tone Profiles generated for a VE880 device can be used "as-is" in a ZL880/miSLIC Application. However, new ZL880/miSLIC Tone Profiles **cannot** be used in a VE880 Application. Doing so could cause undesirable behavior. |
| Tone Cadence | C | In general, the Cadence Profiles between VE880 (and all CSLAC devices) and ZL880/miSLIC are compatible. However, the ZL880/miSLIC Cadence behavior is slightly different.<br><br>Applications using VE880 Cadence Profiles in a ZL880/miSLIC Application should refer to Cadence Profiles, on page 153 before proceeding. |
| Ringing Cadence | | |

## Improved Profile Argument Checking

This section provides the technical details regarding Device Profile compatibility between ABS and Tracker devices for both VE880 and ZL880/miSLIC Applications. Provided the recommendations in Table 3–1 are followed, this content does not need to be understood.

*Occasional problems occur in VE880 designs when Applications migrate from a VE880-ABS design to a VE880-Tracker design, or vise-versa. If the Device Profile is not changed and as a result a VE880-ABS Device Profile is used in a VE880-Tracking design, severe problems can*

*occur. This problem is not prevented by the VP-API-II because there is currently no information in the VE880 Device Profile that indicates whether the profile was created for a VE880-Tracking device or a VE880-ABS device.*

*The ZL880/miSLIC Profile format has changed to correct this problem. The first byte (i.e., `byte[0]`) in all ZL880/miSLIC Profiles is set to the `VpDeviceType` value of the ZL880/miSLIC device selected in Profile Wizard. During* **`VpInitDevice()`**, *this value is compared with the `deviceType` value that was provided to* **`VpMakeDeviceObject()`**. *If these values don't match,* **`VpInitDevice()`** *will immediately return the error* **`VP_STATUS_ERR_PROFILE`** *and the device will not be initialized. Since according to Table 3–1 the ZL880/miSLIC DC Profile is also specific to ZL880/miSLIC-ABS and ZL880/miSLIC-Tracker, this error will also be generated by* **`VpInitLine()`** *or* **`VpConfigLine()`** *if provided a DC Profile that was generated for a different device than the one in the current design.*

## C.5 INTERFACE DIFFERENCES

This section addresses differences between the VE880 VP-API-II and ZL880/miSLIC VP-API-II interfaces.

### C.5.1 Not Compatible between VE880 and ZL880/miSLIC

The value of `eventData` provided with **`VP_DEV_EVID_DEV_INIT_CMP`** is not compatible between the VE880 and ZL880/miSLIC VP-API-II. Applications evaluating this value as indication of Init Device success/fail will have to modify the logic applied. The value of `eventData` for each device is:

**Table 3-2 VP_DEV_EVID_DEV_INIT_CMP::eventData**

| Status \ Device | ZL880/miSLIC | VE880 |
|---|---|---|
| SUCCESS | `eventData = 0` | `eventData = 1` |
| FAILURE | `eventData = non-zero` | `eventData = 0` |

The rationale for this change on ZL880/miSLIC was due to the following observations of the VE880 approach:

1. The VE880 values for "Success" and "Failure" are inconsistent with the values used for similar item status reporting throughout the rest of the VP-API-II and Line Test Software. In general, the Microsemi software defines status indications = 0 to mean "Success", and status indications = non-zero to mean "Failure" (or some kind).
2. The VE880 approach complicates the Application when adding new values that provide more fault isolation (i.e., such that more than one non-zero value indicates a fault).

The ZL880/miSLIC VP-API-II Device Initialization procedure can detect up to 12 separate fault causes and report these in `eventData`. Each fault corresponds to a bit in the `eventData` field, allowing for easy parsing of the `eventData` field into individual fault sources. For further details, see **`VP_DEV_EVID_DEV_INIT_CMP`**, on page 93.

### C.5.2 Supported for Backward Compatibility

- `VpApiTick()` - See **`VpApiTick() and Polling/Interrupt Modes`**, on page 156
- The value `VP_OPTION_WIDEBAND`. New codec mode options have been added for wideband in linear, A-law, and Mu-law encodings. `VP_OPTION_WIDEBAND` is equivalent to `VP_OPTION_LINEAR_WIDEBAND`, but the latter is preferred for clarity.

