



ProSLIC[®] API User Guide

V5.0.2

20 April 2018

Copyright © 2018 Silicon Labs

ProSLIC[®] is a registered trademark of Silicon Labs.

Revision History

Date	Version	Description
20 APR 2018	5.0.2	<ul style="list-style-type: none"> Added information about ProSLIC_testLoadTest() Added information about RSPI Abbreviated install instructions of MinGW & Cygwin. Added information on how to incorporate dissimilar EVBs.
15 SEP 2017	5.0.1	<ul style="list-style-type: none"> Added information on new Si3217x & Si3226x converters
05 MAY 2017	5.0.0	<ul style="list-style-type: none"> Updated for uPBX demo Updated documentation on testIn Updated information on MWI (Neon bulb based) Added information on Si3219x family Added information on the LCCB110 designs for Si3218x/Si3228x Updated id_evb information about ISI based EVB's
20 DEC 2016	4.1.3	<ul style="list-style-type: none"> Updated for new converter designs (Si3226x LCUB)
07 OCT 2016	4.1.2	<ul style="list-style-type: none"> Updated information about pform_tester
22 JUN 2016	4.1.1	<ul style="list-style-type: none"> Added information about Lua script interpreter.
10 MAR 2016	4.1.0	<ul style="list-style-type: none"> Added information about addr2line in demo section. Updated table for Si3228x chipset in demo section. Added information about mingw
21 JAN 2016	4.0.0	<ul style="list-style-type: none"> Added references for Si3218x and Si328x. Removed references to Si321x, Si322x, and Si324x chipsets. Added mention on Perl for (optional) menu generation for the API demo. Updated demo section to include SI3050, Si3218x, & Si328x API specification renamed to API User Guide Added rdwr demo code. Added id_evb utility information.
02 OCT 2015	3.14	<ul style="list-style-type: none"> Updated demo documentation.
18 FEB 2015	3.13	<ul style="list-style-type: none"> Updated example RAMWrite (mask for location B0 was incorrect) Updated to include reference to Si3226X where other chipsets are mentioned. Updated demo documentation for new applications, Linux®

		<p>support¹, and converter types.</p> <ul style="list-style-type: none"> • Removed redundant information in the testin & MWI section – since the functions are mentioned in API reference documentation. • Renamed LCQCUK to LCQC to be consistent with other references found in Silicon Labs documentation. • Updated General configuration description and screenshot.
18 JUL 2014	3.12	<ul style="list-style-type: none"> • Updated demo documentation.
27 JUN 2014	3.11	<ul style="list-style-type: none"> • Updated section on gcc/Cygwin demo application to what is needed to install Cygwin
17 JAN 2014	3.10	<ul style="list-style-type: none"> • Added section on gcc/Cygwin demo application

¹ Linus Torvalds is the owner of the Linux trademark – see
<http://www.linuxfoundation.org/programs/legal/trademark/attribution>

Table of Contents

1.0 Introduction	9
1.1 Terminology	9
2.0 Overview	9
3.0 API Usage	12
3.1 ProSLIC API Software Configuration	12
3.2 ProSLIC Device Initialization	13
3.3 ProSLIC Device Configuration	13
3.4 ProSLIC Device Control	14
4.0 ProSLIC API Configuration Tool	16
4.1 ProSLIC API Configuration Tool Overview	16
4.1.1 ProSLIC API Config Tool Menu Items	16
4.1.2 General Parameters	18
4.1.3 ProSLIC API Configuration Tool Presets	19
4.1.4 Creating a Generic Preset Name Enumeration	19
4.1.5 Renaming a Preset	21
4.1.6 Editing/Modifying a Preset	21
4.1.7 Adding a New Preset	21
4.1.8 Setting a Default Preset	22
4.1.9 Generating C Source Code	23
4.1.10 Saving/Loading the Configuration Tool Input Data	23
4.2 ProSLIC API Configuration Tool Features	23
4.2.1 Si3217x General Parameters	23
4.2.1.1 Si3217x General Parameters: Hardware Dependencies Tab	24
4.2.1.2 Si3217x General Parameters: Interrupts Tab	26
4.2.1.3 Si3217x General Parameters: Power/Thermal Alarms Tab	28
4.2.1.4 Si3217x General Parameters: GPIO Tab	29
4.2.2 Si3217x Ringing/Ringtrip Config	30
4.2.3 Si3217x DC Feed Config	33
4.2.4 Si3217x Impedance Configuration	35
4.2.4.1 Coefficient Selection Wizard	36

4.2.4.2	Load from File	38
4.2.5	Si3217x FSK Configuration.....	40
4.2.6	Si3217x Pulse Metering Configuration.....	41
4.2.7	Si3217x Tone Generator Configuration	42
4.2.8	Si3217x PCM Configuration.....	43
4.3	Si3217x FXO ProSLIC API Configuration Tool Features.....	44
4.3.1	Si3217x VDAA General Parameters	44
4.3.2	Si3217x VDAA Country Configuration	45
4.3.3	Si3217x VDAA Audio Path Gain Configuration	47
4.3.4	Si3217x VDAA Ring Detection and Validation Configuration.....	49
4.3.5	Si3217x VDAA PCM Configuration	50
4.3.6	Si3217x VDAA Hybrid Configuration	51
4.4	Working with the ProSLIC API Configuration Tool Output	53
5.0	ProSLIC and Voice DAA Function Definitions.....	56
6.0	SPI Driver Examples	56
6.1	Hardware SPI	56
6.2	Software SPI (GPIO)	57
6.3	Register Read Example.....	61
6.4	Register Write Example	63
6.5	RAM Read Example	64
6.6	RAM Write Example	65
7.0	ProSLIC and VDAA Code Examples.....	67
7.1	Multi-Channel ProSLIC Initialization	67
7.2	Multi-Channel Mixed ProSLIC/VDAA Initialization	70
7.3	Multi-Channel / Multi-Device Mixed ProSLIC/VDAA Initialization.....	73
7.4	Control Function Examples.....	77
7.5	Provisioning Function Examples.....	78
7.6	Differential (isolated) PSTN Detection	79
7.6.1	Compile Time Configuration	79
7.6.2	Object Storage and Initialization	80
7.6.3	Executing the Differential PSTN Detection Test	81
7.6.4	Interpreting Test Results.....	82
7.7	Interrupt Handling	84

7.8 Neon Message Waiting Indication (MWI).....	85
7.8.1 Configuring Neon MWI.....	85
7.8.2 Using Neon MWI.....	86
7.8.3 Disabling Neon MWI After Loop Closure	87
7.8.4 Disabling Neon MWI When Not In Use	87
7.9 Configuring for Wideband Audio	88
7.9.1 Wideband Audio Example.....	88
7.9.2 Wideband Audio Low-Frequency Response.....	89
8.0 Multiple Device Configurations.....	90
9.0 Inward Testing With the ProSLIC API	90
9.1 Inward Test Descriptions	90
9.2 Inward Test Configuration.....	91
9.2.1 Launching the Test-In Configuration Utility	91
9.2.2 Generating Source Code	94
9.2.3 Saving/Reloading Inward Test Configurations	94
9.3 Inward Test API Support.....	95
9.3.1 Additional Files	95
9.3.2 Inward Test Data Structures	95
9.3.3 User Linkage to Inward Test Data Structures	96
9.3.4 Inward Test Data Structure Initialization	96
9.3.5 Inward Test API Definitions.....	97
9.3.5.1 PCM Loopback.....	97
9.3.5.2 DC Feed.....	97
9.3.5.3 Ringing.....	98
9.3.5.4 Battery.....	98
9.3.5.5 Audio	98
9.4 Inward Test Code Example	99
10.0 ProSLIC API Demo Applications.....	101
10.1 Demo directory structure	102
10.2 Linux desktop VMB2 support	102
10.2.1 Debugging with the Linux Desktop	103
10.3 Cygwin VMB1 & VMB2 support	103
10.3.1 Installing Cygwin	104

10.3.2	Debugging with Cygwin	104
10.4	MinGW VMB1 & VMB2 support	105
10.4.1	Installing MinGW	105
10.4.2	Debugging with MinGW	106
10.5	Platform test tool	108
10.6	Building the rdwr example code	110
10.7	Building & using the EVB ID utility	110
10.8	Building the API Demo	111
10.8.1	Building for dissimilar EVB's	112
10.8.2	A word on software build resources included in the API	114
10.8.3	Running the Demo	115
10.8.4	Lua Script Interpreter (Experimental)	117
10.9	uPBX demo	118
10.9.1	Configuring the uPBX demo	119
10.9.2	Building the uPBX demo	119
10.9.3	Example execution	119
10.9.4	Code overview	121
10.10	Debugging the Demo Applications	122

List of Figures

Figure 1.	ProSLIC API Architecture	10
Figure 2.	ProSLIC API Abstraction	11
Figure 3.	Data Type Hierarchy	13
Figure 4.	ProSLIC System Initialization Flow	15
Figure 5.	ProSLIC Configuration Tool Device Select Dialog Box	16
Figure 6.	ProSLIC API Configuration Tool Main Window	18
Figure 7.	Presets with Common Target Region	20
Figure 8.	Generic Preset Group Names	21
Figure 9.	Selecting a Default Preset	22
Figure 10.	Si3217x General Parameters - Hardware Dependencies	24

Figure 11. Si3217x General Parameters – Interrupts	27
Figure 12. Si3217x General Parameters - Power/Thermal Alarms	28
Figure 13. Si3217x General Parameters - GPIOs	29
Figure 14. Si3217x Ringing Configuration Window	31
Figure 15. Si3217x DC Feed Configuration.....	33
Figure 16. Si3217x Impedance Configuration	36
Figure 17. Coefficient Selection Wizard Dialog box	37
Figure 18. Zref model	38
Figure 19. Rfuse & Rprot circuit model.....	38
Figure 20. Si3217x FSK Configuration	40
Figure 21. Si3217x Pulse Metering Configuration	41
Figure 22. Si3217x Tone Generator Configuration.....	42
Figure 23. Si3217x PCM Configuration	43
Figure 24. Si3217x VDAA General Parameter Window	45
Figure 25. Si3217x VDAA Country Configuration Window	47
Figure 26. Si3217x VDAA Audio Gain Configuration Window.....	48
Figure 27. Si3217x VDAA Ring Detection and Validation Configuration Window	50
Figure 28. Si3217x VDAA PCM Configuration Window.....	51
Figure 29. Si3217x VDAA Hybrid Configuration Window	52
Figure 30. "Bit-Banged" SPI Transfer	57
Figure 31. Simplified Neon MWI Flowchart	87
Figure 32. Test-In Configuration Window	92

1.0 Introduction

The ProSLIC Application Programming Interface (API) is a high-level ANSI C-language (C99 or newer) software interface that enables customers to quickly develop application software to configure and control ProSLIC devices. This high-level interface abstracts configuration and control of ProSLIC devices to allow the user to implement device-agnostic application code.

This document contains detailed information on the following topics:

- Architecture of the ProSLIC API
- Using the ProSLIC API
- Information about the demo collateral provided

Previously, information on the Voice DAA software API was provided in the separate *Voice DAA API Specification* document. The Voice DAA API documentation is now contained in the *ProSLIC API User Guide* (this document).

1.1 Terminology

In the strictest sense, an ‘API’ is an interface specification that defines the calling convention by which an application program accesses another software component. In C-language, this calling convention is specified by a function definition (i.e. prototype). By this strict definition, the underlying code (e.g. the C-language function declaration) is the ‘implementation’ of the API.

However, in the interest of brevity, throughout this document the term ‘API’ will be used to refer to both the interface specification and the code that implements the interface.

2.0 Overview

Figure 1 below illustrates the architecture of the ProSLIC API. The ProSLIC API defines an abstracted interface layer between the user’s application code and the ProSLIC device-specific drivers, providing the user with full access to the ProSLIC device’s features without requiring any knowledge of what device-specific registers must be programmed to

Proprietary Information - No dissemination or use without prior written permission from Silicon Labs.

configure and use those features. Generic wrappers are also defined in the ProSLIC API to allow the device drivers to access system-dependent hardware resources, such as system timers. It is strongly recommended that customers interface with the ProSLIC & SiVoice API layers (shown in blue) and not directly to the device driver layer (shown in red) for ProSLIC chipsets. Customers using the VDAA should use the SiVoice for common functions and call the VDAA API's directly.

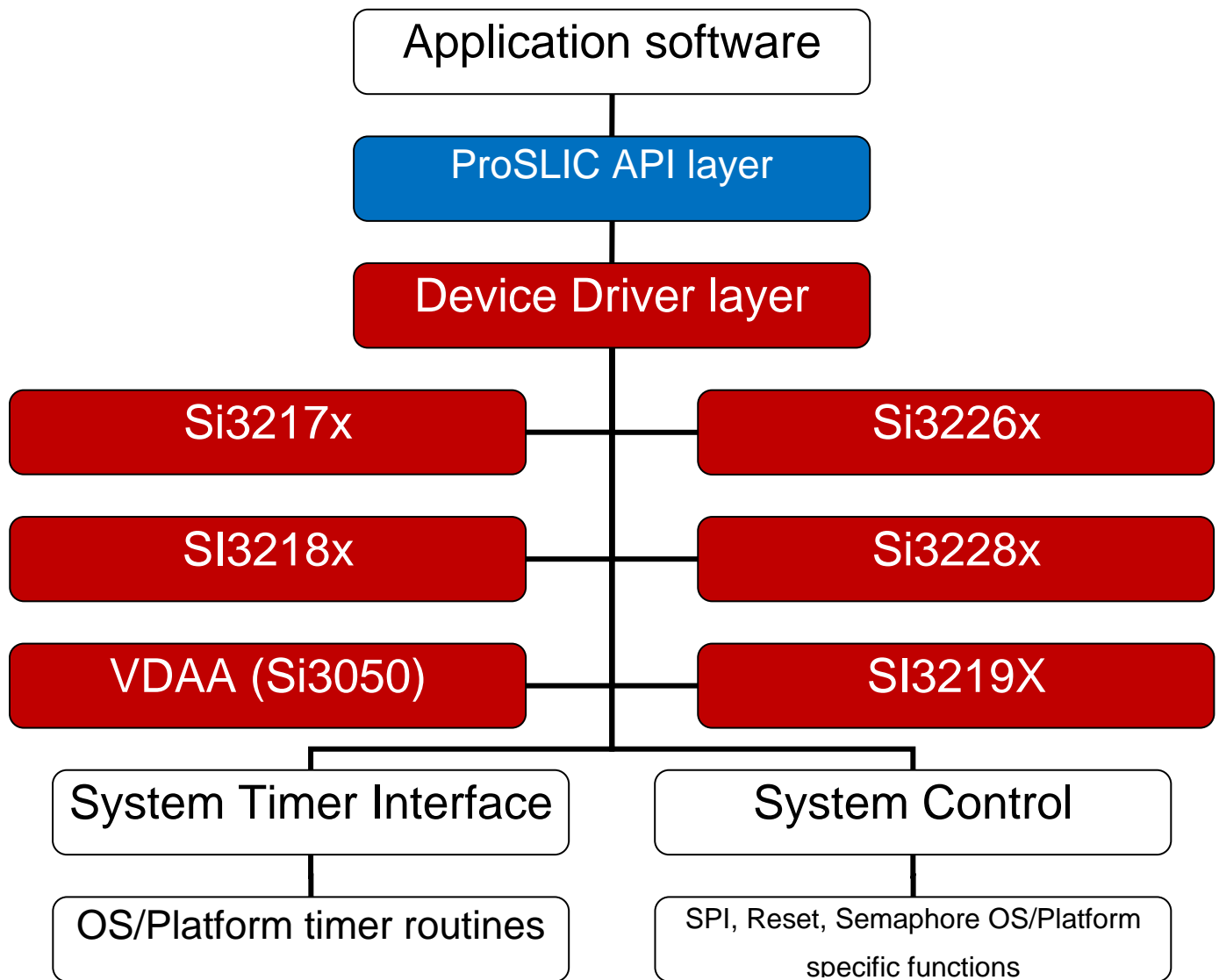


Figure 1. ProSLIC API Architecture

Figure 2 illustrates the flow of control for a typical operation (e.g. switching the ProSLIC to the ringing state).

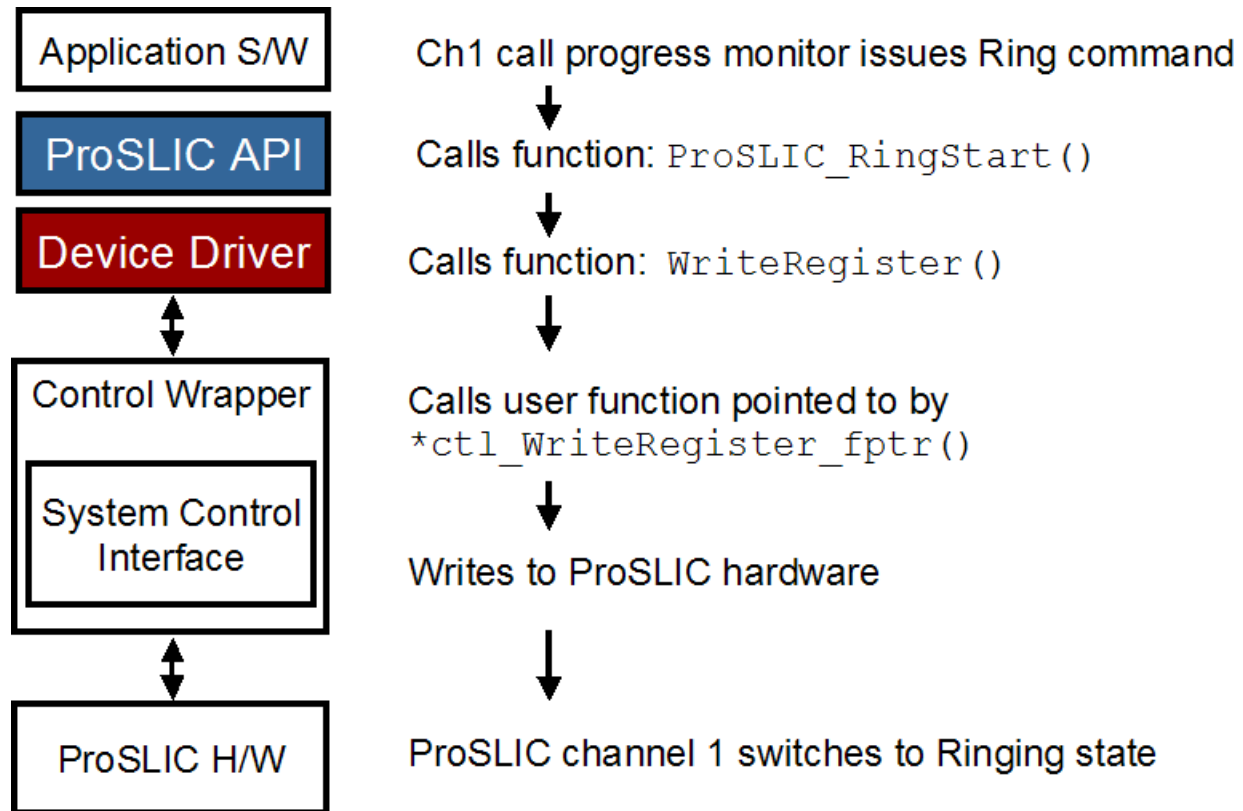


Figure 2. ProSLIC API Abstraction

The customer's application software makes no direct ProSLIC register accesses; instead it calls the device-agnostic `ProSLIC_RingStart()` function. This function invokes the relevant device driver, which determines the required register write(s) and initiates these register writes by calling the user-defined SPI driver function through the generic System Control Interface. The user-defined function executes the physical register write operation and the ProSLIC device enters the Ringing state.

3.0 API Usage

The customer's application software interacts with the ProSLIC API to perform:

- ProSLIC API software configuration
- ProSLIC device initialization
- ProSLIC device configuration
- ProSLIC device control

The following sections describe the various functions implemented in the ProSLIC and Voice DAA APIs.