### C.5.3 No Longer Supported

- `VP_OPTION_ID_PCM_HWY` is no longer supported because only PCM highway "A" is valid.
- `VP_OPTION_ID_ZERO_CROSS` is no longer supported because **`VP_OPTION_ID_RING_CNTRL`** provides a super-set of this functionality.

## C.6 MODIFIED BEHAVIORS

This section addresses behavior differences between VE880 and ZL880/miSLIC VP-API-II. Note that in some cases the VP-API-II can be configured to force the ZL880/miSLIC VP-API-II to behave identical to VE880 VP-API-II.

### C.6.1 Wideband

Each channel can now be set to a wideband mode independently. When one channel is set to wideband, **the other channel will no longer be forced into the same mode**. Applications which depend on this behavior should be changed to set the codec mode for both channels when converting to ZL880/miSLIC or specify the Device Context in the `VpSetOption()` call to emulate the existing behavior.

### C.6.2 Cadence Profiles

The VP-API-II sequencer code for VE880 spent at least one “tick” processing each instruction in a cadence profile. This meant that delays were extended by the number of instructions processed in between them. The ZL880/miSLIC/VP886 VP-API-II sequencer code, by default, can execute cadence profile instructions without delay until it reaches a delay instruction. This causes minor timing differences in cadence execution. The following table shows the timing differences for a typical Ringing Cadence for 10ms VE880 Application compared to non-tick ZL880/miSLIC Application:

**Table 3–3 VE880 and ZL880/miSLIC Cadencer Timing Comparison**

| Step | Command | VE880 (10ms tickrate) | ZL880/miSLIC |
|---|---|---|---|
| 0 | Line State = VP_LINE_RINGING | Line set to Ringing at t = 0 | Line set to Ringing at t = 0 |
| 1 | Time Delay = 2000ms | (+10ms for tick)<br><br>Timer starts at t = 10ms<br><br>(-10ms) - VE880 Cadencer subtracts 1 tick time from specified time to compensate for the added time stepping between commands<br><br>Timer expires at t = 2000 | <br><br>(0 delay)<br><br>Timer starts at t = 0<br><br>Timer expires at t = 2000ms |
| 2 | Line State = VP_LINE_OHT | (+10ms for tick)<br><br>Line set to OHT at t = 2010ms | (0 delay)<br><br>Line set to OHT at t = 2000ms |

**Table 3–3 VE880 and ZL880/miSLIC Cadencer Timing Comparison**

| Step | Command | VE880 (10ms tickrate) | ZL880/miSLIC |
|---|---|---|---|
| 3 | Time Delay = 4000 | (+10ms for tick)<br>Timer starts at t = 2020ms<br>(-10ms) - VE880 Cadencer subtracts 1 tick time from specified time to compensate for the added time stepping between commands<br>Timer expires at t = 6010ms | (0 delay)<br>Timer starts at t = 2000ms<br>Timer expires at t = 6000ms |
| 4 | Branch to 0, Loop Forever | (+10ms for tick)<br>Branch starts at 6020ms | (0 delay)<br>Branch starts at 6000ms |
| 0 | (restart at step 0) Line State = VP_LINE_RINGING | (+10ms for tick)<br>Step 0 (Line set to Ringing) at 6030ms | (0 delay)<br>Step 0 (Line set to Ringing) at 6000ms |
| 1 | Time Delay = 2000 | (+10ms for tick)<br>Timer starts at t = 6040ms<br>(-10ms) - VE880 Cadencer subtracts 1 tick time from specified time to compensate for the added time stepping between commands<br>Timer expires at t = 8030ms | (0 delay)<br>Timer starts at t = 6000ms<br>Timer expires at t = 8000ms |
| ... | ... | ... | ... |

What Table 3–3 shows is that the VE880 Cadencer inserts delays into the Cadence Profile sequence which must be compensated for if precision timing is necessary. Note that in Cadences containing very few steps and very long Time Delays, the error is virtually unnoticeable. This issue becomes a bigger problem for Cadences containing many steps and short Time Delays (such that the tickrate error is a large percent of the Time Delay).

Using the previous Ringing Cadence, Table 3–4 shows how to compensate for the Cadence timing:

**Table 3–4 VE880 Time Compensated Ringing Cadence**

| Step | Command | VE880 Time (10ms tickrate) |
|---|---|---|
| 0 | Line State = VP_LINE_RINGING | Line set to Ringing at t = 0 |
| 1 | Time Delay = 1990ms | (+10ms for tick)<br>Timer starts at t = 10ms<br>(-10ms) - VE880 Cadencer subtracts 1 tick time from specified time to compensate for the added time stepping between commands<br>Timer expires at t = 1990 |
| 2 | Line State = VP_LINE_OHT | (+10ms for tick)<br>Line set to OHT at t = 2000ms |
| 3 | Time Delay = 3980 | (+10ms for tick)<br>Timer starts at t = 2010ms<br>(-10ms) - VE880 Cadencer subtracts 1 tick time from specified time to compensate for the added time stepping between commands<br>Timer expires at t = 5980ms |
| 4 | Branch to 0, Loop Forever | (+10ms for tick)<br>Branch starts at 5990ms |
| 0 | (restart at step 0) Line State = VP_LINE_RINGING | (+10ms for tick)<br>Step 0 (Line set to Ringing) at 6000ms |
| 1 | Time Delay = 1990 | (+10ms for tick)<br>Timer starts at t = 6010ms<br>(-10ms) - VE880 Cadencer subtracts 1 tick time from specified time to compensate for the added time stepping between commands<br>Timer expires at t = 7990ms |
| ... | ... | ... |

If a VE880 application's cadence profiles were built to compensate for this "tick" distortion by specifying shorter delays, there are two options for achieving the same behavior in a ZL880/miSLIC application:

1. Use new profiles that do not attempt to adjust for the distortion as shown in Table 3–3. This is the preferred solution when possible. When using this option, `VP886_LEGACY_SEQUENCER` in `vp_api_cfg.h` should be set to `#undef`.
2. Configure the ZL880/miSLIC sequencer to operate the same way as the VE880 sequencer and use the time-compensated profiles as shown in Table 3–4. This can be done by setting `VP886_LEGACY_SEQUENCER` in `vp_api_cfg.h` to `#define`. Enabling this option forces the ZL880/miSLIC sequencer code to insert a delay between each instruction to emulate the VE880 behavior. This option can only be used if the ZL880/miSLIC application is polling (`VpApiTick()` and/or **`VpGetEvent()`**) at the same consistent tick rate as the original VE880 application.

### C.6.3 Metering Abort

For ZL880/miSLIC, calling **`VpSetLineState()`** during metering will abort metering immediately, regardless of whether a pulse is active or not. To allow the current pulse to finish, the application should call **`VpStartMeter()`** with `numMeters=0` then wait for the **`VP_LINE_EVID_MTR_ABORT`** event before proceeding.

## C.7 RELAXED REQUIREMENTS

This section describes some of the requirements on VE880 applications that are relaxed or no longer applicable for ZL880/miSLIC applications. These relaxed requirements may allow applications to be modified to reduce complexity or consume less system resources.

### C.7.1 VpApiTick() and Polling/Interrupt Modes

VE880 Applications are required to call `VpApiTick()` at a constant rate in order for the VP-API-II to keep track of time. For ZL880/miSLIC Applications, calling `VpApiTick()` is no longer required, though it is still supported for backward compatibility. The preferred way to manage ZL880/miSLIC interrupts is by one of the methods described in **Handling Interrupts from ZL880/miSLIC Devices**, on page 79. However, for Applications that will manage ZL880/miSLIC interrupts in the same way as VE880, only the following needs to be done:

1. Configure the ZL880/miSLIC polling/interrupt mode options in `vp_api_cfg.h`. These will apply in the same manner as for VE880. For clarification, these modes are:
   - `VP886_SIMPLE_POLLED_MODE` - ZL880/miSLIC silicon is accessed every `VpApiTick()`. Physical connection to ZL880/miSLIC interrupt line not necessary.
   - `VP886_EFFICIENT_POLLED_MODE` - ZL880/miSLIC interrupt line status is requested during `VpApiTick()`. ZL880/miSLIC silicon is accessed only if interrupt pending.
   - `VP886_INTERRUPT_EDGETRIG_MODE` and `VP886_INTERRUPT_LEVTRIG_MODE` - Application calls `VpVirtualISR()` when a ZL880/miSLIC interrupt occurs. If configured for "Level Triggered", requires implementation of System Service functions `VpSysDisableInt() and VpSysEnableInt()`. ZL880/miSLIC silicon is accessed only if interrupt pending.
2. Call `VpApiTick()` and **`VpGetEvent()`** in the same way as a VE880 application.
   - Note that due to the "real-time" capabilities of the ZL880/miSLIC silicon, the ZL880/miSLIC VP-API-II does not need the user to specify the VP-API-II tickrate (i.e., polling rate of `VpApiTick()`) as the case is with VE880 and other CSLAC devices.

Note that some interrupt modes requires implementation of System Services Functions. Details are provided in **System Service Functions**, on page 157.

For polling/interrupt mode details, refer to "CSLAC VP-API-II Reference Guide", Rev 16, Chapter 7.

### C.7.2 System Service Functions

The following System Service Functions are not required by the ZL880/miSLIC VP-API-II:

- `VpSysWait()` - not used and therefore does not need to be implemented.

The following System Service Functions may be required:

- `VpSysDisableInt() and VpSysEnableInt()`: required if `VpApiTick()` is called and `VP886_INTERRUPT_LEVTRIG_MODE` is set to `#define` in `vp_api_cfg.h`.
- `VpSysTestInt()`: required if `VpApiTick()` is called and `VP886_EFFICIENT_POLLED_MODE` is set to `#define` in `vp_api_cfg.h`.

It is no longer required that the **`VpSysEnterCritical() and VpSysExitCritical()`** functions support “nesting” of critical sections (critical section depth > 1). The ZL880/miSLIC VP-API-II uses only non-nested critical sections (critical section depth is always <= 1).

### C.7.3 VpMpiCmd()

The VE880 series devices require a 2.5 microsecond chipselect off time in between each byte. The ZL880/miSLIC series devices do not have this requirement, so the delay can be removed to improve performance.

Note however for ZSI interface, there is a requirement for NO_OP instruction after each "Read" Command. This is the same for all silicon that supports the ZSI interface. Details of these requirements can be found in Section 12.2.1.

## C.8 NEW FEATURES

These are brief descriptions of new VP-API-II features available for ZL880/miSLIC. For more complete descriptions, see the corresponding sections (accessible by the links provided).

### C.8.1 Events

- **`VP_LINE_EVID_GEN_TIMER`** is generated after the specified delay when the application calls the new function **`VpGenTimerCtrl()`**.
- **`VP_DEV_EVID_EVQ_OFL_FLT`** indicates an event queue overflow.
- **`VP_LINE_EVID_DC_FLT`** indicates a DC fault. This must be enabled by a DC profile setting.
- **`VP_LINE_EVID_LINE_IO_RD_CMP`** indicates completion of a **`VpLineIoAccess()`** read operation (i.e., `direction = VP_IO_READ`).
- **`VP_LINE_EVID_LINE_IO_WR_CMP`** indicates completion of a **`VpLineIoAccess()`** write operation (i.e., `direction = VP_IO_WRITE`).

### C.8.2 Functions

- **`VpGenTimerCtrl()`** uses a device timer to generate an interrupt and event after the specified amount of time.
- **`VpLineIoAccess()`** provides GPIO pin access on a per-line basis. In some cases easier to use than **`VpDeviceIoAccess()`**.
- **`VpQueryImmediate()`** retrieves non-boolean information about a line. Information is provided immediately.
- **`VpGetOptionImmediate()`** provides an alternate interface to **`VpGetOption()`** with results returned immediately rather than with an event.