3.1 *ProSLIC API Software Configuration*

The ProSLIC API utilizes its own data structures for maintaining linkage between the system hardware and the ProSLIC device drivers. Three data structure types must be created and initialized to provide the ProSLIC drivers with this hardware linkage

- ProSLIC Control Interface Object (controlInterfaceType)
 - Provides void pointer to host control object
 - Provides function pointers to system resources (reset, timers, SPI)
 - One instance per control interface (i.e. per chip select)
- ProSLIC Device Interface Object (proslicDeviceType)
 - Provides pointer to ProSLIC control interface object
 - Stores ProSLIC device-specific information (type, revision)
 - One instance per physical device
- ProSLIC Channel Interface Object (proslicChanType)
 - Provides pointer to ProSLIC device interface object
 - Stores channel options and status (enable, error, BOM options)
 - One instance per channel created

The hierarchy for the main structures used to access the ProSLIC is shown in Figure 3. In this example, we have 2 ProSLIC devices with 2 channels each, all sharing a single SPI control interface.

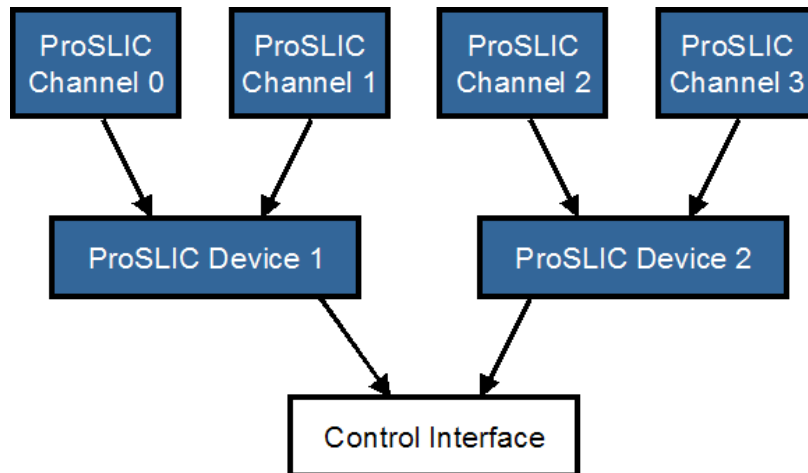


Figure 3. Data Type Hierarchy

3.2 ProSLIC Device Initialization

The ProSLIC API provides function calls to perform a basic initialization of the ProSLIC device. The driver will initialize the ProSLIC device, execute any mandatory calibrations, and implement any customizations made to the General Parameter settings (see *Section 4.0 ProSLIC API Configuration Tool*).

3.3 ProSLIC Device Configuration

The ProSLIC device may be configured to meet the user's design requirements once the basic initialization is complete. The ProSLIC API provides function calls to load the user's precomputed "presets" that are generated using the ProSLIC API Configuration Tool (see *Section 4.0 ProSLIC API Configuration Tool*). The ProSLIC API allows these functions to be called at any time after initialization to dynamically provision the ProSLIC device to meet their application requirements.

3.4 *ProSLIC Device Control*

A vast library of control functions are defined in the ProSLIC API to simplify the implementation control of the ProSLIC device rich feature set. This library allows the user to implement most standard telephony functions with little host software overhead.

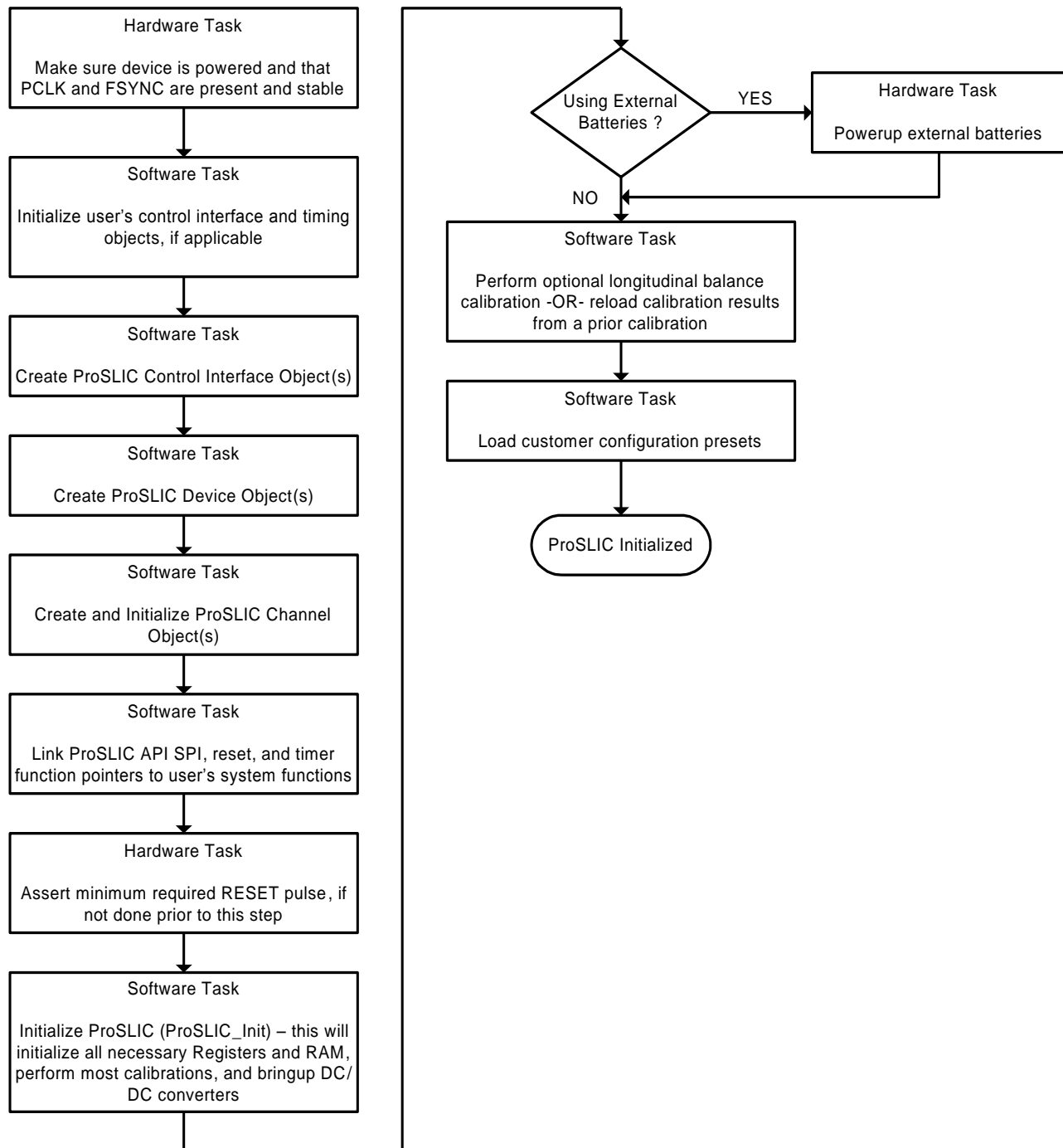


Figure 4. ProSLIC System Initialization Flow

4.0 ProSLIC API Configuration Tool

The ProSLIC API package includes a software utility that simplifies the process of configuring the ProSLIC device to meet the user's design requirements. This utility allows the user to input their design parameters and produce C source code in the form of "presets" that are used by the ProSLIC API to configure the ProSLIC device. This eliminates manual computations by the user and ensures that all necessary register and RAM settings are generated and get properly loaded through the use of the appropriate ProSLIC API function calls.

4.1 ProSLIC API Configuration Tool Overview

After launching the ProSLIC API Configuration Tool, a dialog window prompts the user to select the ProSLIC device they will be using. Once selected, the user is presented with the main window of the ProSLIC API Config Tool, as is shown in Figure 6. Though each ProSLIC device may have unique features only applicable to that device, the process for adding and modifying presets is the same for all devices. For simplicity, this guide will reference the Si3217x's features to present the ProSLIC Configuration Tool usage.

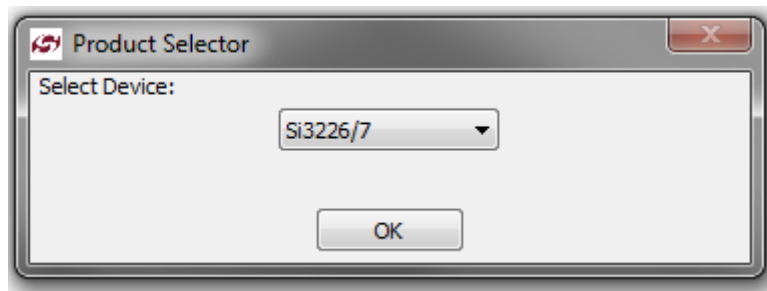


Figure 5. ProSLIC Configuration Tool Device Select Dialog Box

4.1.1 ProSLIC API Config Tool Menu Items

The main window of the Si3217x ProSLIC API Config Tool has two menu options that are described below.

File

<i>Save Config</i>	Saves the current ProSLIC Config Tool session in XML format so it can be reloaded at a later date. It is recommended that customers save their configurations.
<i>Load Config</i>	Loads previous session saved in XML format
<i>Generate Source Code</i>	This generates the *.c and *.h source code to be used by the ProSLIC API.
<i>About</i>	ProSLIC Config Tool version information

Edit

<i>Insert New Preset</i>	Inserts new preset into highlighted function
<i>Delete Preset</i>	Deletes highlighted preset
<i>Rename Preset</i>	Allows the user to rename the selected preset
<i>Edit Preset</i>	Launches the selected preset's configuration window
<i>Create Generic Preset Names</i>	Launches the generic preset group enumeration configuration window

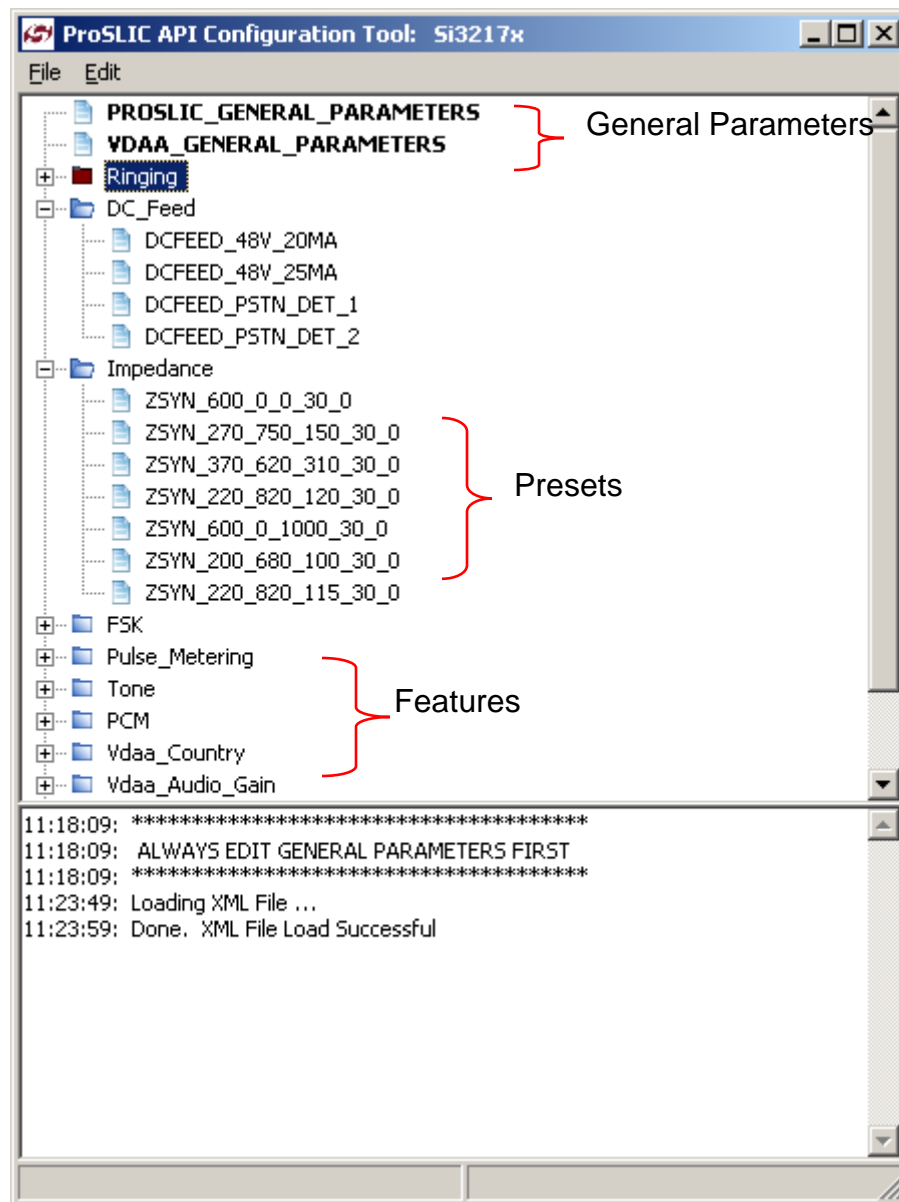


Figure 6. ProSLIC API Configuration Tool Main Window

4.1.2 General Parameters

For each device, there are a set of general parameters – part being used within the family selected, interrupts, power/thermal alarms and GPIO usage in the case of the ProSLIC. These settings differ from presets in that there is only one general parameter setting and multiple presets. These options are used by the device driver during initialization to properly configure hardware dependencies or other options that only need to be configured once and will not change during normal operation.

4.1.3 ProSLIC API Configuration Tool Presets

For each ProSLIC device, there is a set of features (such as Ringing, DC Feed, FSK, etc.) that may be configured using presets generated by the ProSLIC API Configuration Tool. The features that may be programmed using presets are indicated at the top level of the tree in the ProSLIC API main window.

For each supported feature, the user may define up to 32 presets, if using the “Generic Preset Name” feature (see section 4.1.4). To view the presets that are defined for that feature, expand the tree for that feature by left-clicking on the [+]. By default, each feature has 1 preset predefined. If the user does not plan on utilizing a feature in their application, they may leave the default preset unchanged.

The presets are implemented as arrays of data structures in which the preset names are defined as an enumerated list used to index the array of structures. When the user generates the source code, the arrays of data structures will be placed in the .c file, while the enumerated list of preset names are define in the .h file. Each feature will have a unique data structure and enumerated list of preset names.

4.1.4 Creating a Generic Preset Name Enumeration

The user may optionally create an enumerated list of generic preset names. This allows the user to reference presets from multiple features using the same preset name.

Selecting *Edit->Create Generic Preset Names* will launch the Generic Preset Group Enumeration Config window. The user may then enter from 0 to 31 generic names (that must comply with proper c variable syntax). By selecting the checkbox, these generic names will be placed in the .h output file as an enumerated list.

For example, in Figure 7, the presets for ringing, dc feed, and impedance have been created to comply with the regulatory standards for 3 regions. By creating a generic preset group name, the user may reference the generic region name in their code when calling the API functions to setup each feature (see Figure 8).

The generic preset group names would appear in the .h file as shown below.

```
/** Generic Preset Enumeration */
enum {
    GEN_REGION_1,
    GEN_REGION_2,
    GEN_REGION_3
};
```

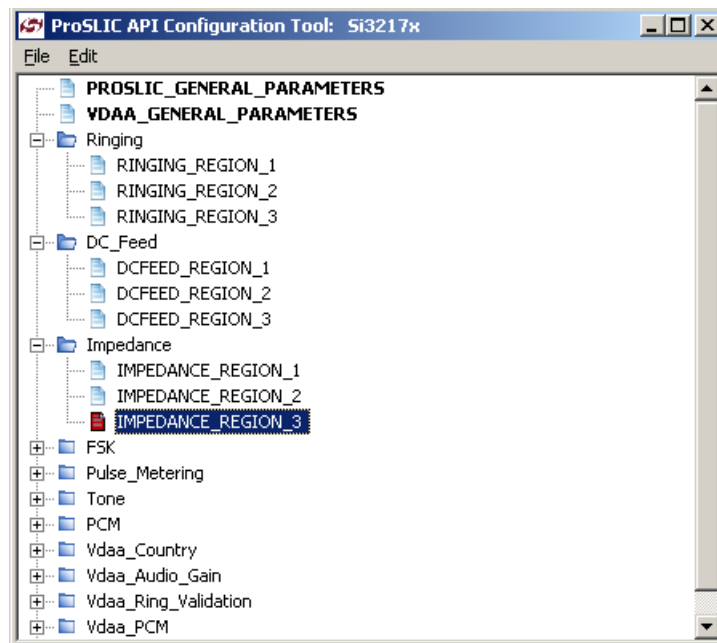


Figure 7. Presets with Common Target Region

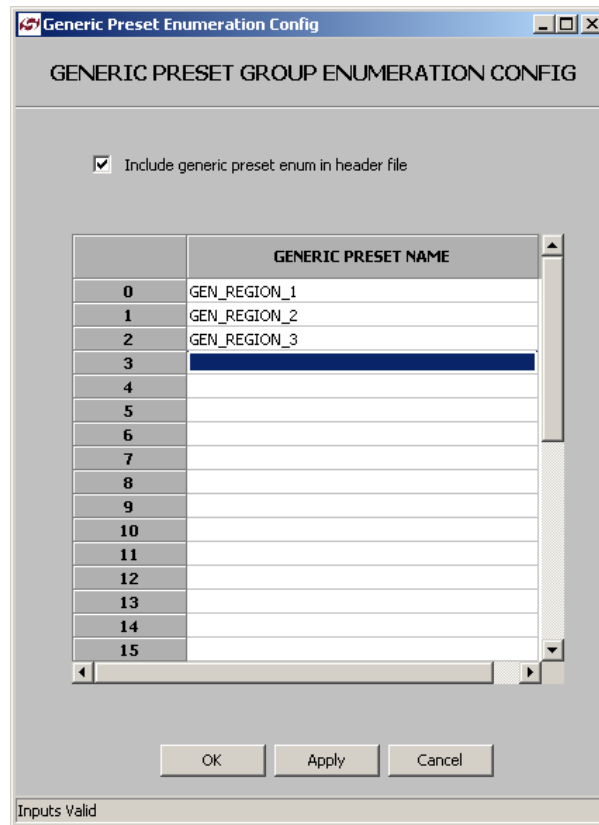


Figure 8. Generic Preset Group Names

4.1.5 Renaming a Preset

The user may rename any preset by highlighting the preset and selecting *Edit -> Rename Preset* or by right clicking and selecting *Rename Preset* from the popup menu

4.1.6 Editing/Modifying a Preset

To edit a preset, double-click on the preset or right click and select *Edit Preset*. This will open a configuration window specific to that feature. The selected preset may also be edited by selecting *Edit -> Edit Preset*.

4.1.7 Adding a New Preset

To add a new preset, select the feature (such as Ringing, FSK), then select *Edit->Insert New Preset* or by right clicking on the feature and selecting *Insert New Preset*. This will add a preset named NEW_CONFIG to the end of the list of presets. The user can then rename and modify this preset.

Proprietary Information - No dissemination or use without prior written permission from Silicon Labs.

4.1.8 Setting a Default Preset

The user may optionally distinguish one preset from each feature as the default preset by highlighting the preset, right-clicking and selecting *Set As Default* from the popup menu. By declaring a preset as the default, a macro is added to the generated header file defining that preset. This is useful if the user wants to abstract a default set of presets to be loaded by default.

For example, in Figure 9 the DC_Feed preset DCFEED_48V_25MA is selected as the default DC_Feed preset. This will result in the following macro being added to the header file once generated using *File->Generate Source Code*.

```
#define INIT_API_PRESET_DC_FEED DCFEED_48V_25MA
```

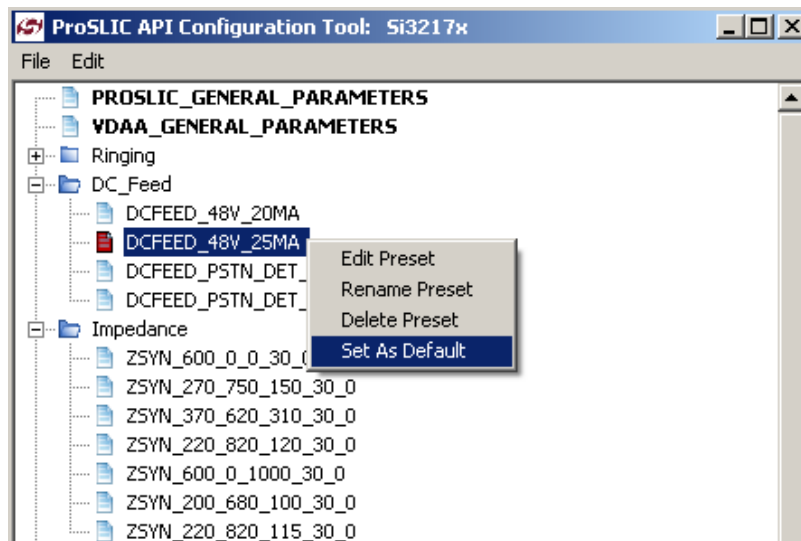


Figure 9. Selecting a Default Preset

The user is not required to specify a default preset for all features.