### C.8.3 Options

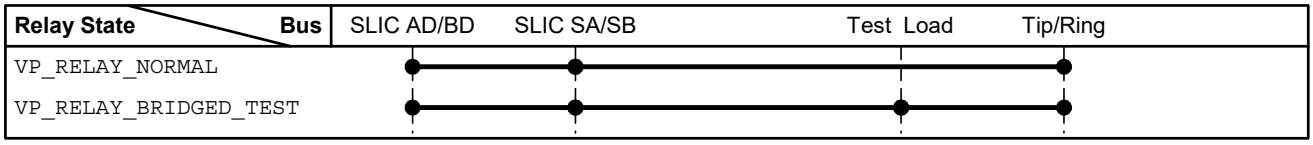
- VP_CODEC_ALAW_WIDEBAND, VP_CODEC_MLAW_WIDEBAND, and VP_CODEC_LINEAR_WIDEBAND modes added to **VP_OPTION_ID_CODEC** to allow different encodings in wideband mode.
- VP_DBG_TIMER added to **VP_OPTION_ID_DEBUG_SELECT** for timer-related debug output.
- **VP_OPTION_ID_LINE_IO_CFG** provides GPIO pin configuration on a per-line basis. In some cases easier to use than **VP_DEVICE_OPTION_ID_DEVICE_IO**.

CHAPTER

# D RELAY CONFIGURATIONS

This appendix describes relay configurations for the ZL880/miSLIC supported line termination types. The relay states described in this section are exercised using the function `VpSetRelayState()`, on page 106. Only those relay states that are described in this section are valid relay states for a given line termination types.

**Figure D–1 VP_TERM_FXS_GENERIC and VP_TERM_FXS_LOW_PWR Terminations**



CHAPTER

# E REVISION HISTORY

## REV 11 - OCTOBER 19, 2017

- In *VP_OPTION_ID_DCFEED_PARAMS*, on page 39, expanded "LP" to "Low Power" in the description of `lpHookThreshold` for clarity.
- Added *VP_DEV_EVID_SYSTEM_FLT*, on page 88.
- Added *VP_DEVICE_OPTION_ID_SPI_ERROR_CTRL*, on page 51.
- Added *VP_DEVICE_OPTION_ID_ACCESS_CTRL*, on page 50.
- Added *VP_STATUS_ACCESS_BLOCKED*, on page 17.
- Added a mention of **`VP_OPTION_ID_PULSE_MODE`** to *VP_LINE_EVID_HOOK_PREQUAL*, on page 92.
- Added **`intProductCode`** to *VpGetDeviceInfo()*, on page 124.

## REV 10 - SEPTEMBER 26, 2017

- Added *VpQuery()*, on page 128.
- Added *VP_LINE_EVID_QUERY_CMP*, on page 98.
- Fixed alignment of timeslot diagram in *VP_OPTION_ID_CODEC*, on page 30.
- Added `VP_OPTION_LB_TIMESLOT_WITH_DAC` to *VP_OPTION_ID_LOOPBACK*, on page 32.
- Added `VP_ADAPT_RING_FULL_TRACKER` to *VP_DEVICE_OPTION_ID_ADAPTIVE_RINGING*, on page 44.
- Fixed example diagrams in *VP_DEVICE_OPTION_ID_DEVICE_IO*, on page 34 and *VpDeviceIoAccess()*, on page 115 where the device GPIO bits were ordered incorrectly. Also changed channel numbering from 1-2 to 0-1 to match the formulas given for calculating bit positions based on the software `channelId` .

## REV 9 - SEPTEMBER 10, 2015

- Removed VP_LINE_EVID_CAL_BUSY, formerly section 8.5.4.
- Added **`VpBatteryBackupMode()`**.
- Corrected several missing or duplicate listings in Figure 1–5, *Function Summary by VP-API-II SW Package*, on page 13.
- Corrected arguments to **`VpObjectDump()`**. The names of pDevCtx and pLineCtx were backwards.
- Added a note to **`VpConfigLine()`** that gain settings will be reset when applying a new AC profile.
- Added `VP_ADAPT_RING_FIXED_TRACKER` to **`VP_DEVICE_OPTION_ID_ADAPTIVE_RINGING`**.
- Clarified the `selectedBat` result and the meanings of `vbat1` and `vbat2` between tracker and ABS for **`VP_LINE_EVID_RD_LOOP`**.

## REV 8 - NOVEMBER 14, 2014

- Replaced instances of "ZL880/Le9662/72" with "ZL880/miSLIC"
- Added Le9652, Le9651, Le9642, and Le9641 to Table 1–3, Supported ZL880/miSLIC Device Configurations, on page 5 and to the miSLIC definition in the *Glossary*, on page 143.
- Added **`VpShutdownDevice()`**.
- Added notes about clock fault and timestamp rollover events being masked in **`VpFreeRun()`**.