4.1.9 Generating C Source Code

Once all presets have been edited, the user may now generate C source code to be used in conjunction with the ProSLIC API. To generate source code select *File->Generate Source Code*. A dialog box will appear prompting the user to accept the default name or rename the output files. Only the *.h file is listed in the dialog box, but the *.c file will be stored under the same prefix. In the case of the Si3217x, you will be asked twice – once for the ProSLIC settings and once for the DAA settings.

4.1.10 Saving/Loading the Configuration Tool Input Data

To store input data and preset names so they can be recalled to regenerate C source code in the future, select *File->Save Config*. This will store the user's inputs for each preset so it can be recalled in a future session by selecting *File->Load Config*. The config data is stored in XML format. **It is strongly recommended to save your inputs for future modifications.**

Note: impedance coefficient “load from file” option requires that the files must be located in the same path if the XML files are moved to another computer. Otherwise, when reloaded, the load process will fail to locate the data needed to regenerate the source code.

4.2 ProSLIC API Configuration Tool Features

The following section discusses features specific to the Si3217x FXS portion of the ProSLIC API Configuration Tool, but the process is the same for all devices.

When editing any preset, computations are not made until the **Apply** button is pressed. To accept changes and apply them to that preset, hit **OK**. If **Cancel** is pressed, no changes to that preset are made.

4.2.1 Si3217x General Parameters

Prior to adding or modifying any presets, the ProSLIC General Parameters (and VDAA General Parameters for an FXO-enabled Si3217x device) should be reviewed and updated to match the user's design. This section does not generate a preset, but rather a set of register and RAM settings that are loaded when the *ProSLIC_Init()* and *Vdaa_Init()*

functions are called. These settings pertain to the general operation of the device, hardware options, battery limitations, power thresholds, and GPIO configuration. *This must be updated first*, as selections made here may effect preset computations made elsewhere.

Once selections have been made, close the window by clicking on the “x” in the upper right corner of the window. Specifics of the Si3217x ProSLIC (FXS) General Configuration are detailed below, while details of the Si3217x VDAA (FXO) General Configuration are covered in section 4.3.1

The screenshot shows the 'Si3217x General Parameters' window with the 'HARDWARE DEPENDENCIES' tab selected. The window is divided into two main sections: 'HARDWARE DEPENDANT PARAMETERS' on the left and 'OUTPUTS' on the right. The left section contains several dropdown menus and text input fields for configuring hardware parameters. The right section displays read-only output values for various system options. At the bottom, there are 'OK', 'Cancel', and 'Apply' buttons.

HARDWARE DEPENDANT PARAMETERS		OUTPUTS	
SI32177	PART NUMBER	DCDC_BOM_OPT	BO_DCDC_FLYBACK
NOT INSTALLED	GATE DRIVE CIRCUIT <small>info</small>	GATE_DRIVE_OPT	BO_GDRV_NOT_INSTALLED
FLYBACK	DCDC CONVERTER TYPE	INPUT_RANGE_OPT	VDC_7P0_20P0
VDC 7v-20v	INPUT VOLTAGE RANGE	AUTO_ZCAL	AUTO_ZCAL_ENABLED
3W	MAX POWER	DAA_CNTL	VDAA_DISABLED
136.000	MAX VBAT DURING RINGING ¹	PULSE_METER_OPT	BO_STD_BOM
<small>Max VBATR Supported by Device/BOM = 136</small>			
60.000	MAX VBAT ONHOOK		
STANDARD	AC SENSING OPTION ²		
DISABLED	FXO ENABLE (Si32178/9 RevB Only)		
ENABLED	AUTOMATIC ZCAL		

1. Value programmed during initialization. This may be overwritten by subsequently loaded Ringing presets.
2. See AN340 for Pulse Metering support options

Figure 10. Si3217x General Parameters - Hardware Dependencies

4.2.1.1 Si3217x General Parameters: Hardware Dependencies Tab

The following table describes the inputs to the Hardware Dependencies tab on the Si3217x General Parameters configuration window. Any parameter not listed in the table that is present on the tab is a read-only output.

Proprietary Information - No dissemination or use without prior written permission from Silicon Labs.

Input Parameter	Description
Part Number	What part number is being used. This enables/disables certain capabilities
Gate Drive Circuit	Is the hardware design using a gate driver circuit or not. Care should be taken in selection of this parameter (Si3217x only).
Battery Rail Type	What is the battery rail type – refer to your hardware design.
DC-DC Converter Type	Select DC-DC hardware architecture that matches user's hardware design. Refer to AN340 for description of options.
Input voltage range	What is the voltage range being fed to the converter circuit. In some cases there is only one choice.
MAX POWER	What is the power capability of the converter design.
Max VBAT During Ringing	Maximum VBAT that will be required during ringing. Will be overwritten with the VBAT value supplied with the Ringing preset when loaded.
Max VBAT During Onhook	Enter the required VBAT during the normal onhook state.
AC Sensing Option	Select AC sensing option. See AN340
FXO Enabled	Select this checkbox if using the Si32178 and the FXO port is to be enabled.
Automatic ZCAL	Enable automatic calibration of the Z-synthesis when going off hook. It is suggested to leave this enabled unless you have a specific reason to disable it.

Table 1. Si3217x General Parameters: Hardware Dependencies

4.2.1.2 Si3217x General Parameters: Interrupts Tab

The following table describes the inputs to the Interrupts tab on the Si3217x General Parameters configuration window. Refer to AN340 on descriptions of the interrupts.

Input Parameter	Description
IRQEN1	Select checkbox of each interrupt in IRQ1 to be enabled.
IRQEN2	Select checkbox of each interrupt in IRQ2 to be enabled.
IRQEN3	Select checkbox of each interrupt in IRQ3 to be enabled.
IRQEN4	Select checkbox of each interrupt in IRQ4 to be enabled.

Table 2. Si3217x General Parameters Interrupts

Si3217x General Parameters

Si3217X GENERAL PARAMETERS

HARDWARE DEPENDENCIES | **INTERRUPTS** | **POWER/THERMAL ALARMS** | **GPIO**

IRQEN1 CLEAR ALL SET ALL	<input type="checkbox"/> VBAT_IE	<input type="checkbox"/> RING_TI_IE	<input type="checkbox"/> OSC2_TI_IE	<input type="checkbox"/> OSC1_TI_IE
	<input type="checkbox"/> FSKBUF_AVAIL_IE	<input type="checkbox"/> RING_TA_IE	<input type="checkbox"/> OSC2_TA_IE	<input type="checkbox"/> OSC1_TA_IE
IRQEN2 CLEAR ALL SET ALL	<input type="checkbox"/> RXMDM_IE	<input type="checkbox"/> INDIRECT_RAM_IE	<input type="checkbox"/> VOC_TRACK_IE	<input type="checkbox"/> LCR_IE
	<input type="checkbox"/> TXMDM_IE	<input type="checkbox"/> DTMF_IE	<input type="checkbox"/> LONG_HI_IE	<input type="checkbox"/> RTP_IE
IRQEN3 CLEAR ALL SET ALL	<input checked="" type="checkbox"/> P_HVIC_IE	<input type="checkbox"/> MADC_IE		
	<input checked="" type="checkbox"/> P_THERM_IE			
IRQEN4 CLEAR ALL SET ALL	<input type="checkbox"/> USER_IE[0]	<input type="checkbox"/> USER_IE[2]	<input type="checkbox"/> USER_IE[4]	<input type="checkbox"/> USER_IE[6]
	<input type="checkbox"/> USER_IE[1]	<input type="checkbox"/> USER_IE[3]	<input type="checkbox"/> USER_IE[5]	<input type="checkbox"/> USER_IE[7]

IRQEN1
 IRQEN2
 IRQEN3
 IRQEN4

OK Cancel Apply

Figure 11. Si3217x General Parameters – Interrupts

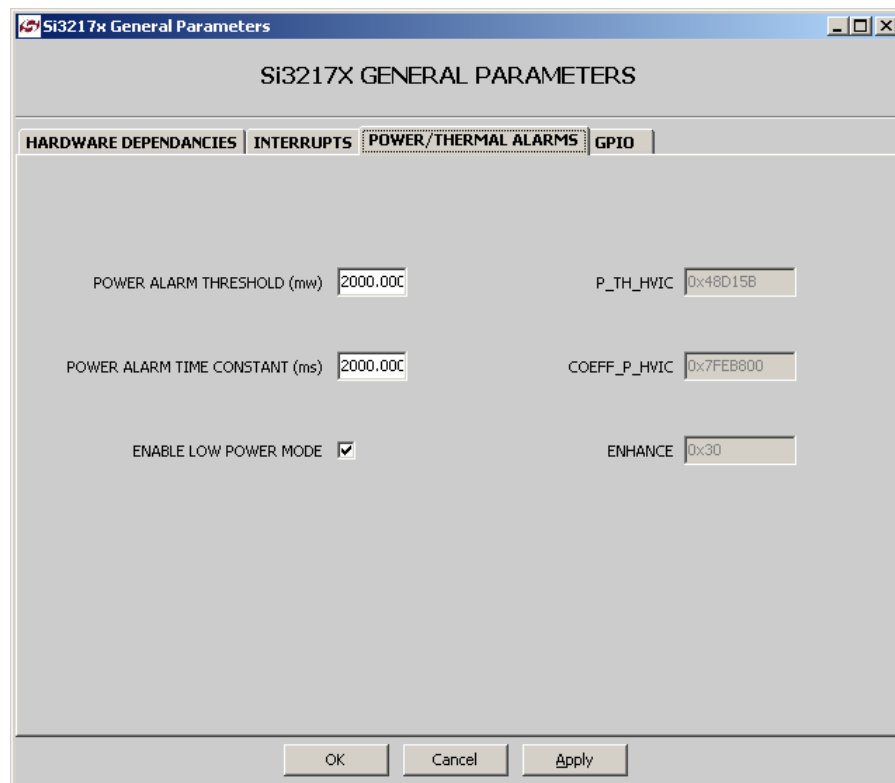


Figure 12. Si3217x General Parameters - Power/Thermal Alarms

4.2.1.3 Si3217x General Parameters: Power/Thermal Alarms Tab

The following table describes the inputs to the Power/Thermal Alarms tab on the Si3217x General Parameters configuration window. Refer to AN340 for information on power and thermal alarms.

Input Parameter	Description
Power Alarm Threshold	Enter the desired power alarm threshold in mW.
Power Alarm Time Constant	Enter the desired time constant in ms.
Enable Low Power Mode	Select this if you with the device to automatically switch to low power mode.

Table 3. Si3217x General Parameters Power/Thermal Alarm

Si3217x General Parameters

Si3217X GENERAL PARAMETERS

HARDWARE DEPENDANCIES | INTERRUPTS | POWER/THERMAL ALARMS | GPIO

MANUAL CONFIGURATION

GPIO	OUTPUT ENABLE	DATA	MODE	DIR	CONTROL	POLARITY	OUTPUT DRIVE
GPIO1	TRI-STATE	0	ANALOG	OUTPUT	AUTOMATIC	POS ACTIVE	NORMAL
GPIO2	TRI-STATE	0	ANALOG	OUTPUT	AUTOMATIC	POS ACTIVE	NORMAL

☐ **CONFIGURE AS COARSE VOLTAGE SENSORS (VTPC/VRINGC)**

GPIO: 0x0 GPIO_CFG1: 0x60 GPIO_CFG2: 0x0 GPIO_CFG3: 0x0

OK Cancel Apply

Figure 13. Si3217x General Parameters - GPIOs

4.2.1.4 Si3217x General Parameters: GPIO Tab

The following table describes the selections to be made for GPIO1 and GPIO2 in the GPIO tab of the Si3217x General Parameters configuration window. Refer to AN340 for information on GPIO configuration. If GPIOs are not used, no modification to this tab is necessary. By selecting Configure as Course Voltage Sensors, the GPIO parameters are automatically configured as coarse voltage sensors. See AN340 for applicable BOM requirements.

Input Parameter	Description
Output Enable	Select TRI-STATE or ENABLED
Data	Select initial data (only applicable if output)
Mode	Select DIGITAL or ANALOG
Dir	Select OUTPUT or INPUT direction
Control	Select AUTOMATIC or MANUAL
Polarity	Select POS or NEG
Output Drive	Select NORMAL or OPEN DRAIN

Table 4. Si3217x General Parameters GPIO

4.2.2 Si3217x Ringing/Ringtrip Config

The Ringing/Ringtrip configuration window is used to edit Ringing presets (see Figure 14).

Table 5 describes the user inputs to the Ringing/Ringtrip configuration window.

Ring Tool

RINGING/RINGTRIP CONFIG

RINGER SETTINGS:

RINGING TYPE: LPR

RINGING WAVESHAPE: SINE

CREST FACTOR: 1.414

RING VOLTAGE: 45.000 *vrms (@ load)*

RINGING DC OFFSET: 0.000 *vdc*

FREQUENCY: 20.000 *Hz*

RING SOURCE IMPEDANCE: 100 *ohms*

ACTIVE TIMER ENABLE: ☐

INACTIVE TIMER ENABLE: ☐

RING ACTIVE TIME: 2.000 *sec*

RING INACTIVE TIME: 4.000 *sec*

SILENT PERIOD STATE: OHT

LOOP/LOAD CONDITIONS:

LOOP LENGTH: 500.000 ☒ ft ☐ m

WIRE UNIT RESISTANCE: 0.044 *ohms/ft*

MAX REN LOAD: 5

CPE RESISTANCE: 600 *ohms*

PROTECTION RESISTANCE: 30 *ohms*

RINGER COMPUTATIONS:

VBATR REQUESTED: 83.519 *v*

VBATR APPLIED: 83.519 *v*

VSRC REQUIRED: 71.519 *Vpk*

RINGTRIP TYPE: AC

AC RT CURRENT: 57.922 *mA*

DC RT CURRENT: 450.000 *mA*

WARNING FLAGS:

OK Cancel Apply

API REG/RAM SETTINGS:

RTPER: 0x50000	RTACDB: 0x6000	ADAP_RING_MIN_I: 0x0	
RINGFR: 0x7EFE000	RTDCDB: 0x6000	COUNTER_IRING_VAL: 0x3000	
RINGAMP: 0x1B9D62	VOV_RING_BAT: 0xC49BA0	COUNTER_VTR_VAL: 0x51EB8	RINGCON: 0x40
RINGPHAS: 0x0	VOV_RING_GND: 0x0	DCDC_VREF_MIN_RNG: 0x1893740	RINGTALO: 0x80
RINGOF: 0x0	VBATR_EXPECT: 0x5585FBE	DCDC_RINGTYPE: 0x200000	RINGTAHI: 0x3E
SLOPE_RING: 0x15E5200E	CONST_028: 0x0	RRD_DELAY: 0x0	RINGTILO: 0x0
IRING_LIM: 0xD16348	CONST_032: 0x0	RRD_DELAY2: 0x0	RINGTIHI: 0x7D
RTACTH: 0x68E2E1	CONST_038: 0x0	VCM_RING: 0x2AC2FDF	USERSTAT: 0x1
RTDCTH: 0xFFFFFFFF	CONST_046: 0x0	VCM_RING_FIXED: 0x2AC2FDF	
VOV_DCDC_SLOPE: 	VOV_DCDC_OS: 	VOV_RING_BAT_MAX: 	DELTA_VCM: 0x3126E8

WARNINGS:

Inputs Valid

Figure 14. Si3217x Ringing Configuration Window

Input Parameter	Description
Ringing Type	Select LPR, BAL, UNBAL or Smart Ringing (supported on some Si3228x chipsets)
Ringing Waveshape	Select SINE (sinusoidal) or TRAP (trapezoidal)
Crest Factor	Enter desired crest factor if Ringing Waveshape is TRAP (automatically set to 1.414 for SINE)
Ring Voltage	Enter the desired rms voltage <i>at the specified REN load</i> at the end of the specified loop.
Ringing DC Offset	Enter the desire DC offset component of the ring signal
Frequency	Enter the desired ringing frequency in Hz
Ring Source Impedance	Enter the desired ringer source impedance (limited from 100Ω to 320Ω)
Active Timer Enable	Select if using Si3217x ring timers for cadencing.
Inactive Timer Enable	Select if using Si3217x ring timers for cadencing.
Ring Active Time	Enter ring active time if using Si3217x ring timers.
Ring Inactive Time	Enter ring inactive time if using Si3217x ring timers.
Silent Period State	Select linefeed state to be automatically switched to during the ring inactive period if using Si3217x ring timers. The choices are OHT (onhook transmission) and active.
Loop Length	Enter length of loop that must be supported with specified Ring Voltage, in feet or meters (if selected)
Wire Unit Resistance	Enter the unit resistance of the loop in Ω/ft
Max REN Load	Select maximum REN at which Ring Voltage must be supported.
CPE Resistance	Enter estimate of maximum CPE offhook resistance (excluding loop impedance) in Ω
Protection Resistance	Enter total series resistance of protection resistance (sum of both TIP and RING paths).

Table 5. Ringing/Ringtrip Configuration Inputs

4.2.3 Si3217x DC Feed Config

The DC Feed configuration window is used to edit DC Feed presets. An expandable graph plotting the I/V characteristics of the dc feed for the given input values is continuously updated. If at any time a combination of input parameters results in an illegal dc feed slope(s), the input parameter name is highlighted in **red**. Table 6 describes the input parameters, but refer to AN340 for a detailed description of these parameters and their design constraints.

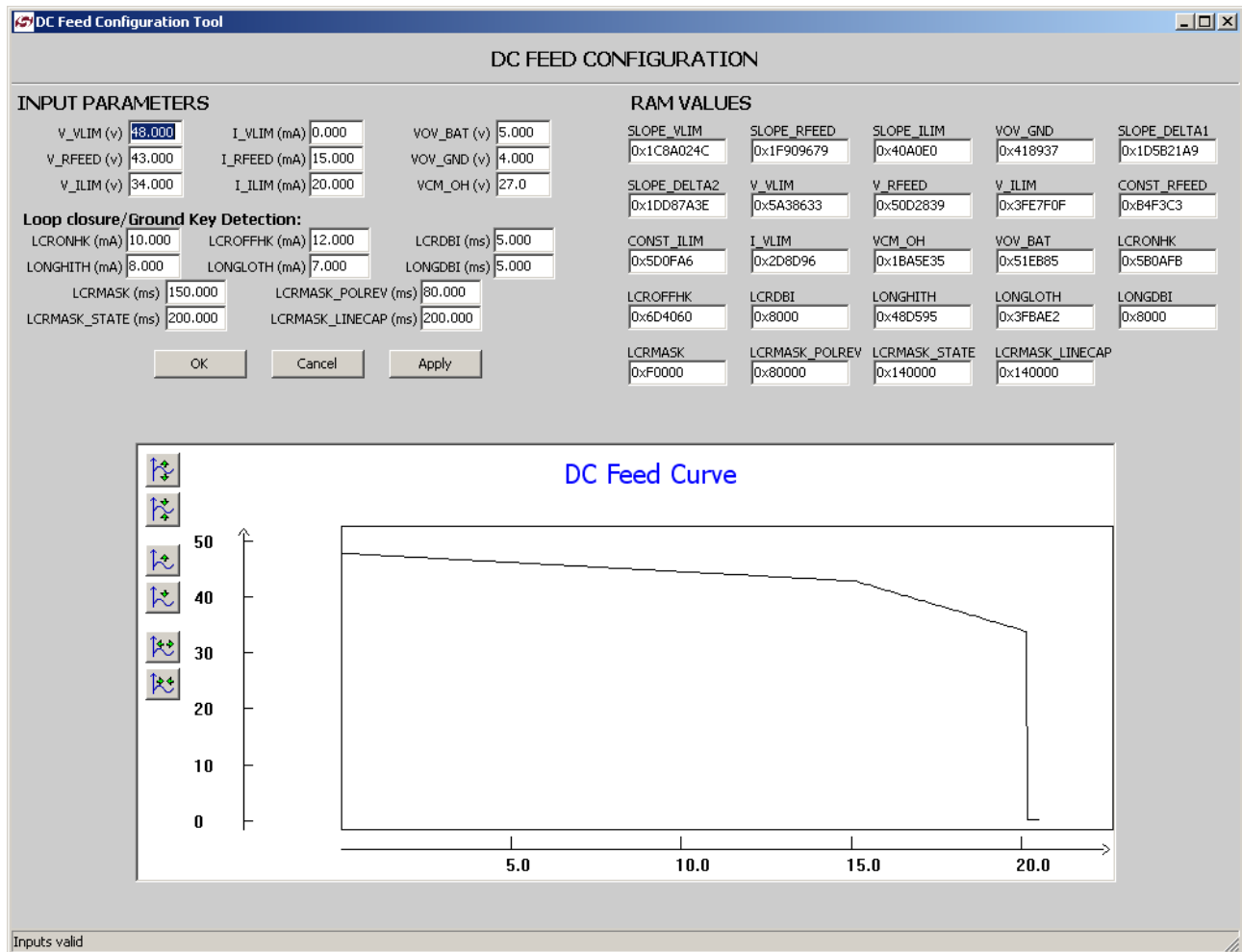


Figure 15. Si3217x DC Feed Configuration

Input Parameter	Description
V_VLIM	Enter onhook, open circuit voltage in v
V_RFEED	Enter voltage at boundary of constant voltage and resistive region in v
V_ILIM	Enter voltage at boundary of resistive and constant current region in v
I_VLIM	Enter current at open circuit voltage in mA
I_RFEED	Enter current at boundary of constant voltage and resistive region in mA
I_ILIM	Enter short circuit current in mA
VOV_BAT	Enter battery overhead voltage in v
VOV_GND	Enter ground overhead voltage in v
VCM_OH	Enter common mode voltage in v
LCRONHK	Enter loop open threshold voltage (offhook to onhook transition) in mA
LCROFFHK	Enter loop closure threshold voltage (onhook to offhook voltage) in mA
LCRDBI	Enter loop closure debounce interval in ms
LONGHITH	Enter ground key detection threshold in mA
LONGLOTH	Enter ground key removal threshold in mA
LONGDBI	Enter ground key detection debounce interval in ms
LCRMASK	Enter loop closure mask interval after in ms
LCRMASK_POLREV	Enter loop closure mask interval after polarity reversal in ms
LCRMASK_STATE	Enter loop closure mask after state change, other than OPEN, in ms
LCRMASK_LINECAP	Enter loop closure mask after entry into the OPEN state, in ms

Table 6. Si3217x DC Feed Configuration Inputs

4.2.4 Si3217x Impedance Configuration

The Impedance configuration window is used to select the desired impedance coefficients. The default coefficients for a new preset are for a 600Ω impedance. The user may load a different set of coefficients by selecting either *Search Database* or *Load from file* button. Once the user has finished selecting the desired settings, click on *OK* or click *Cancel* to abort the operation.

In addition, the user may specify the desired RX and TX path gain. Changing these gains DOES NOT change the impedance coefficients, but the requested gains are automatically applied when `ProSLIC_ZsynthSetup()` is called using the existing gain provisioning functions available in the ProSLIC API. The user may also specify the desired path gains using these provisioning functions, regardless of what gain is specified in the preset. By default, a 1dB gain resolution is possible. If Gain Resolution is selected to be 0.1dB, the user must add the following macro to *prosllic_api_config.h*, otherwise the gains will be rounded down to the nearest integer.

```
#define ENABLE_HIRES_GAIN
```

IMPEDANCE SYNTHESIS/GAIN PLAN CONFIG

SEARCH DATABASE Search database using Coefficient Wizard

LOAD FROM FILE Load coefficients from file

0.1 dB Gain Resolution

-3.2 Desired TX Path Gain (LINE to PCM)*

-4.5 Desired RX Path Gain (PCM to LINE)*

* Gain applied at runtime when preset is loaded
Gain may also be modified at runtime using API calls

ZSYNTH_B0	0x810E00	ZSYNTH_B1	0x1EFE8E80	ZSYNTH_B2	0x803500
ZSYNTH_A1	0xFF66D00	ZSYNTH_A2	0x18099080	ECFIR_C2	0x27CB00
ECFIR_C3	0x1F8A8880	ECFIR_C4	0x2801180	ECFIR_C5	0x1F625C80
ECFIR_C6	0x314FB00	ECFIR_C7	0x1E6B8E80	ECFIR_C8	0xC5FF00
ECFIR_C9	0x1FC96F00	ECIIR_B0	0x1FFD1200	ECIIR_B1	0x23C00
ECIIR_A1	0xED29D00	ECIIR_A2	0x192A9400	RXACEQ_C0	0x7EF5000
RXACEQ_C1	0x13F580	RXACEQ_C2	0x1FFDE000	RXACEQ_C3	0x1FFCB280
RXACHPF_B0_1	0x7ABE580	RXACHPF_B1_1	0x18541B00	RXACHPF_A1_1	0x757CB00
TXACEQ_C0	0x7F46C00	TXACEQ_C1	0xE4600	TXACEQ_C2	0x8580
TXACEQ_C3	0x1FFD6100	TXACGAIN	0x88E0D80	RXACGAIN	0x1456D80
TXACGAIN_SAVE	0x1456D80	RA	0x59		

OK Apply Cancel

Inputs Valid

Figure 16. Si3217x Impedance Configuration

4.2.4.1 Coefficient Selection Wizard

Search Database will open a secondary dialog box with the Coefficient Selection Wizard as shown below:

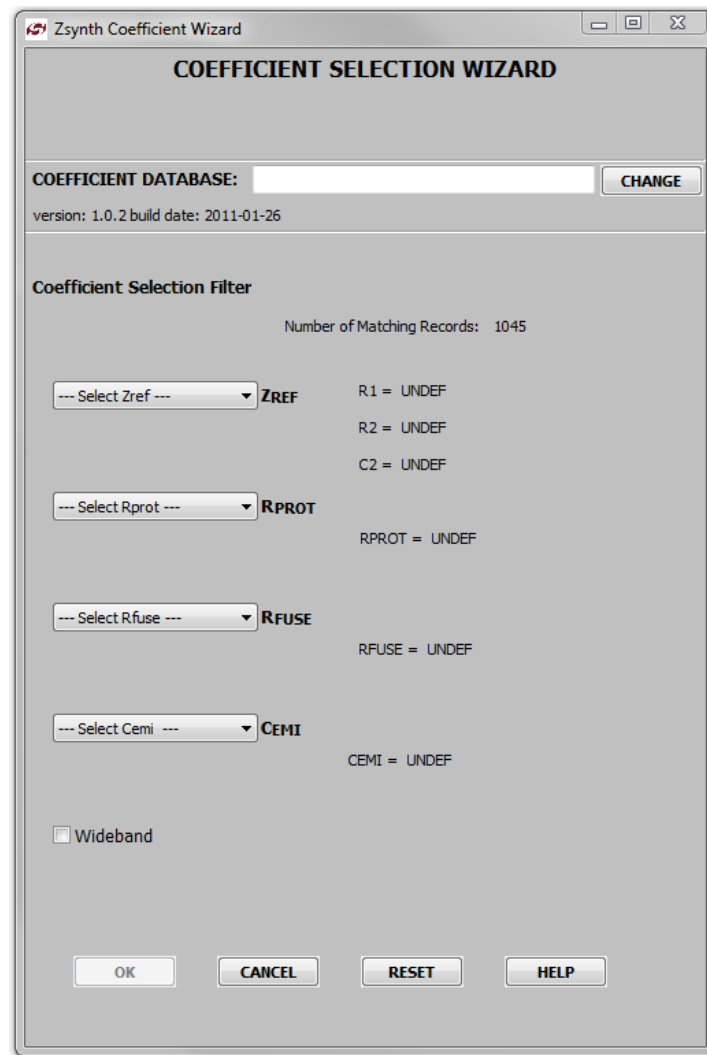


Figure 17. Coefficient Selection Wizard Dialog box

This dialog box has the following information: location of the coefficient database, the version of the database, the number of matching coefficients/records and the pull downs for the various parameters that are used to query the database (Zref, Rprot, Rfuse, Cemi, and wideband). Click on the *OK* or *Cancel* button to exit this dialog box.

Until there is just one record/coefficient available, the *OK* button will be deselected. This may require the user to select all three or fewer parameters. As the user selects the various parameters, the number of matching records/coefficients available is updated.

Z_{ref} is the characteristic line impedance as follows:

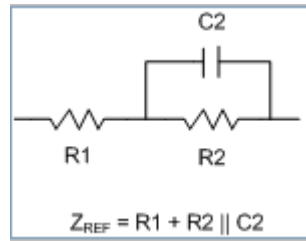


Figure 18. Zref model

Where $R1$ and $R2$ are in terms of Ω and $C2$ is in terms of nF. For example, 180_630_60 is $R1=180\Omega$, $R2=630\Omega$ and $C2$ is 60 nF.

R_{prot} is the total impedance in ohms of the protection circuitry and R_{fuse} is the total impedance of the overcurrent protection device. The diagram below shows the relationship of the model to the implemented circuitry:

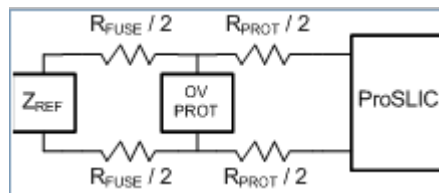


Figure 19. Rfuse & Rprot circuit model

C_{emi} is the EMI capacitor value in nF. Please refer to your design to select the correct value. Typically this value is either 0 nF or 10 nF.

The wideband checkbox is used in conjunction with the PCM wideband setting. The coefficients selected with this option were calculated and tested to work beyond the normal narrowband settings.

4.2.4.2 Load from File

This will open a file open dialog box and the user may select the desired precomputed coefficient file. The precomputed coefficients assume the $R_1 + R_2 || C_2$ impedance model. The naming convention for the precomputed coefficient files is

<Device Family>_<R₁>_<R₂>_<C₂>_<R_{PROT}>_<R_{FUSE}>.txt

For example, the file containing the ETSI harmonized impedance model for a BOM with 15Ω series protection resistors and using fuses of ~ 0Ω would have the filename

Si3217x_270_750_150_30_0.txt

4.2.5 Si3217x FSK Configuration

The FSK configuration window is used to setup the FSK tone generator. The inputs to the FSK configuration window are described in Table 7.

Figure 20. Si3217x FSK Configuration

Input Parameter	Description
FSK FIFO Buffer Depth	Select FIFO buffer depth to generate interrupt (1-8)
FSK Space Frequency	Enter “space” frequency in Hz
FSK Mark Frequency	Enter “mark” frequency in Hz
Level	Enter level in vrms
Disable Start/Stop Bits	Select to disable start/stop bits

Table 7. Si3217x FSK Configuration Inputs

4.2.6 Si3217x Pulse Metering Configuration

The Pulse Metering configuration window is used to setup the Pulse Metering feature of the Si32178 (only) tone generator. The inputs to the Pulse Metering configuration window are described in Table 8.

Figure 21. Si3217x Pulse Metering Configuration

Input Parameter	Description
Metering Amplitude	Enter meter level in mVrms
Metering Frequency	Select metering frequency
Metering Ramp Rate	Select metering ramp rate
Metering Calibration	Select calibration force option
Metering Power Save Mode	Select power save mode

Table 8. Si3217x Pulse Metering Configuration Inputs

Proprietary Information - No dissemination or use without prior written permission from Silicon Labs.

Copyright © 2018 Silicon Labs

4.2.7 Si3217x Tone Generator Configuration

The Tone Generator configuration window is used to setup Tone Generator presets. The inputs to the Tone Generator configuration window are described in Table 9. Input parameter descriptions are the same for both oscillators.

ProSLIC Tone Generator Config

TONE GENERATOR CONFIG

	OSCILLATOR 1	OSCILLATOR 2
AMPLITUDE (dBm)	-18	-18
FREQUENCY (Hz)	350	440
TIME ON (ms)	0	0
TIME OFF (ms)	0	0
SIGNAL ROUTING	RX PATH (to line)	RX PATH (to line)
OSC TURN OFF	ZERO CROSSING	ZERO CROSSING

Note: Oscillators and oscillator timers are enabled at runtime through API calls

OSC1AMP	0xC6000	O1TALO	0x0	O2TALO	0x0
OSC1FREQ	0x7B30000	O1TAHI	0x0	O2TAHI	0x0
OSC1PHAS	0x0	O1TILO	0x0	O2TILO	0x0
OSC2AMP	0xFA000	O1TIHI	0x0	O2TIHI	0x0
OSC2FREQ	0x7870000				
OSC2PHAS	0x0	OMODE	0x66		

OK Apply Cancel

Inputs Valid

Figure 22. Si3217x Tone Generator Configuration

Input Parameter	Description
Amplitude	Enter tone amplitude in dBm
Frequency	Enter tone frequency in Hz
Time On	Enter tone active time if using oscillator timers – in ms

Time Off	Enter tone inactive time if using oscillator timers – in ms
Signal Routing	Select if tone is to be routed to TX, RX, Both or none (disconnected)
OSC Turn OFF	Select to disable feature to only turn off tone when zero-voltage crossing is detected (zero crossing) or Abrupt termination.

Table 9. Si3217x Tone Generator Configuration Inputs

4.2.8 Si3217x PCM Configuration

The PCM configuration window is used to setup the PCM presets. The inputs to the PCM configuration window are described in. Note that PCM timeslots are not part of the PCM preset since these will be unique to each channel in the system.

PCM Configuration

PCM CONFIGURATION

DTX TRISTATE EDGE: PCLK RISING

DTX DRIVE EDGE: PCLK RISING

PCM DATA FORMAT: u-LAW

A-LAW OPERATING MODE: INVERT NONE

WIDEBAND: DISABLED

PCMMODE: 0x1

ENHANCE: 0x0

PCMTXHI: 0x0

Note: PCM enable and timeslots are only configurable through API calls

OK Apply Cancel

Figure 23. Si3217x PCM Configuration

Input Parameter	Description
DTX Tristate Edge	Select PCLK edge in which DTX tri-states
DTX Drive Edge	Select which edge the data should be sent relative to the clock

Proprietary Information - No dissemination or use without prior written permission from Silicon Labs.

PCM Format	Select coding format
Alaw	Select A-Law operating mode
Wideband	Is wideband mode to be enabled (depends on chipset support)

Table 10. Si3217x PCM Configuration Inputs

4.3 Si3217x FXO ProSLIC API Configuration Tool Features

The following section discusses features specific to the Si3217x FXO portion of the ProSLIC API Configuration Tool.

4.3.1 Si3217x VDAA General Parameters

General Parameters specific to the FXO portion of the Si3217x device can be customized using the VDAA General Parameters window. Table 11 below details the options that may be customizable options presented to the user in the VDAA General Parameters window.

Input Parameter	Description
Interrupt Pin	Select Enabled or Disabled (AOUT for Si3050)
Interrupt Polarity	Select INT pin active state to Active LOW or Active HIGH
Resistor Calibration	Enable or Disable resistor calibration
High-Speed Sample Mode	Select audio sample rate
Off-hook Speed	Select offhook speed
Line Voltage Force Disable	Select Normal Operation or Disable LVS forced 0's below 3v
CVI Interrupt Source	Select CVI interrupt source to be LCS2 or LVS
CVI Interrupt Polarity	Select polarity of CVI source versus threshold that generates interrupt.
Guarded Clear	Enable or disable guarded clear
TX/RX IIR Filter	Select between FIR and IIR filter
Fullscale Level	Select fullscale audio level
RX HPF Filter Pole	Select RX HPF Pole frequency
Hybrid Gain Adjust	Enable or disable hybrid gain adjust

AOUT PWM Mode	Set AOUT PWM mode (Si3050 only – ignored on Si3217x)
AOUT PWM Enable	Enable or disable AOUT PWM (Si3050 only – ignored on Si3217x)
SDO Tristate Edge	Set SDO tristate edge (Si3050 only – ignored on Si3217x)

Table 11. Si3217x VDAA General Parameters

VOICE DAA GENERAL CONFIGURATION

INTERRUPT PIN	Disabled or AOUT on Si3050	INTE	0		
INTERRUPT PIN POLARITY	Active LOW	INTP	0		
RESISTOR CALIBRATION	Enabled	RCALD	0		
HIGH-SPEED SAMPLE MODE	8 kHz	HSSM	0		
OFF-HOOK SPEED	128 ms	FOH	1		
LINE VOLTAGE FORCE DISABLE	Normal Operation	LVFD	0		
CVI INTERRUPT SOURCE	LCS2 Current	CVS	0		
CVI SOURCE INTERRUPT POLARITY	Above CVT	CVP	1		
GUARDED CLEAR	Disabled	GCE	0		
TX/RX IIR FILTER	IIR Disabled (FIR Enabled)	IIRE	0		
FULLSCALE LEVEL	0 dBm	FULL	0	FULL2	0
RX HPF FILTER POLE	200 Hz	FILT	1		
HYBRID GAIN ADJUST	0 dB	RG1	0		
AOUT PWM MODE	Delta-Sigma 16.384 MHz	PWMM	0		
AOUT PWM ENABLE	Disabled	PWME	0		
SDO TRISTATE EDGE	/CS Rising Edge	SPIM	0		

OK Apply Cancel

Figure 24. Si3217x VDAA General Parameter Window

4.3.2 Si3217x VDAA Country Configuration

The Country configuration window is used to select the FXO parameters for a selected country. By selecting **Use Default Country Configs**, the user may select from a precompiled list of field proven country settings. If **Customize Country Configs** is

Proprietary Information - No dissemination or use without prior written permission from Silicon Labs.

selected, the user may customize the default parameters from the presently selected country. Figure 25 shows the VDAA Country Config window and Table 12 lists the customizable parameters.

Input Parameter	Description
Ringer Impedance	Select ringer impedance
DC Termination	Select DC termination
AC Termination	Select AC termination
TIP-RING Voltage Adjust	Select DCT pin voltage adjust
Min Operating Loop Current	Select minimum operating loop current
Current Limiting	Enable or disable current limiting
Hybrid Enable	Enable or disable hybrid
Off-hook Speed	Select off-hook speed

Table 12. Si3217x VDAA Country Configuration Parameters

VOICE DAA COUNTRY CONFIG

USE DEFAULT COUNTRY CONFIGS ☒

COUNTRY PROFILE: SWITZERLAND

CUSTOMIZE COUNTRY CONFIGS ☐

RINGER IMPEDANCE	High Impedance (Default)	RZ	0		
DC TERMINATION	50 ohms (Default)	DCR	0		
AC TERMINATION	270 + (750 150nF)	ACIM	2		
TIP/RING VOLTAGE ADJUST	3.50 V	DCV	3		
MIN OPERATING LOOP CURRENT	10 mA	MINI	0		
CURRENT LIMITING	Enabled	ILIM	1		
HYBRID ENABLE	Enabled	HBE	1		
OFF-HOOK SPEED	3 ms	OHS	0	OHS2	1
		SQ0	0	SQ1	0

OK Apply Cancel

Figure 25. Si3217x VDAA Country Configuration Window

4.3.3 Si3217x VDAA Audio Path Gain Configuration

The Audio Path Gain configuration window is used to setup audio gain presets. These presets may be used to setup the TX and RX path independently. Table 13 lists the configurable audio gain parameters.