- Updated VP_LINE_EVID_DTMF_DIG with details about the detection level reported in the eventData.
- Added a note to VpSetLineState() that VP_LINE_DISABLED may only be set by the application on Tracker devices.
- Added new mode VP_ADAPT_RING_SINGLE_BB_TRACKER to VP_DEVICE_OPTION_ID_ADAPTIVE_RINGING.
- Updated VP_OPTION_ID_DTMF_MODE to describe the upstream and downstream detection modes for new devices.
- Added VP_OPTION_ID_DTMF_PARAMS.
- Added VP_DEVICE_OPTION_ID_FSYNC_RATE.
- Added VP_DEVICE_OPTION_ID_RING_PHASE_SYNC.
- Added VP_OPTION_ID_RINGTRIP_CONFIRM.
- Added a line to VP_OPTION_ID_DEBUG_SELECT to specify that the data type is uint32.
- Re-added VpSysWait() to support VP_DEVICE_OPTION_ID_RING_PHASE_SYNC on older devices.
- Reversed the order of the revision history so that the latest revisions come first.

## REV 7 - JUNE 5, 2014

- Replaced all instances of "VE960" with "miSLIC".
- Added VP_OPTION_ID_DTMF_MODE.
- Added VP_OPTION_ID_HIGHPASS_FILTER.
- Added information on Edge-triggered interrupt handling.
- Updated **VP_STATUS_ERR_SPI** description to include VpGetEvent(). See Table 1–7, VP-API-II Return Codes (VpStatusType), on page 17.
- Added Le9661 to Table 1–3, Supported ZL880/miSLIC Device Configurations, on page 5.
- Updated VP_OPTION_ID_DEBUG_SELECT default value.
- Updated VP_LINE_EVID_DTMF_DIG to refer to the new VP_OPTION_ID_DTMF_MODE option.
- Changed the description for the bat3 result in VP_LINE_EVID_RD_LOOP to "N/A". It does not actually return the IO2 voltage as previously described.
- Updated VP_DEV_EVID_BAT_FLT to include new ZL880/miSLIC specific enumeration values for battery fault codes.
- Added new DTMF debug types to Table 13–1, Debug Flags Used by ZL880/miSLIC VP-API-II, on page 138.
- Added adaptive ringing and DTMF detection compile flags to Table 4–1, Code Size Management Settings (vp_api_cfg.h), on page 59.
- Removed adaptive ringing flag from Table 4–2, ZL880/miSLIC VP-API-II Configuration Settings, on page 60.

## REV 6 - SEPTEMBER 18, 2013

- Corrected timestamp information in events.
- Corrected event VP_LINE_EVID_GND_FLT.
- Modified state diagram for VP_OPTION_ID_GND_FLT_PROTECTION (see Figure 3–3 for details).

## REV 5 - AUG 29, 2013

- Added support for Le9662 and Le9672 silicon (miSLIC series). See Table 1–3 for details.
- Updated option VP_DEVICE_OPTION_ID_ADAPTIVE_RINGING with new **mode** **VP_ADAPT_RING_SHARED_BB_ABS** used for Shared Buckboost ABS Supply configurations only. Removed obsoleted mode **VP_ADAPT_RING_POWER_SLIC** (not necessary).
- Updated description of VpMpiCmd() to include discussion of ZSI requirements. Note that the new silicon (Le9672 and Le9662) supports only the ZSI interface.

- Added Figure 1–3, *ZSI System Architecture Using miSLIC or ZL880 Devices*, on page 7.
- Added return value VP_STATUS_ERR_SPI and description in VpInitDevice().
- Added **offHookMin** to options VP_DEVICE_OPTION_ID_PULSE and VP_DEVICE_OPTION_ID_PULSE2, and associated event VP_LINE_EVID_HOOK_PREQUAL. See Section 3.3.5 for details.
- Improved requirements explanation for VpInitDevice(), VpInitLine() and VpConfigLine().