Input Parameter	Description
Coarse Gain	Select Gain or Attenuation and coarse gain to be applied
Fine Gain	Select Gain or Attenuation and fine gain to be applied
Total Gain	Indicates total path gain with current coarse and fine gain selection
AOUT Gain	Selects gain of AOUT path (Si3050 only)

Table 13. Si3217x VDAA Audio Gain Configuration Parameters

Voice DAA Audio Gain Configuration

AUDIO PATH GAIN
TX or RX GAIN

COARSE GAIN

SELECT
☐ GAIN
☒ ATTN

3dB xXG2 3 xGA2 1

FINE GAIN

SELECT
☐ GAIN
☒ ATTN

0.6dB xXG3 6 xGA3 1

TOTAL GAIN -3.600 dB

AOUT GAIN 1.83 AxM 79

OK Apply Cancel

Inputs Valid

Figure 26. Si3217x VDAA Audio Gain Configuration Window

4.3.4 Si3217x VDAA Ring Detection and Validation Configuration

The Ring Detection and Validation configuration window is used to setup ring detection and validation presets. Table 14 lists the configurable parameters.

Input Parameter	Description
Enable Ring Validation	Select to enable ring validation
Min Ring Frequency	Select minimum ring frequency to be detected
Max Ring Frequency	Select maximum ring frequency to be detected
On Cadence	Enter minimum active timeout
Off Cadence	Enter minimum inactive timeout
Ring Validation to /RGDT Assert	Select validated ring to /RGDT bit assert delay
Ring Detect Polarity	Select polarity of /RGDT bit
Ring Detect Interrupt Mode	Select when in ring burst interrupts are generated
Ring Detect Threshold	Select ring detection threshold
/RGDT Bit Behavior	Select /RGDT Bit Cadencing

Table 14. Si3217x VDAA Ring Detection and Validation Configuration Parameters

Figure 27. Si3217x VDAA Ring Detection and Validation Configuration Window

4.3.5 Si3217x VDAA PCM Configuration

The VDAA PCM configuration window is used to setup the PCM bus presets. Table 15 lists the configurable parameters.

Input Parameter	Description
PCM Format	Enter PCM data companding format
PCM Clock Format	Enter PCM clock format
DTX Tristate	Enter DTX tristate edge (Si3050 only)

Table 15. Si3217x VDAA PCM Configuration Parameters

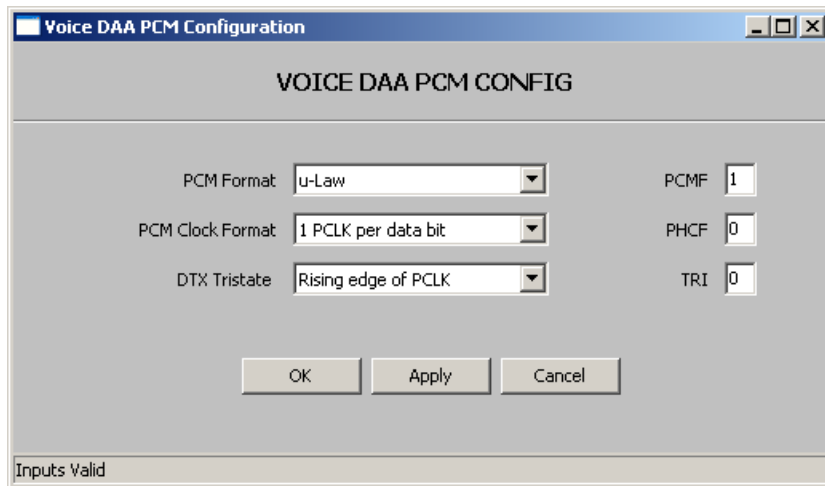


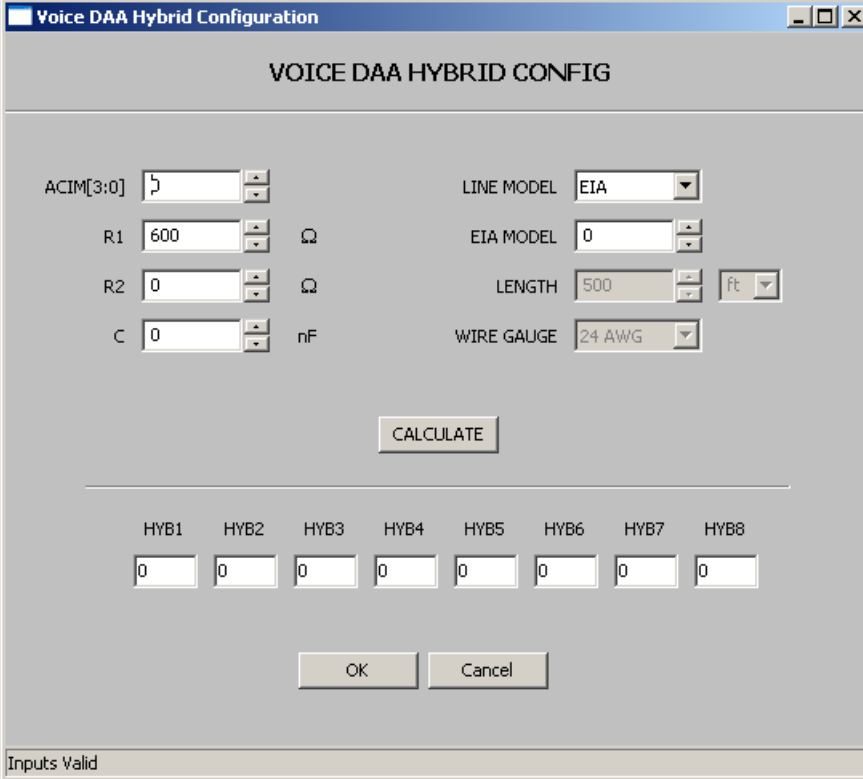
Figure 28. Si3217x VDAA PCM Configuration Window

4.3.6 Si3217x VDAA Hybrid Configuration

The VDAA Hybrid configuration window is used to setup the hybrid presets. This allows the user to have multiple sets of hybrid coefficients for a given AC impedance setting, which may be necessary for optimizing transhybrid balance over a variety of loop lengths. Table 16 lists the configurable parameters.

Input Parameter	Description
ACIM	Enter ACIM value
R1*	Enter R1 value of AC impedance model
R2*	Enter R2 value of AC impedance model
C	Enter C value of AC impedance model
Line Model	Select standard EIA line model or custom
Length	Enter loop length if using custom line model
Wire Gauge	Enter wire gauge if using custom line model
* AC impedance model assumed to be $R1 + (R2 C)$	

Table 16. Si3217x VDAA Hybrid Configuration Parameters



The image shows a software window titled "Voice DAA Hybrid Configuration". The window has a title bar with standard Windows controls (minimize, maximize, close). The main area is titled "VOICE DAA HYBRID CONFIG". It contains several input fields and dropdown menus for configuring a hybrid circuit. The inputs are arranged in two columns. The left column includes "ACIM[3:0]" (a dropdown menu), "R1" (a numeric input field with a unit selector set to Ω), "R2" (a numeric input field with a unit selector set to Ω), and "C" (a numeric input field with a unit selector set to nF). The right column includes "LINE MODEL" (a dropdown menu set to "EIA"), "EIA MODEL" (a numeric input field), "LENGTH" (a numeric input field with a unit selector set to "ft"), and "WIRE GAUGE" (a dropdown menu set to "24 AWG"). Below these inputs is a "CALCULATE" button. Underneath the button is a row of eight input fields labeled "HYB1" through "HYB8", each containing the value "0". At the bottom of the window are "OK" and "Cancel" buttons. A status bar at the very bottom indicates "Inputs Valid".

VOICE DAA HYBRID CONFIG

ACIM[3:0] LINE MODEL

R1 Ω EIA MODEL

R2 Ω LENGTH

C nF WIRE GAUGE

CALCULATE

HYB1 HYB2 HYB3 HYB4 HYB5 HYB6 HYB7 HYB8

OK Cancel

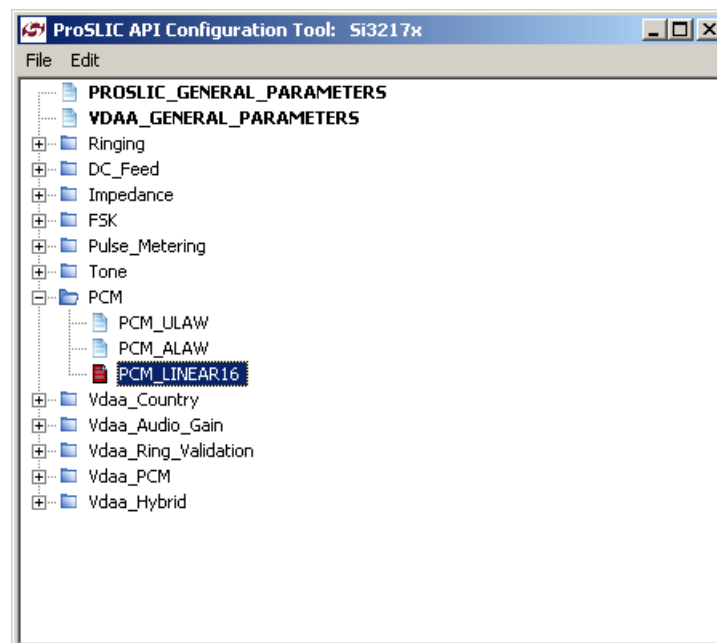
Inputs Valid

Figure 29. Si3217x VDAA Hybrid Configuration Window

4.4 Working with the ProSLIC API Configuration Tool Output

The ProSLIC API Configuration Tool produces C code used by the ProSLIC API to properly configure the ProSLIC device per the user's inputs. The "preset" is a data structure specific to each device and each device feature. For each feature, an array of data structures (or array of presets) is created. For each feature, the names of the presets are defined as enumerations with the value of the enumeration representing the index of the array of the corresponding data structure.

For example, if 3 PCM presets were created, as shown below,



the header file would enumerate the preset names as:

```
enum {
    PCM_ULAW,
    PCM_ALAW,
    PCM_LINEAR16,
    PCM_LAST_ENUM
};
```

and the C source file will define the preset as an array of data structures.

```
Si3217x_PCM_Cfg Si3217x_PCM_Presets[] ={
{
    0x01,          /* PCM_FMT - u-Law */
    0x00,          /* WIDEBAND - DISABLED (3.4kHz BW) */
    0x00,          /* PCM_TRI - PCLK RISING EDGE */
    0x00,          /* TX_EDGE - PCLK RISING EDGE */
    0x00 /* A-LAW - INVERT NONE */
}, /* PCM_ULAW */
{
    0x00,          /* PCM_FMT - A-Law */
    0x00,          /* WIDEBAND - DISABLED (3.4kHz BW) */
    0x00,          /* PCM_TRI - PCLK RISING EDGE */
    0x00,          /* TX_EDGE - PCLK RISING EDGE */
    0x00 /* A-LAW - INVERT NONE */
}, /* PCM_ALAW */
{
    0x03,          /* PCM_FMT - 16-bit Linear */
    0x00,          /* WIDEBAND - DISABLED (3.4kHz BW) */
    0x00,          /* PCM_TRI - PCLK RISING EDGE */
    0x00,          /* TX_EDGE - PCLK RISING EDGE */
    0x00 /* A-LAW - INVERT NONE */
} /* PCM_LINEAR16 */};
};
```

A specific preset is referenced by the array index enumeration. To load the preset, the user must call the ProSLIC_API function that corresponds to that feature. Table 17 lists the ProSLIC API function and the type of preset it loads.

For example, to select and load the PCM_ALAW preset,

```
ProSLIC_PCMSetup(pProslic, PCM_ALAW);
```

This selects the preset Si3217x_PCM_Presets[PCM_ALAW] and loads all relevant Register and RAM locations.

ProSLIC API Function	Corresponding Preset(s)
<i>ProSLIC_RingSetup()</i>	Si3228x_Ring_Presets, Si3218x_Ring_Presets, Si3219x_Ring_Presets, Si3226x_Ring_Presets, Si3217x_Ring_Presets
<i>ProSLIC_ToneGenSetup()</i>	Si3228x_Tone_Presets, Si3218x_Tone_Presets, Si3219x_Tone_Presets, Si3226x_Tone_Presets, Si3217x_Tone_Presets
<i>ProSLIC_FSKSetup()</i>	Si3228x_FSK_Presets, Si3218x_FSK_Presets, Si3219x_FSK_Presets, Si3226x_FSK_Presets, Si3217x_FSK_Presets
<i>ProSLIC_ZsynthSetup()</i>	Si3228x_Impedance_Presets, Si3218x_Impedance_Presets, Si3219x_Impedance_Presets, Si3226x_Impedance_Presets, Si3217x_Impedance_Presets
<i>ProSLIC_GciClSetup()</i>	Si3226x_CI_Presets, Si3217x_CI_Presets
<i>ProSLIC_DCFeedSetup()</i>	Si3228x_DCfeed_Presets, Si3218x_DCfeed_Presets, Si3219x_DCfeed_Presets, Si3226x_DCfeed_Presets, Si3217x_DCfeed_Presets
<i>ProSLIC_PulseMeterSetup()</i>	Si3228x_PulseMeter_Presets, Si3218x_PulseMeter_Presets, Si3219x_PulseMeter_Presets, Si3226x_PulseMeter_Presets Si3217x_PulseMeter_Presets
<i>ProSLIC_PCMSetup()</i>	Si3228x_PCM_Presets, Si3218x_PCM_Presets, Si3219x_PCM_Presets,

Proprietary Information - No dissemination or use without prior written permission from Silicon Labs.

	Si3226x_PCM_Presets, Si3217x_PCM_Presets
--	---

Table 17. ProSLIC API Preset Loading Functions

5.0 ProSLIC and Voice DAA Function Definitions

ProSLIC API data structure and function prototype descriptions are presented in a navigable HTML documentation set. The main page of the installed documentation may be found at the API installation start menu (*Start->All Programs->Silicon Laboratories->ProSLIC API X.Y.Z*)

6.0 SPI Driver Examples

The following section provides the user with pseudo-code design examples of how to implement a SPI port driver for the ProSLIC family of devices.

It is presumed that the user has already decided upon and verified their implementation of an SPI port. Common implementations include

- Hardware SPI Port (ie. Freescale compatible SPI with CPOL=1 and CPHA=1)
- Software SPI Port (ie. “bit-banged” GPIO)

All ProSLIC Register and RAM accesses can be reduced to a series of 8-bit SPI transfers, therefore once the user has implemented the basic read and write operation, development of the ProSLIC SPI driver is trivial.

6.1 Hardware SPI

When using a hardware SPI, the user should configure the port for CPOL=1 (SCLK/SPICLK is high when not clocking) and CPHA=1 (SDI/MOSI data change on SCLK/SPICLK falling and SDO/MISO is latched on SCLK /SPICLK rising). The user must

write a simple functions to implement 8-bit SPI port read/writes using the hardware SPI. If the host SPI interface provides buffered or burst-mode operation, the user must be certain that the /CS is deasserted for at least 220ns between each 8-bit transfer. Refer to the ProSLIC device's datasheet for timing specifications.

Customers may want to investigate using either 16-bit or 32-bit modes in addition to the 8-bit mode presented in this document. In some cases, these modes may improve total transfer data throughput. The Linux example SPI drivers include configuration options to try 8-bit, 16-bit, and 32-bit modes in both userspace and kernelspace.

6.2 Software SPI (GPIO)

When implementing a software SPI using general purpose I/O pins (GPIOs), or what is commonly referred to as “bit-banging”, the user must write robust 8-bit SPI read and write functions.

Figure 30 shows a diagram of a simple 8-bit SPI transfer. In this example implementation, the state changes are on 1/2 SCLK period boundaries, that is, the GPIO outputs will be updated once for every 1/2 SCLK period. The SDO data may be latched only during specific cycles when the data is expected to be stable. In this case, it would be during those 1/2 SCLK cycles in which SCLK is high (ie. loop 3, 5, 7, 9, 11, 13, 15, and 17).

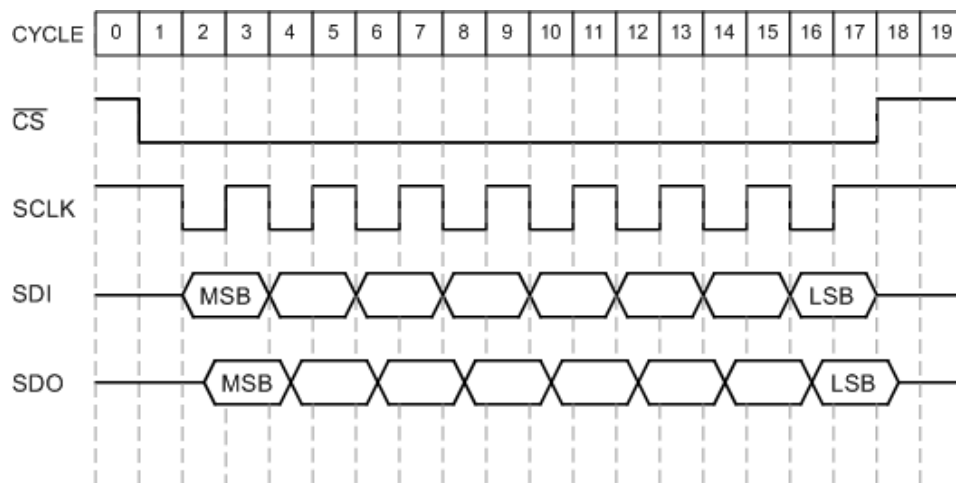


Figure 30. "Bit-Banged" SPI Transfer

Below are simple code examples that illustrate how to implement SPI read and write functions:

```

/*
** GPIO Configuration for example driver:
**
** Bit 0          /CS
** Bit 1          SCLK
** Bit 2          SDI
** Bit 3          SDO
** Bits 4-7 Don't Care
**
*/

/* GPIO Bitmasks */
#define CSB_PIO    0x01
#define SCLK_PIO   0x02
#define SDI_PIO    0x04
#define SDO_PIO    0x08

/* SPI delay parameter */
#define HALF_SCLK_PER 256

/*
** SpiReadByte()
**
** This function performs an 8-bit SPI read by controlling all
** SPI signals with GPIO
**
*/
uInt8 SpiReadByte(void) {
    uInt8 gpioWrData[20];
    uInt8 gpioRdData;
    uInt8 spiRdData = 0;
    uInt8 dummyData = 0
    int i;
    int dataCnt = 7;
    int dummyCnt = 100;

```

```

/*
** Precompute outputs for all 20 cycles
** to minimize SCLK jitter. Inputs will be
** latched during readback.
*/
gpioWrData[0] = (CSB_PIO | SCLK_PIO);
gpioWrData[1] = (SCLK_PIO);
for(i=2;i<18;i=i+2) {
    gpioWrData[i] = 0x00;
    gpioWrData[i+1] = (SCLK_PIO);
}
gpioWrData[18] = (CSB_PIO | SCLK_PIO);
gpioWrData[19] = (CSB_PIO | SCLK_PIO);

/*
** Send Data - try to keep execution of each 1/2 SCLK period
** as close as possible to minimize variation in SCLK frequency
** by reading and processing on cycles in which there is no valid
** data to read.
**
** The opposite could also be done to prevent a periodic SCLK
** frequency to minimize EMI. User's choice.
*/

spiRdData = 0;

for(i=0;i<2;i++) { /* cycles 0-1 */
    gpioRdData = readGPIO(); /* User's GPIO read func */
    writeGPIO(gpioWrData[i]); /* User's GPIO write func */
    dummyRdData |= (gpioRdData & SDO_PIO) ? 1:0; /* NOP */
    dummyCnt--; /* NOP */
    spiDelay(HALF_SCLK_PER); /* User's delay func */
}

for(i=2;i<18;i=i+2) { /* cycles 2-17 */

    /* SCLK Low - nop */
    gpioRdData = readGPIO(); /* User's GPIO read func */
    writeGPIO(gpioWrData[i]); /* User's GPIO write func */

```

```

        dummyRdData |= (gpioRdData & SDO_PIO) ? 1:0; /* NOP */
        dummyCnt--;                                /* NOP */
        spiDelay(HALF_SCLK_PER); /* User's delay func */

        /* SCLK High - Latch Data */
        gpioRdData = readGPIO(); /* Valid SDO bit expected */
        writeGPIO(gpioWrData[i+1]);
        spiRdData |= (gpioRdData & SDO_PIO) ? (1<<dataCnt) : 0;
        dataCnt--;
        spiDelay(HALF_SCLK_PER); /* User's delay func */
    }

    for(i=18;i<20;i++) { /* cycles 18-19 */
        gpioRdData = readGPIO(); /* User's GPIO read func */
        writeGPIO(gpioWrData[i]); /* User's GPIO write func */
        dummyRdData |= (gpioRdData & SDO_PIO) ? 1:0; /* NOP */
        dummyCnt--;                                /* NOP */
        spiDelay(HALF_SCLK_PER); /* User's delay func */
    }
    return spiRdData;
}

/*
** SpiWriteByte()
**
** This function performs an 8-bit SPI write by controlling all
** SPI signals with GPIO
*/
void SpiWriteByte(uInt8 wrData) {
    uInt8 gpioWrData[20];
    int i;

    /*
    ** Precompute outputs for all 20 cycles
    */
    gpioWrData[0] = (CSB_PIO | SCLK_PIO);
    gpioWrData[1] = (SCLK_PIO);
    gpioWrData[2] = (wrData&0x80) ? SDI_PIO : 0x00;
    gpioWrData[3] = (SCLK_PIO) | ((wrData&0x80) ? SDI_PIO : 0x00);
    gpioWrData[4] = (wrData&0x40) ? SDI_PIO : 0x00;

```

```

gpioWrData[5]  = (SCLK_PIO) | ((wrData&0x40) ? SDI_PIO : 0x00);
gpioWrData[6]  = (wrData&0x20) ? SDI_PIO : 0x00;
gpioWrData[7]  = (SCLK_PIO) | ((wrData&0x20) ? SDI_PIO : 0x00);
gpioWrData[8]  = (wrData&0x10) ? SDI_PIO : 0x00;
gpioWrData[9]  = (SCLK_PIO) | ((wrData&0x10) ? SDI_PIO : 0x00);
gpioWrData[10] = (wrData&0x08) ? SDI_PIO : 0x00;
gpioWrData[11] = (SCLK_PIO) | ((wrData&0x08) ? SDI_PIO : 0x00);
gpioWrData[12] = (wrData&0x04) ? SDI_PIO : 0x00;
gpioWrData[13] = (SCLK_PIO) | ((wrData&0x04) ? SDI_PIO : 0x00);
gpioWrData[14] = (wrData&0x02) ? SDI_PIO : 0x00;
gpioWrData[15] = (SCLK_PIO) | ((wrData&0x02) ? SDI_PIO : 0x00);
gpioWrData[16] = (wrData&0x01) ? SDI_PIO : 0x00;
gpioWrData[17] = (SCLK_PIO) | ((wrData&0x01) ? SDI_PIO : 0x00);
gpioWrData[18] = (CSB_PIO | SCLK_PIO);
gpioWrData[19] = (CSB_PIO | SCLK_PIO);

/*
** Send Buffer
*/

for(i=0;i<20;i++) { /* cycles 0-19 */
    writeGPIO(gpioWrData[i]); /* User's GPIO write func */
    spiDelay(HALF_SCLK_PER); /* User's delay func */
}
}

```

Using the SpiReadByte() and SpiWriteByte() functions that have been created the user may now build an SPI driver for their application code.

6.3 Register Read Example

A register read operation on the Si3050, Si3217x, Si3218x, Si3219x, Si3226x, & Si3228x devices is a simple 3-byte transfer:

- Byte 1: Write Control Word
- Byte 2: Write Address
- Byte 3: Read Data

The following example decodes the channel ID, generates the proper control word, and performs a register read using the SpiWriteByte() and SpiReadByte() functions supplied by the user.

```

/*
** chanNumToCID()
**
** Convert channel # to CID word - ProSLIC
**
*/
uInt8 chanNumToCID(uInt8 channelNum) {
    uInt8 cid;

    cid = (channelNum<<4) & 0x10;
    cid |= (channelNum<<2) & 0x08;
    cid |= (channelNum>>2) & 0x02;
    cid |= (channelNum>>4) & 0x01;
    cid |= channelNum & 0x04;

    return cid;
}

/*
** ReadRegister()
**
** Convert channel # to CID word
**
*/
uInt8 ReadRegister(uInt8 channel, uInt8 address) {
    uInt8 controlWord;

    if(channel == 0xFF) { /* Broadcast */
        controlWord = 0x80;
    }
    else {
        controlWord = chanNumToCID(channel); /* Encode CID */
    }

    controlWord |= 0x60; /* Set R/W and REG/RAM */

```

```

    SpiWriteByte(controlWord);
    SpiWriteByte(address);
    return SpiReadByte();
}

```

6.4 Register Write Example

A register write operation on the Si3050, Si3217x, Si3218x, Si3219x, Si3226x, & Si3228x is also a simple 3-byte transfer:

- Byte 1: Write Control Word
- Byte 2: Write Address
- Byte 3: Write Data

The following example decodes the channel ID, generates the proper control word, and performs a register write using the SpiWriteByte() function supplied by the user.

```

/*
** WriteRegister()
**
** Write to ProSLIC Register Space
**
*/
void WriteRegister(uInt8 channel, uInt8 address, uInt8 data) {
    uInt8 controlWord;

    if(channel == 0xFF) { /* Broadcast */
        controlWord = 0x80;
    }
    else {
        controlWord = chanNumToCID(channel); /* Encode CID */
    }

    controlWord |= 0x20; /* Set REG/RAM */
    controlWord &= 0xBF; /* Clear R/W */

    SpiWriteByte(controlWord);
    SpiWriteByte(address);
}

```

```

        SpiWriteByte(data);
    }

```

6.5 RAM Read Example

A RAM read operation on the Si3217x, Si3218x, Si3219x, Si3226x, & Si3228x is a series of register writes and reads. To read from RAM address *ramAddr[10:0]*, the following byte stream is needed:

- Bytes 1-3: Write register RAM_ADR_HI[7:5] with ramAddr[10:8]
- Bytes 4-6: Write register RAM_ADR_LO[7:0] with ramAddr[7:0]
- Bytes 7-9: Read register RAM_STAT to ensure valid data
- Bytes 10-12: Read register RAM_DATA_B0[7:0]
- Bytes 13-15: Read register RAM_DATA_B1[7:0]
- Bytes 16-18: Read register RAM_DATA_B2[7:0]
- Bytes 19-21: Read register RAM_DATA_B3[7:0]

The following example correctly partitions the address and data and performs a RAM write using the user's new WriteRegister() function.

```

/*
** ReadRAM()
**
** Read from ProSLIC RAM space
**
*/
uInt32 ReadRAM(uInt8 channel, uInt16 address) {
    uInt8 dataByte;
    uInt8 addrByte;
    uInt32 dataWord = 0L;
    uInt32 userTimeout = USER_TIMEOUT_VAL; /* user defined timeout counter */

    /* Wait for RAM to finish */
    while((ReadRegister(channel, RAMSTAT) & 0x01) && (userTimeout > 0)) {
        userTimeout--;
    }
}

```



```

    }

    /* RAM_ADR_HI[7:5] = ramAddr[10:8] */
    addrByte = (uInt8)((address >> 3)&0x00E0);
    WriteRegister(channel, RAM_ADR_HI, addrByte);

    /* RAM_ADR_LO[7:0] = ramAddr[7:0] */
    addrByte = (uInt8)((address &0x00FF);
    WriteRegister(channel, RAM_ADR_LO, addrByte);

    /* Wait for RAM to finish */
    while((ReadRegister(channel, RAMSTAT)&0x01)&&(userTimeout > 0)) {
        userTimeout--;
    }

    /* ramData[4:0] = RAM_DATA_B0[7:3] */
    dataByte = ReadRegister(channel, RAM_DATA_B0);
    dataWord |= ((dataByte >> 3)&0x0000001FL);

    /* ramData[12:5] = RAM_DATA_B1[7:0] */
    dataByte = ReadRegister(channel, RAM_DATA_B1);
    dataWord |= ((dataByte << 5)&0x000001FE0L);

    /* ramData[20:13] = RAM_DATA_B2[7:0] */
    dataByte = ReadRegister(channel, RAM_DATA_B2);
    dataWord |= ((dataByte << 13) & 0x0001FE000L);

    /* ramData[28:21] = RAM_DATA_B3[7:0] */
    dataByte = ReadRegister(channel, RAM_DATA_B3);
    dataWord |= ((dataByte << 21) & 0x1FE00000L);

    return dataWord;
}

```

6.6 RAM Write Example

A RAM write operation on the Si3217x, Si3218x, Si3219x, Si3226x, & Si3228x is a series of *register* writes and reads. To write the data *ramData*[28:0] to RAM address *ramAddr*[10:0], the following byte stream is required:

- Bytes 1-3: Write register RAM_ADR_HI[7:5] with ramAddr[10:8]
- Bytes 4-6: Write register RAM_DATA_B0[7:3] with ramData[4:0]
- Bytes 7-9: Write register RAM_DATA_B1[7:0] with ramData[12:5]
- Bytes 10-12: Write register RAM_DATA_B2[7:0] with ramData[20:13]
- Bytes 13-15: Write register RAM_DATA_B3[7:0] with ramData[28:21]
- Bytes 16-18: Write register RAM_ADR_LO[7:0] with ramAddr[7:0]

The following example correctly partitions the address and data and performs a RAM write using the user's new WriteRegister() function.

```

/*
** WriteRAM()
**
** Write to ProSLIC RAM space
**
*/
void WriteRAM(uInt8 channel, uInt16 address, uInt32 data) {
    uInt8 dataByte;
    uInt8 addrByte;
    uInt32 userTimeout = USER_TIMEOUT_VAL;    /* User defined timeout counter */

    /* Wait for RAM to finish */
    while((ReadRegister(channel, RAMSTAT) & 0x01) && (userTimeout > 0)) {
        userTimeout--;
    }

    /* RAM_ADR_HI[7:5] = ramAddr[10:8] */
    addrByte = (uInt8)((address >> 3) & 0x00E0);
    WriteRegister(channel, RAM_ADR_HI, addrByte);

    /* RAM_DATA_B0[7:3] = ramData[4:0] */
    dataByte = (uInt8)((data << 3) & 0x000000F8L);
    WriteRegister(channel, RAM_DATA_B0, dataByte);
}

```

```

    /* RAM_DATA_B1[7:0] = ramData[12:5] */
    dataByte = (uInt8)((data >> 5)&0x000000FFL);
    WriteRegister(channel, RAM_DATA_B1, dataByte);

    /* RAM_DATA_B2[7:0] = ramData[20:13] */
    dataByte = (uInt8)((data >> 13)&0x000000FFL);
    WriteRegister(channel, RAM_DATA_B2, dataByte);

    /* RAM_DATA_B3[7:0] = ramData[28:21] */
    dataByte = (uInt8)((data >> 21)&0x000000FFL);
    WriteRegister(channel, RAM_DATA_B3, dataByte);

    /* RAM_ADR_LO[7:0] = ramAddr[7:0] */
    addrByte = (uInt8)((address &0x00FF);
    WriteRegister(channel, RAM_ADR_LO, addrByte);
}

```

7.0 ProSLIC and VDAA Code Examples

The following section provides example implementations of the ProSLIC and VDAA API. Two example initializations are provided for two different system configurations, along with additional examples of commonly used control functions.

7.1 Multi-Channel ProSLIC Initialization

The following code is an example initialization of a ProSLIC system with multiple channels, in this case, two Si3217x devices (without FXO support).

```

/*
**
** Multi-port ProSLIC Initialization
**
*/

#define NUMBER_OF_DEVICES 2
#define CHAN_PER_DEVICE 1
#define NUMBER_OF_CHAN (NUMBER_OF_DEVICES*CHAN_PER_DEVICE)

```

Proprietary Information - No dissemination or use without prior written permission from Silicon Labs.

```

#define PROSLIC_DEVICE_TYPE SI3217X_TYPE

int CustomerProSLIC_HWInit()
{
    ctrl_S spiGciObj;           /* User's control interface object */
    systemTimer_S timerObj;     /* User's timer object */
    chanState ports[NUMBER_OF_CHAN] /* User's channel object, which has
                                   ** a member defined as
                                   ** proslicChanType_ptr ProObj;
                                   */

    /* Define ProSLIC control interface object */
    controlInterfaceType *ProHWIntf;

    /* Define array of ProSLIC device objects */
    ProslicDeviceType *ProSLICDevices[NUMBER_OF_PROSLIC];

    /* Define array of ProSLIC channel object pointers */
    proslicChanType_ptr arrayOfProslicChans[NUMBER_OF_CHAN];

    /*
    ** Step 1: (optional)
    ** Initialize user's control interface and timer objects - this
    ** may already be done, if not, do it here
    */

    /*
    ** Step 2: (required)
    ** Create ProSLIC Control Interface Object
    */
    SiVoice_createControlInterface(&ProHWIntf);

    /*
    ** Step 3: (required)
    ** Create ProSLIC Device Objects
    */
    SiVoice_createDevices(&(ProSLICDevices), NUMBER_OF_PROSLIC);

    /*
    ** Step 4: (required)
    ** Create and initialize ProSLIC channel objects
    ** Also initialize array pointers to user's proslic channel object
    ** members to simplify initialization process.
    */
    SiVoice_createChannel(&(ports.ProObj), NUMBER_OF_CHAN);

```

```

for(i=0;i<NUMBER_OF_CHAN;i++)
{
    SiVoice_SWInitChan(ports[i].ProObj,i,PROSLIC_DEVICE_TYPE,
                      ProSLICDevices[i/CHAN_PER_DEVICE],ProHWIntf);
    arrayOfProslicChans[i] = ports[i].ProObj;
    ProSLIC_setSWDebugMode(ports[i].ProObj,TRUE); /* optional */
}

/*
** Step 5: (required)
** Establish linkage between host objects/functions and
** ProSLIC API
**/
SiVoice_setControlInterfaceCtrlObj (ProHWIntf, &spiGciObj);
SiVoice _setControlInterfaceReset (ProHWIntf, ctrl_ResetWrapper);
SiVoice _setControlInterfaceWriteRegister (ProHWIntf, ctrl_WriteRegisterWrapper);
SiVoice _setControlInterfaceReadRegister (ProHWIntf, ctrl_ReadRegisterWrapper);
SiVoice setControlInterfaceWriteRAM (ProHWIntf, ctrl_WriteRAMWrapper);
SiVoice _setControlInterfaceReadRAM (ProHWIntf, ctrl_ReadRAMWrapper);
SiVoice _setControlInterfaceTimerObj (ProHWIntf, &timerObj);
SiVoice _setControlInterfaceDelay (ProHWIntf, time_DelayWrapper);
SiVoice _setControlInterfaceTimeElapsed (ProHWIntf, time_TimeElapsedWrapper);
SiVoice _setControlInterfaceGetTime (ProHWIntf, time_GetTimeWrapper);
SiVoice _setControlInterfaceSemaphore (ProHWIntf, NULL);

/*
** Step 6: (system dependent)
** Assert hardware Reset - ensure VDD, PCLK, and FSYNC are present and stable
** before releasing reset
**/
ProSLIC_Reset(ports[0].ProObj);

/*
** Step 7: (required)
** Initialize device (loading of general parameters, calibrations,
** dc-dc powerup, etc.)
**/
ProSLIC_Init(arrayOfProslicChans,NUMBER_OF_CHAN);

/*
** Step 8: (design dependent)
** Execute longitudinal balance calibration
** or reload coefficients from factory LB cal
**
** Note: all batteries should be up and stable prior to

```

```

** executing the lb cal
*/
ProSLIC_LBCal(arrayOfProslicChans,NUMBER_OF_CHAN);

/*
** Step 9: (design dependent)
** Load custom configuration presets(generated using
** ProSLIC API Config Tool)
**/
for(i=0;i<NUMBER_OF_CHAN;i++)
{
    ProSLIC_DCFeedSetup(ports[i].ProObj,DCFEED_48V_20MA);
    ProSLIC_RingSetup(ports[i].ProObj,RING_F20_45VRMS_0VDC);
    ProSLIC_PCMSetup(ports[i].ProObj,PCM_DEFAULT_CONFIG);
    ProSLIC_ZsynthSetup(ports[i].ProObj,ZSYN_600_0_0);
    ProSLIC_ToneGenSetup(ports[i].ProObj,TONEGEN_FCC_DIALTONE);
}

/*
** END OF TYPICAL INITIALIZATION
**/

```

7.2 Multi-Channel Mixed ProSLIC/VDAA Initialization

The following code is an example initialization of a ProSLIC/VDAA system with multiple channels, in this case, a single Si3217x device with FXO support. To support the unified ProSLIC and VDAA architectures, the API configuration structures reference the *SiVoice* API functions in which the ProSLIC and VDAA functions are derived from.

```

/*
**
** Multi-port Voice Initialization
**
*/

#define NUMBER_OF_DEVICES 1
#define CHAN_PER_DEVICE 2
#define NUMBER_OF_CHAN (NUMBER_OF_DEVICES*CHAN_PER_DEVICE)
#define VOICE_DEVICE_TYPE SI3217X_TYPE

int CustomerVoice_HWInit()
{
    ctrl_S    spiGciObj;                /* User's control interface object */

```

```

systemTimer_S timerObj;           /* User's timer object */
chanState ports[NUMBER_OF_CHAN]   /* User's channel object, which has
                                   ** a member defined as
                                   ** SiVoiceChanType_ptr VoiceObj;
                                   */

/* Define Voice control interface object */
SiVoiceControlInterfaceType *VoiceHWIntf;

/* Define array of Voice device objects */
SiVoiceDeviceType *VoiceDevices[NUMBER_OF_DEVICES];

/* Define array of ProSLIC channel object pointers */
SiVoiceChanType_ptr arrayOfVoiceChans[NUMBER_OF_CHAN];

/*
** Step 1: (optional)
** Initialize user's control interface and timer objects - this
** may already be done, if not, do it here
*/

/*
** Step 2: (required)
** Create Voice Control Interface Object
*/
SiVoice_createControlInterface(&VoiceHWIntf);

/*
** Step 3: (required)
** Create Voice Device Objects
*/
SiVoice_createDevices(&(VoiceDevices[i]), NUMBER_OF_DEVICES);

/*
** Step 4: (required)
** Create and initialize Voice channel objects
** Also initialize array pointers to user's proslic channel object
** members to simplify initialization process.
*/
SiVoice_createChannels(&(ports[i].VoiceObj), NUMBER_OF_CHAN);
for(i=0;i<NUMBER_OF_CHAN;i++)
{
    SiVoice_SWInitChan(ports[i].VoiceObj,i,VOICE_DEVICE_TYPE,
                      VoiceDevices[i/CHAN_PER_DEVICE],VoiceHWIntf);
}

```

```

    arrayOfVoiceChans[i] = ports[i].VoiceObj;
    SiVoice_setSWDebugMode(ports[i].VoiceObj, TRUE); /* optional */
}

/*
** Step 5: (required)
** Establish linkage between host objects/functions and
** API
**/
SiVoice_setControlInterfaceCtrlObj (VoiceHWIntf, &spiGciObj);
SiVoice_setControlInterfaceReset (VoiceHWIntf, ctrl_ResetWrapper);
SiVoice_setControlInterfaceWriteRegister (VoiceHWIntf, ctrl_WriteRegisterWrapper);
SiVoice_setControlInterfaceReadRegister (VoiceHWIntf, ctrl_ReadRegisterWrapper);
SiVoice_setControlInterfaceWriteRAM (VoiceHWIntf, ctrl_WriteRAMWrapper);
SiVoice_setControlInterfaceReadRAM (VoiceHWIntf, ctrl_ReadRAMWrapper);
SiVoice_setControlInterfaceTimerObj (VoiceHWIntf, &timerObj);
SiVoice_setControlInterfaceDelay (VoiceHWIntf, time_DelayWrapper);
SiVoice_setControlInterfaceTimeElapsed (VoiceHWIntf, time_TimeElapsedWrapper);
SiVoice_setControlInterfaceGetTime (VoiceHWIntf, time_GetTimeWrapper);
SiVoice_setControlInterfaceSemaphore (VoiceHWIntf, NULL);

/*
** Step 6: (system dependent)
** Assert hardware Reset - ensure VDD, PCLK, and FSNC are present and stable
** before releasing reset
**/
SiVoice_Reset(ports[0].VoiceObj);

/*
** Step 7: (required)
** Initialize channels (loading of general parameters, calibrations,
** dc-dc powerup, etc.)
** Note that VDAA channels are ignored by ProSLIC_Init and ProSLIC
** channels are ignored by Vdaa_Init.
**/
ProSLIC_Init(arrayOfVoiceChans, NUMBER_OF_CHAN);
Vdaa_Init(arrayOfVoiceChans, NUMBER_OF_CHAN);

/*
** Step 8: (design dependent)
** Execute longitudinal balance calibration
** or reload coefficients from factory LB cal
**
** Note: all batteries should be up and stable prior to