## REV 4 - MARCH 29, 2013

- Added support for ZL88801, Shared-Tracking Supply Device (see Table 1–3 for details).
- Throughout document (where appropriate) emphasized that all FXS lines MUST be calibrated prior to being put into service. They can be put into VP_LINE_STANDBY prior to calibration, but all detectors and Voice Data must be ignored.
- Added option VP_OPTION_ID_GND_FLT_PROTECTION and associated event VP_LINE_EVID_GND_FLT.
- Added option VP_DEVICE_OPTION_ID_ADAPTIVE_RINGING.
- Corrected information for eventData for event VP_LINE_EVID_USER when I/O2 changes.
- Added the timeStamp output to many events where previously the event member parmHandle was not being used. See Chapter Events for details.
- Rewrites and or Improvements to:
  - Section 1.11: Clarification to VP-API-II Source Version Numbering system.
  - Section 3.2: Most notable is emphasis on VpGetOptionImmediate() instead of VpGetOption() with no loss of information.
  - Section 3.3.3: Provided clarification of debug output (i.e., if used will generally require providing debug information to Microsemi Customer Support).
  - Section 3.3.14: Clarified all possible uses of I/O2 per the Device Profile (see "I/O2 Voltage Monitor Note" and "Default" settings).
  - Section 3.3.18: Corrected description under "Default" values.
  - Section 4.2.2: Changed to use one function call - VpQueryImmediate(), for retrieving Coefficients during "Fast Initialization" procedure (instead of calling three functions)..
  - In cases where the number of return values are very limited, replaced "**RETURNS**: VP-API-II Return Codes (VpStatusType)", with list of corresponding possible return values.
  - Chapter 8: Added names of eventCategoryType struct members that correspond to the item being discussed throughout Chapter.
- Added VP_OPTION_LB_DIGITAL to option VP_OPTION_ID_LOOPBACK.
- Note that VP_OPTION_LB_DIGITAL is NOT supported in the ZL880 VP-API-II (or VP-API-II Lite). This is similar to VE880 - Digital Loopback for ZL880 is supported only in the Microsemi Line Test Software (ZL880SLVVP)

## REV 3 - NOV 13, 2012

- Added options VP_OPTION_ID_DCFEED_PARAMS and VP_OPTION_ID_RINGING_PARAMS.
- Updated description to VP_OPTION_ID_ABS_GAIN and included 'C' code examples.
- Added VP_QUERY_ID_LINE_CAL_COEFF to VpQueryImmediate().
- Updated Design Considerations Chapter Section Fast Initialization (Calibration Bypass) to use VpQueryImmediate() for retrieving calibration coefficients.

## REV 2 - JUNE 15, 2012

- Added Functions: VpLineIoAccess() and VpQueryImmediate().
- Added Options: VP_OPTION_ID_LINE_IO_CFG. Updated "Description" section for option VP_DEVICE_OPTION_ID_DEVICE_IO and corrected "Example 1" (pin order is incorrect for both

directionPins_30_0 and outputTypePins_30_0. In both cases, the middle sets of pins, [ch1, I/O2] and [ch2, I/O1] should be swapped. The I/O-1 pin for both channels are assigned the lower bits in the bit-mask, then the I/O-2 pin for both channels is assigned the next bits in the bit mask, and so on..).

- Added Events: VP_LINE_EVID_LINE_IO_RD_CMP and VP_LINE_EVID_LINE_IO_WR_CMP.

## REV 1 - APR 5, 2012

- Release for ZL880 B2.20.0 VP-API-II Release.

**For more information about all Microsemi products visit our website at www.microsemi.com**

TECHNICAL DOCUMENTATION – NOT FOR RESALE

Microsemi
a Microchip company

**Microsemi Corporate Headquarters**
One Enterprise, Aliso Viejo CA 92656 USA
Within the USA: +1 (949) 380-6100
Sales: +1 (949) 380-6136
Fax: +1 (949) 215-4996

Microsemi Corporation (NASDAQ: MSCC) offers a comprehensive portfolio of semiconductor solutions for: aerospace, defense and security; enterprise and communications; and industrial and alternative energy markets. Products include high-performance, high-reliability analog and RF devices, mixed signal and RF integrated circuits, customizable SoCs, FPGAs, and complete subsystems. Microsemi is headquartered in Aliso Viejo, Calif. Learn more at **www.microsemi.com**.