```



```

** executing the lb cal
*/
ProSLIC_LBCal(arrayOfVoiceChans,NUMBER_OF_CHAN);

/*
** Step 9: (design dependent)
** Load custom configuration presets(generated using
** ProSLIC API Config Tool)
**
for(i=0;i<NUMBER_OF_CHAN;i++)
{
    ProSLIC_DCFeedSetup(ports[i].VoiceObj,DCFEED_48V_20MA);
    ProSLIC_RingSetup(ports[i].VoiceObj,RING_F20_45VRMS_0VDC);
    ProSLIC_PCMSetup(ports[i].VoiceObj,PCM_DEFAULT_CONFIG);
    ProSLIC_ZsynthSetup(ports[i].VoiceObj,ZSYN_600_0_0);
    ProSLIC_ToneGenSetup(ports[i].VoiceObj,TONEGEN_FCC_DIALTONE);
    Vdaa_CountrySetup(ports[i].VoiceObj,DEFAULT_COUNTRY);
    Vdaa_HybridSetup(ports[i].VoiceObj,HYB_24AWG_2000FT);
    Vdaa_PCMSetup(ports[i].VoiceObj,PCM_ULAW);
}

/*
** END OF TYPICAL INITIALIZATION
*/

```

7.3 Multi-Channel / Multi-Device Mixed ProSLIC/VDAA Initialization

The following code is an example initialization of a ProSLIC/VDAA system with multiple channels, in this case, three Si3226x devices and two Si32178 devices (with FXO) on the same daisy chain. To support the unified ProSLIC and VDAA architectures, the API configuration structures reference the *SiVoice* API functions in which the ProSLIC and VDAA functions are derived from.

```

/*
**
** Multi-port/Multi-device Voice Initialization
**
*/

/*
** Macros describing the daisy chain
**
#define NUMBER_OF_DEVICES 5          /* 3 Si3226x + 2 Si32178 */

```

```

#define CHAN_PER_DEVICE 2          /* Since both Si3226x and Si32178 have 2 voice chans */
#define NUMBER_OF_SI3226X_CHAN 6   /* 3 x Si3226x */
#define NUMBER_OF_SI3217X_CHAN 4   /* 2 x Si32178 */
#define NUMBER_OF_CHAN (NUMBER_OF_SI3226X_CHAN + NUMBER_OF_SI3217X_CHAN)

int CustomerVoice_HWInit()
{
    ctrl_S    spiGciObj;           /* User's control interface object */
    systemTimer_S timerObj;        /* User's timer object */
    chanState ports[NUMBER_OF_CHAN] /* User's channel object, which has
                                   ** a member defined as
                                   ** SiVoiceChanType_ptr VoiceObj;
                                   */

    /* Define Voice control interface object */
    SiVoiceControlInterfaceType *VoiceHWIntf;

    /* Define array of Voice device objects */
    SiVoiceDeviceType *VoiceDevices[NUMBER_OF_DEVICES];

    /* Define array of ProSLIC channel object pointers */
    SiVoiceChanType_ptr arrayOfVoiceChans[NUMBER_OF_CHAN];

    /*
    ** Step 1: (optional)
    ** Initialize user's control interface and timer objects - this
    ** may already be done, if not, do it here
    */

    /*
    ** Step 2: (required)
    ** Create Voice Control Interface Object
    */
    SiVoice_createControlInterface(&VoiceHWIntf);

    /*
    ** Step 3: (required)
    ** Create Voice Device Objects
    */
    SiVoice_createDevices(VoiceDevices, NUMBER_OF_DEVICES);

    /*
    ** Step 4: (required)

```

```

** Create and initialize Voice channel objects
** Also initialize array pointers to user's proslic channel object
** members to simplify initialization process.
**
** If multiple ProSLIC device types are on the same daisy chain
** then each device type must be initialized separately
*/

    SiVoice_createChannels((ports.VoiceObj),NUMBER_OF_CHAN);

/* Create Si3226X Channels (ch 0-5) */
for(i=0;i<NUMBER_OF_SI3226X_CHAN;i++)
{
    SiVoice_SWInitChan(ports[i].VoiceObj,i,SI3226X_TYPE,
                        VoiceDevices[i/CHAN_PER_DEVICE],VoiceHWIntf);
    arrayOfVoiceChans[i] = ports[i].VoiceObj;
}

/* Create Si3217x Channels (ch 6-9) */
for(i=NUMBER_OF_SI3226X_CHAN;i<NUMBER_OF_CHAN;i++)
{
    SiVoice_SWInitChan(ports[i].VoiceObj,i,SI3217X_TYPE,
                        VoiceDevices[i/CHAN_PER_DEVICE],VoiceHWIntf);
    arrayOfVoiceChans[i] = ports[i].VoiceObj;
}

/*
** Step 5: (required)
** Establish linkage between host objects/functions and
** API
*/
SiVoice_setControlInterfaceCtrlObj (VoiceHWIntf, &spiGciObj);
SiVoice_setControlInterfaceReset (VoiceHWIntf, ctrl_ResetWrapper);
SiVoice_setControlInterfaceWriteRegister (VoiceHWIntf, ctrl_WriteRegisterWrapper);
SiVoice_setControlInterfaceReadRegister (VoiceHWIntf, ctrl_ReadRegisterWrapper);
SiVoice_setControlInterfaceWriteRAM (VoiceHWIntf, ctrl_WriteRAMWrapper);
SiVoice_setControlInterfaceReadRAM (VoiceHWIntf, ctrl_ReadRAMWrapper);
SiVoice_setControlInterfaceTimerObj (VoiceHWIntf, &timerObj);
SiVoice_setControlInterfaceDelay (VoiceHWIntf, time_DelayWrapper);
SiVoice_setControlInterfaceTimeElapsed (VoiceHWIntf, time_TimeElapsedWrapper);
SiVoice_setControlInterfaceGetTime (VoiceHWIntf, time_GetTimeWrapper);
SiVoice_setControlInterfaceSemaphore (VoiceHWIntf, NULL);

/*
** Step 6: (system dependent)
** Assert hardware Reset - ensure VDD, PCLK, and FSYNC are present and stable

```

```
** before releasing reset
*/
SiVoice_Reset(ports[0].VoiceObj);

/*
** Step 7: (required)
** Initialize channels (loading of general parameters, calibrations,
**   dc-dc powerup, etc.)
** Note that VDAA channels are ignored by ProSLIC_Init and ProSLIC
** channels are ignored by Vdaa_Init.
**/

/* Initialize Si3226X Channels */
if(ProSLIC_Init(&(arrayOfVoiceChans[0]),NUMBER_OF_SI3226X_CHAN))
{
    LOGPRINT("ERROR: Si3226 FXS Initialization Failed!\n");
}

/* Initialize Si3217X FXS Channels */
if(ProSLIC_Init(&(arrayOfVoiceChans[NUMBER_OF_SI3226_CHAN]),NUMBER_OF_SI3217X_CHAN))
{
    LOGPRINT("ERROR: Si3217x FXS Initialization Failed!\n");
}

/* Initialize Si3217X FXO Channels */
if(Vdaa_Init(&(arrayOfVoiceChans[NUMBER_OF_SI3226_CHAN]),NUMBER_OF_SI3217X_CHAN))
{
    LOGPRINT("ERROR: Si3217x FXO Initialization Failed!\n");
}

/*
** Step 8: (design dependent)
** Execute longitudinal balance calibration
** or reload coefficients from factory LB cal
**
** Note: all batteries should be up and stable prior to
** executing the lb cal
**/

/* Calibrate Si3226 Channels */
ProSLIC_LBCal(&(arrayOfVoiceChans[0]),NUMBER_OF_SI3226_CHAN);

/* Calibrate Si3217x Channels */
ProSLIC_LBCal(&(arrayOfVoiceChans[NUMBER_OF_SI3226_CHAN]),NUMBER_OF_CHAN);
```

```

/*
** Step 9: (design dependent)
** Load custom configuration presets (generated using
** ProSLIC API Config Tool)
*/
for (i=0; i<NUMBER_OF_CHAN; i++)
{
    ProSLIC_DCFeedSetup (ports[i].VoiceObj, DCFEED_48V_20MA);
    ProSLIC_RingSetup (ports[i].VoiceObj, RING_F20_45VRMS_0VDC);
    ProSLIC_PCMSetup (ports[i].VoiceObj, PCM_DEFAULT_CONFIG);
    ProSLIC_ZsynthSetup (ports[i].VoiceObj, ZSYN_600_0_0);
    ProSLIC_ToneGenSetup (ports[i].VoiceObj, TONEGEN_FCC_DIALTONE);
    Vdaa_CountrySetup (ports[i].VoiceObj, DEFAULT_COUNTRY);
    Vdaa_HybridSetup (ports[i].VoiceObj, HYB_24AWG_2000FT);
    Vdaa_PCMSetup (ports[i].VoiceObj, PCM_ULAW);
}

/*
** END OF TYPICAL INITIALIZATION
*/

```

7.4 Control Function Examples

```

/*
** EXAMPLE: Change ALL ProSLIC ports to FWD_ACTIVE linefeed state
** NOTE: if the daisychain contains ONLY ProSLIC's, one can use the broadcast
** version.
*/
for (i=0; i<NUMBER_OF_CHAN; i++)
{
    ProSLIC_SetLinefeedStatus (ports[i].VoiceObj, LF_FWD_ACTIVE);
}

/*
** EXAMPLE: Setup and start busy signal on port 1 using tone generators
** TONEGEN_FCC_BUSY is preset created using ProSLIC API Config Tool
*/
ProSLIC_ToneGenSetup (ports[1].VoiceObj, TONEGEN_FCC_BUSY);
ProSLIC_ToneGenStart (ports[1].VoiceObj, PROSLIC_TG_TIMER_ENABLED);

/*
** EXAMPLE: Shutdown channel 3
*/
ProSLIC_ShutdownChannel (ports[3].VoiceObj);

```

```

/*
** EXAMPLE: Read channel 1 VDAA hookswitch state.  If onhook, set to offhook
*/
uint8 hsState;

hsState = Vdaa_GetHookStatus(ports[1].VoiceObj);
if(hsState == VDAA_ONHOOK) {
    Vdaa_SetHookStatus(ports[1].VoiceObj, VDAA_OFFHOOK);
}

```

7.5 Provisioning Function Examples

```

/*
** EXAMPLE: Change TX and RX gain on channel 2 using ZSYN_270_750_150
** impedance preset (created by ProSLIC API Config Tool)
*/

ProSLIC_dbgSetTXGain(ports[2].VoiceObj,-4,ZSYN_270_750_150,0);
ProSLIC_dbgSetRXGain(ports[2].VoiceObj,-2,ZSYN_270_750_150,1);
ProSLIC_TXAudioGainSetup(ports[2].VoiceObj,0);
ProSLIC_RXAudioGainSetup(ports[2].VoiceObj,1);

/*
** Alternate EXAMPLE: Change TX and RX gain on channel 2 using ZSYN_270_750_150
** impedance preset (created by ProSLIC API Config Tool)
*/

ProSLIC_AudioGainSetup(ports[2].VoiceObj,-2,-4,ZSYN_270_750_150);

/*
** EXAMPLE: Change ringing to 62Vpk, 25Hz, Sinusoidal, no dc offset
*/

ProSLIC_dbgRingCfg ringCfg;

ringCfg.ringtype = ProSLIC_RING_SINE;
ringCfg.amp = 62;
ringCfg.freq = 25;
ringCfg.offset = 0;

ProSLIC_dbgSetRinging(ports[2].VoiceObj, &ringCfg, DEFAULT_RINGING);

```

```

/*
** EXAMPLE: Change ringing to 71Vpk, 35Hz, Trap, 1.2 crest factor, 10vdc offset
*/

ProSLIC_dbgRingCfg ringCfg;

    ringCfg.ringtype = ProSLIC_RING_TRAP_CF12;
    ringCfg.amp = 71;
    ringCfg.freq = 35;
    ringCfg.offset = 10;

    ProSLIC_dbgSetRinging(ports[2].VoiceObj, &ringCfg, DEFAULT_RINGING);
}

```

7.6 Differential (isolated) PSTN Detection

The following section describes an implementation of the differential PSTN detection feature that is useful for monitoring for the presence of a foreign PSTN on an isolated design in which the foreign PSTN and ProSLIC have no common ground reference.

7.6.1 Compile Time Configuration

The following configuration parameters are defined in the `proslc_api_config.h` file. The user may modify these parameters to fit the needs of their application.

Parameter	Description
<i>PSTN_DET_ENABLE</i>	Enables compilation of Differential PSTN detection code. If this parameter is not defined, the differential PSTN detection code will not be compiled.
<i>PSTN_DET_POLL_RATE</i>	Rate at which reentrant function <code>ProSLIC_DiffPSTNCheck()</code> will be called. Default is 10ms.
<i>PSTN_DET_OPEN_FEMF_SETTLE</i>	Delay (in ms) from changing linefeed to OPEN state to making femf measurement. Default is 1500ms.
<i>PSTN_DET_DIFF_SAMPLES</i>	Number of voltage samples to be averaged. Measured at the rate defined by <code>PSTN_DET_POLL_RATE</code> . Default is 4.
<i>PSTN_DET_MIN_ILOOP</i>	Minimum loop current (in uA) to be considered valid reading. Large relative variations in small currents

	effect accuracy and need to be clamped to an absolute minimum. Default is 700uA.
<i>PSTN_DET_MAX_FEMF</i>	Foreign loop voltage threshold (if enabled) in mV. Default is 10000mV.
<i>PSTN_DET_DIFF_IV1_SETTLE</i>	Delay (in ms) from loading preset1 and making I/V measurement.
<i>PSTN_DET_DIFF_IV2_SETTLE</i>	Delay (in ms) from loading preset2 and making I/V measurement.

Table 18 PTSN Detection

7.6.2 Object Storage and Initialization

The user must create and initialize an object of type *proslicDiffPSTNCheckObjType* for each channel that will implement the differential PSTN detection function. In a typical implementation, the user would include a pointer to an instance of this structure in their channel data structure.

In the example below, the user has implemented a channel structure type *userVoiceChannelObj*, which includes the member *proslicDiffPSTNCheckObjType* **diffCheckObj*. Memory is allocated for these structures and the structures are initialized as shown below.

```

userVoiceChannelObj VoiceChans[NUMBER_OF_CHAN];    /* User's channel structure */

/*
** Create and initialize objects during hardware initialization
*/

int i;

for(i=0;i<NUMBER_OF_CHAN;i++) {

    ProSLIC_CreateDiffPSTNCheckObj (&(VoiceChans[i].diffCheckObj));

    ProSLIC_InitDiffPSTNCheckObj (VoiceChans[i].diffCheckObj,
```



```

        DCFEED_PSTN_DET_1,
        DCFEED_PSTN_DET_2,
        DCFEED_48V_20MA,
        FEMF_MEAS_ENABLE);
}

```

7.6.3 Executing the Differential PSTN Detection Test

Relatively long delays are necessary to allow line voltage changes to settle during the execution of the differential PSTN detection test. Therefore, the ProSLIC_DiffPSTNCheck() function was architected as a reentrant function that is called periodically at the rate defined by PSTN_DET_POLL_RATE. The user is responsible for the timing of the loop. An example implementation is shown below.

```

void check_for_foreign_pstn(userVoiceChannelObj *voiceChan)
{
    __int64 timeout;
    __int64 time1;
    int status = 0;
    int32 max_rl_ratio = 2000;

    while(status == RC_NONE)
    {
        /* Establish timeout at PSTN_DET_POLL_RATE */
        time1 = readCurrentTime(); /* User's getTime function, */
        timeout = (__int64)(PSTN_DET_POLL_RATE) + time1;

        /* Call test function -will return RC_NONE until complete/error */
        status = ProSLIC_DiffPSTNCheck(voiceChan->ProObj, voiceChan->diffCheckObj);

        while(readCurrentTime() < timeout); /* Wait for next poll period */
    }

    switch(status)
    {

        case RC_PSTN_OPEN_FEMF: /* PSTN Detected */
            LOGPRINT("Foreign PSTN Detected!\n");

```

```

        break;

    case RC_COMPLETE_NO_ERR:
        LOGPRINT("PSTN Check Completed!\n");

        if(voiceChan->diffCheckObj.rl_diff > max_rl_ratio)
        {
            LOGPRINT("Foreign PSTN Detected!!\n");
        }
        break;

    default:
        LOGPRINT("Unexpected Return Code in PSTN Check!!\n");
        break;
}

```

7.6.4 Interpreting Test Results

The ProSLIC_DiffPSTNCheck() function will return one of two codes upon completion.

RC_PSTN_OPEN_FEMF

This return code indicates the following has occurred

- Foreign voltage test was enabled
- Line was found to be ONHOOK, thus allowing the foreign voltage measurement to be made
- Measured foreign voltage was found to exceed the limit established by the parameter PSTN_DET_MAX_FEMF

RC_COMPLETE_NO_ERR

This return code indicates the following has occurred

- OPEN state voltage test was either not performed (line in use or feature was disabled) or did not measure a voltage in excess of the limit set by the parameter PSTN_DET_MAX_FEMF
- PSTN detection test was executed to completion

The *RC_COMPETE_NO_ERR* return DOES NOT indicate that a PSTN is not present, but rather, that the test executed to completion and measurement data is now available for analysis by the user. When the test returns *RC_COMPLETE_NO_ERR*, the user may examine the following members of the *proslcPSTNCheckObj* to make the determination if a PSTN is present or not.

rl_ratio Ratio of loop impedance measure at 2 DC Feed values. For a normal offhook loop, the ratio will be close to 1000 (ratio of 1.0, multiplied by 1000). A value > 2000 is typically a strong indicator that a foreign PSTN is present and altering the I/V characteristics of the loop.

vdifff1_avg The loop voltage measured using preset1. The user may check the polarity and magnitude of this voltage.

iloop1_avg The loop current measured using preset1. The user may check the polarity and magnitude of this current.

vdifff2_avg The loop voltage measured using preset2. The user may check the polarity and magnitude of this voltage.

iloop2_avg The loop current measured using preset2. The user may check the polarity and magnitude of this current.

ProSLIC_DiffPSTNCheck() does not analyze the results and make the determination of whether or not a PSTN is present. This is left to the user based on the characteristics of

their target system (maximum/minimum loop length, expected foreign PSTN characteristics, etc.)

7.7 Interrupt Handling

During hardware/software development, exposure to ringing conditions that exceed the capabilities of the design may result in power or thermal alarms. In some rare instances, a ringtrip may cause isolated corruption to the ProSLIC internal controller which requires a soft reset of the device to restore normal functionality.

Though these occurrences are limited to the lab environment and should never occur in a system that is appropriately designed for the target environment, enhanced interrupt handling has been developed to alert the user when this event occurs (as opposed to a power or thermal alarm due to other factors) and it is recommended that the user adopt this method for handling interrupts.

Unlike a “hard” reset, the soft reset/re-initialization effects only one channel and will not disturb adjacent channels on the same daisy chain – even if they are on the same multi-channel ProSLIC device.

The following code demonstrates how the enhanced interrupt handling may be used to alert the user that the channel must be reinitialized.

```
int usersPerChannelInterruptPollingFunc(proslicChanType_ptr pProslic)
{
    ProslicInt arrayIrqs[MAX_PROSLIC_IRQS];
    proslicIntType irqs;
    int intStatus;

    irqs.irqs = arrayIrqs;

    intStatus = ProSLIC_GetInterrupts(pProslic, &irqs);

    if(intStatus > 0) /* one or more interrupts are pending */
    {
        if(intStatus == RC_REINIT_REQUIRED) /* fatal error due to power/thermal alarm */
        {
            LOGPRINT("\n\n**** REINIT REQUIRED!!!! ****\n");
        }
    }
}
```

```

        ProSLIC_Reinit(pState[chan].ProObj,1);
        /*
        ** After soft reset, need to
        **   - reload desired Presets
        **   - set desired linefeed state
        **   - reload IRQEN (set as part of general parameters)
        */
        ProSLIC_DCFeedSetup(pProslic,USERS_DCFEED_PRESET);
        ProSLIC_RingSetup(pProslic, USERS_RINGING_PRESET);
        ProSLIC_PCMSetup(pProslic, USERS_PCM_PRESET);
        ProSLIC_ZsynthSetup(pProslic, USERS_IMPEDANCE_PRESET);
        ProSLIC_EnableInterrupts(pProslic);
    }
    else /* handle interrupts normally */
    {
        LOGPRINT("Num IRQs = %d\n", irqs.number);
        for(i=0;i<irqs.number;i++)
        {
            LOGPRINT("IRQ %d is set \n", irqs.irqs[i]);
            userInterruptHandler(irqs.irqs[i]);
        }
    }
}
else /* no pending interrupts */
{
    LOGPRINT("\n\n**** NO INTERRUPTS ****\n");
}
return (intStatus);
}

```

7.8 Neon Message Waiting Indication (MWI)

The ProSLIC API may be used to implement Message Waiting Indication (MWI) on older analog handsets that use a neon lamp requiring >90VDC illumination voltage. This feature is optional and all supporting code is conditionally compiled in if the user adds the following to *proslic_api_config.h*

```
#define SIVoice_NEON_MWI_SUPPORT
```

7.8.1 Configuring Neon MWI

Prior to enabling the MWI feature, `ProSLIC_MWISetV()` may be used to modify the configurable parameter that sets the MWI voltage while MWI is active.

Proprietary Information - No dissemination or use without prior written permission from Silicon Labs.

If the defaults are to be modified, this function must be called for all channels, as in the code examples below, and only needs to be called once during initialization.

```

/*
** Setup MWI for 98V peak voltage
*/
    for(i=0;i<NUMBER_OF_CHAN;i++)
    {
        ProSLIC_MWISetV(VoiceChans[i].pProslic, 98);
    }

```

7.8.2 Using Neon MWI

While the line is in the ONHOOK state, MWI may be enabled and the state toggled to flash the neon lamp. Toggling the state means changing the loop voltage between the normal onhook voltage that was established prior to enabling MWI, and the MWI active voltage, which is the default 80V or the modified value set by calling `ProSLIC_MWISetV()`.

The code below demonstrates use of the Neon MWI APIs:

```

/*
** Enable MWI on channel 1
*/
ProSLIC_MWIEnable(VoiceChans[1].pProslic);

/*
** Set the MWI state to ON
*/
ProSLIC_SetMWIState(VoiceChans[1].pProslic, MWI_FLASH_ON);

/*
** Set the MWI state to OFF
*/
ProSLIC_SetMWIState(VoiceChans[1].pProslic, MWI_FLASH_OFF);

/*
** Disable MWI on channel 1
*/

```

```
ProSLIC_MWIDisable(VoiceChans[1].pProslic);
```

Please refer to the ProSLIC API Documentation “Message Waiting Indicator routines” section for details.

7.8.3 Disabling Neon MWI After Loop Closure

In the event of a loop closure or ring-trip while Neon MWI is enabled, the user must call the `ProSLIC_MWIDisable()` function to ensure proper operation the next time the line is ONHOOK. It is presumed that the user has LCR/RTP monitoring capabilities (polling, irq, etc.) as part of their call progress state machine and has existing ISRs to deal with loop closure and ring-trip events.

7.8.4 Disabling Neon MWI When Not In Use

When ceasing to toggle the MWI state, `ProSLIC_MWIDisable()` should be called, rather than leaving it enabled in the MWI_FLASH_OFF state. The figure below is a simplified flowchart demonstrating MWI usage.

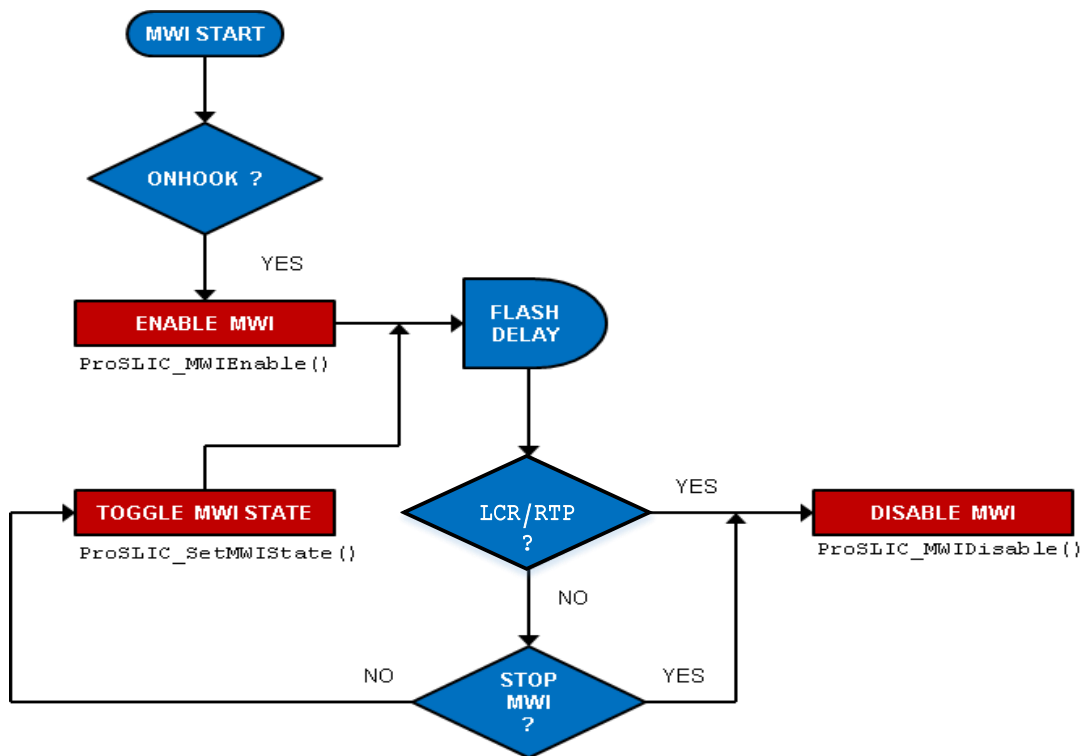


Figure 31. Simplified Neon MWI Flowchart

7.9 Configuring for Wideband Audio

To configure the ProSLIC device for Wideband Audio (16kHz PCM, 7kHz audio bandwidth) the user must load wideband-specific presets generated by the ProSLIC API Configuration Tool (Version 2.12.0 or later) for both the impedance synthesis block and the PCM interface.

7.9.1 Wideband Audio Example

Presuming wideband-compatible impedance and PCM presets, the following code example demonstrates how to enable wideband audio

```

/*
** Stop PCM bus (if already enabled)
*/
for(i=0;i<NUMBER_OF_CHAN;i++)
{
    ProSLIC_PCMStop(VoiceChans[i].pProslic);
}

/*
** Load wideband impedance coefficients and set
** channel gain (optional) BEFORE loading PCM preset
*/
for(i=0;i<NUMBER_OF_CHAN;i++)
{
    ProSLIC_ZsynthSetup(VoiceChans[i].pProslic, WB_ZSYN_600_0_0_20_0);
    ProSLIC_AudioGainSetup(VoiceChans[i].pProslic, -2, -4, WB_ZSYN_600_0_0_20_0);
}

/*
** Load wideband PCM preset
*/
for(i=0;i<NUMBER_OF_CHAN;i++)
{
    ProSLIC_PCMSetup(VoiceChans[i].pProslic, PCM_16LIN_WB);
}

/*
** Start PCM
*/
for(i=0;i<NUMBER_OF_CHAN;i++)
{

```



```
ProSLIC_PCMStart(VoiceChans[i].pProslic);  
}
```

7.9.2 Wideband Audio Low-Frequency Response

When wideband audio is enabled, the default low-frequency corner for both the TX and RX paths is set to 150Hz. This provides adequate fidelity, while minimizing 60Hz noise as well as group delay.

A compile option exists to disable the TX and RX high-pass filters when wideband audio is enabled. This is accomplished by defining the following parameter in *proslic_api_config.h*

```
#define DISABLE_HPF_WIDEBAND
```

8.0 *Multiple Device Configurations*

One can now use multiple general configurations for a single device in a single binary image. This requires manual editing of the constants file to include multiple general parameters and sharing of the remaining presets. In the API release, we provide examples multiple general configurations – such as `si3226x_MULTI_BOM_constants.c` and `si3217x_MULTI_BOM.c`. These example constants files, used in conjunction with the `ProSLIC_Init_MultiBOM()` function, setting `SIVoice_MULTI_BOM_SUPPORT` and a few other modifications, one can have multiple converter designs implemented in one binary. Please contact Silicon Labs for assistance in using this feature.

9.0 *Inward Testing With the ProSLIC API*

The ProSLIC API may be used to implement inward testing to verify the integrity and proper configuration of the hardware. This feature is optional and the user may elect to not include this code in their software. This feature can be used for implementing capabilities such as a self-test in the field and/or production testing.

The user may customize the inward tests to match their system by using the ProSLIC API Configuration Tool. This utility generates all the necessary source to configure the inward tests implemented in the ProSLIC API.

9.1 *Inward Test Descriptions*

The ProSLIC API supports the following inward tests

PCM Loopback

Enable/Disable PCM Loopback mode. User may select 16-bit linear format or a special 8-bit linear test mode for bit-exact loopback verification.

DC Feed, Loop I-V, and Loop Closure Verification

Measures on-hook and off-hook I/V characteristics of the DC feed without the use of an external load. Optional LCS check to support system level signaling behavior. Please note: the chipsets, depending on DCFeed settings, may not report accurately the off-hook conditions.

Ringin and Ringtrip Verification

Measures open-circuit ringin voltage (AC and DC). Optional RTP check to support system level signaling behavior

Battery Verification

Measures VBAT

TX and RX Path Audio Gain Verification

Measures the gain of the RX path (PCM to line) and TX path (line to PCM) without the use of an external AC load or test equipment. Typical accuracy is 1 dB.

9.2 Inward Test Configuration

9.2.1 Launching the Test-In Configuration Utility

The ProSLIC API Configuration Tool is used to generate C source code used by the ProSLIC API to configure the inward tests. This configuration window may be accessed by selecting: *Utilities->Test-In->Test-In Configuration*, which launches the window show below.

ProSLIC Test-In Configuration

PCM LOOPBACK
PCM WORD SIZE: 8-Bit

DC FEED
ENABLE LCR TEST: ☒
ABORT IF LINE-IN-USE: YES
Alt LCROFFHK (mA): 8.000
Alt LCRONHK (mA): 6.000

TEST LIMITS	ON-HOOK		OFF-HOOK	
	MIN	MAX	MIN	MAX
VTIP	1.000	5.000	2.000	10.000
VRING	45.000	58.000	44.000	51.000
VLOOP	43.000	52.000	34.000	46.000
VBAT	43.000	62.000	36.000	60.000
ITIP	-2.000	2.000	8.000	20.000
IRING	-2.000	2.000	-20.000	-8.000
ILOOP	-1.500	1.500	8.000	20.000
ILONG	-2.000	2.000	-2.000	2.000

RINGING
ENABLE RTP TEST: ☒
ABORT IF LINE-IN-USE: YES
SAMPLE SIZE: 90.000
SAMPLE INTERVAL (ms): 10.000

TEST LIMITS	MIN	MAX
VAC	45.0	70.0
VDC	-3.0	3.0

BATTERY

TEST LIMITS	MIN	MAX
VBAT	50.000	60.000

AUDIO
ABORT IF LINE-IN-USE: YES
0dBm REF (mVpk): 1095.000

TEST LIMITS	MIN	MAX
TX PATH GAIN	-11.000	1.000
RX PATH GAIN	-11.000	1.000

OK Apply Cancel

Inputs Valid

Figure 32. Test-In Configuration Window

From this single window, all inward tests may be configured. The following table enumerates the inward test parameters that are configurable and provides a brief description of their function

Test	Parameter	Description
PCM Loopback	PCM Word Size	8 or 16-bit linear format
DC Feed	Enable LCR Test	Enable to apply alternate LCR thresholds prior to test load

Proprietary Information - No dissemination or use without prior written permission from Silicon Labs.

Test	Parameter	Description
		being connected. This allows the user to set LCR thresholds that ensure a loop closure occurs when the test load is connected.
	Abort If Line-In-Use	If YES is selected, upon entry, the test will check to see if LCR is set. If so, it will return and not execute the test
	Alt LCROFFHK	Alternate threshold that results in LCR being set when transitioning from on-hook to off-hook state
	Alt LCRONHK	Alternate threshold that results in LCR being cleared when transitioning from off-hook to on-hook state
	VTIP	Min/Max TIP voltage in on-hook and off-hook states
	VRING	Min/Max RING voltage in on-hook and off-hook states
	VLOOP	Min/Max loop voltage in on-hook and off-hook states
	VBAT	Min/Max battery voltage in on-hook and off-hook states
	ITIP	Min/Max TIP current in on-hook and off-hook states
	IRING	Min/Max RING current in on-hook and off-hook states
	ILOOP	Min/Max loop current in on-hook and off-hook states
	ILONG	Min/Max longitudinal current in on-hook and off-hook states
Ringing	Enable RTP Test	Enable to test RTP functionality by applying low voltage ring signal that may be tripped by the internal test load. This DOES NOT test for ringtrip with the ringer settings loaded prior to executing the test
	Abort If Line-In-Use	If YES is selected, upon entry, the test will check to see if LCR is set. If so, it will return and not execute the test
	Sample Size	Number of samples taken to measure the open-circuit ringing voltage
	Sample Interval	The interval in which ringing voltage samples are taken (ms)

Test	Parameter	Description
	VAC	Min/Max Ringer AC voltage limit (vrms)
	VDC	Min/Max Ringer DC voltage limit (v)
Battery	VBAT	Min/Max battery voltage
Audio	Abort If Line-In-Use	If YES is selected, upon entry, the test will check to see if LCR is set. If so, it will return and not execute the test
	0dBm Ref	0dBm voltage (in mvpk) for the desired characteristic impedance (eg. 0dBm for 600Ω = 1095 mvpk)
	TX Path Gain	Min/Max gain of the TX path (Line to PCM)
	RX Path Gain	Min/Max gain of the RX path (PCM to Line)

Table 19. Test-In Configuration Options

9.2.2 Generating Source Code

Once all parameters and limits are updated, source code may be generated by selecting *Utilities->Test-In->Generate Test-In Source Code*. This will create two files (default names referenced here, but the user may rename)

proslic_tstin_limits.c

Contains initialization values for the test structures used by the ProSLIC API inward test feature

proslic_tstin_limits.h

Contains extern declarations of the test structures instantiated in *proslic_tstin_limits.c*. This header will need to be included in any file that calls the inward test APIs that will be described in the next section.

9.2.3 Saving/Reloading Inward Test Configurations

The user may save their inward test configuration as an XML file in order to reload it into the ProSLIC API Configuration Tool at a later date. In the event that the Test-In Configuration utility is expanded in the future, having an XML version of the parameters would allow the user to load

them into the new ProSLIC API Configuration Tool and minimize the need to reenter information. This can be done by selecting *Utilities->Test-In->Save Test-In Config (xml)*.

To reload parameters store in XML format during a later session or on a newer version of the ProSLIC API Configuration Tool, select *Utilities->Test-In->Load Test-In Config (xml)*.

9.3 Inward Test API Support

9.3.1 Additional Files

In addition to the two configuration files generated by the ProSLIC API Configuration Tool, to implement inward testing, the following files must be included in the user's build

<i>proslic_tstin.c</i>	All inward test API definitions and source code
<i>proslic_tstin.h</i>	All inward test API declarations. Must be included in any code that calls inward test functions

9.3.2 Inward Test Data Structures

In support of the inward tests, new data structures are defined that are independent of the channel structures used by the ProSLIC API.

One instance of the data type `proslicTestInObjType` must be instantiated per channel to support inward testing. This data structure encapsulates data structures for each inward test. The user may link this to their existing channel data structure, as will be shown in later examples.

```
typedef struct {
    proslicPcmLpbkTest   pcmLpbkTest;
    proslicDcFeedTest    dcFeedTest;
    proslicRingingTest   ringingTest;
    proslicBatteryTest   batteryTest;
    proslicAudioTest     audioTest;
}proslicTestInObjType;
```

9.3.3 User Linkage to Inward Test Data Structures

Just as is done with the ProSLIC channel data structure (`proslicChanType`), the user's per-channel data structure needs to link to an instance of the ProSLIC Inward Test data structure (`proslicTestInObjType`) for each channel that will support inward testing. For simplicity, the user may link to a pointer to this data structure and utilize the memory allocation APIs provided.

For example, if the user's channel data structure, `chanStat`, is defined as

```
typedef struct {
    proslicChanType      *ProObj;
    proslicTestInObjType *pTstin;
    int                  numberOfChan;
}chanState;
```

and the user has instantiated one instance of this data structure per channel as follows

```
chanState ports[MAX_NUMBER_OF_CHAN];
```

the user may allocate memory for the inward test data structure using the provided create function.

```
ProSLIC_createTestInObjs(&(ports[0].pTstin), ports[0].numberOfChan);
```

Likewise, the user may free memory using the provided destroy function

```
ProSLIC_destroyTestInObjs(&(ports[0].pTstin));
```

9.3.4 Inward Test Data Structure Initialization

Each inward test data structure provides storage for configuration parameters, test limits, and the test results and must therefore be initialized per channel. A unique initialization function for each inward test is provided to allow the user to independently select tests to be run while also allows for future enhancements to the inward suite of tests. The initialization functions are listed below and an implementation example is provided in a later section.

```
ProSLIC_testInPcmLpbkSetup
```

```
ProSLIC_testInDcFeedSetup
```

```
ProSLIC_testInRingingSetup
```



```
ProSLIC_testInBatterySetup
```

```
ProSLIC_testInAudioSetup
```

9.3.5 Inward Test API Definitions

The following section describes the inward test APIs. Please refer to the ProSLIC API Documentation for function parameters. Each test typically returns one of the following return values:

<i>RC_UNSUPPORTED_FEATURE</i>	Returned if unsupported by device
<i>RC_TEST_DISABLED</i>	Returned if test not previously initialized
<i>RC_LINE_IN_USE</i>	Returned if LCS is already set at start of test
<i>RC_TEST_FAILED</i>	Returned if test completed but results outside of test limits defined.
<i>RC_TEST_PASSED</i>	Returned if test completed and results within test limits (if any).

9.3.5.1 PCM Loopback

The PCM loopback test is the only test that the user is responsible for making the pass/fail determination. The following APIs are provided to place the ProSLIC device in the proper PCM loopback state and to restore the PCM interface to its entry conditions. The user is responsible for sourcing and capturing PCM data and making the determination that the interface is working properly.

Functions:

```
ProSLIC_testInPCMLpbkEnable
```

```
ProSLIC_testInPCMLpbkDisable
```

9.3.5.2 DC Feed

The DC Feed test measures all the vital I/V characteristics of the DC feed with and without the embedded test load connected. A test failure is determined by checking the limits defined in pTstin->dcFeedTest settings. Please note: the off-hook values may not represent the configured values in all cases.

Proprietary Information - No dissemination or use without prior written permission from Silicon Labs.

Function:

```
ProSLIC_testInDCFeed
```

9.3.5.3 Ringing

The Ringing test measures the ringing voltage and optionally verifies RTP using a low level ring signal that allows the internal test load to cause a ringtrip when connected. A test failure is determined by checking the limits set in pTstin->ringingTest.

Function:

```
ProSLIC_testInRinging
```

9.3.5.4 Battery

The Battery test measures the battery voltage under the entry conditions. A test failure is determined by checking the limits set in pTstin->batteryTest

Function:

```
ProSLIC_testInBattery
```

9.3.5.5 Audio

The Audio test measures the gain of the RX (PCM to line) and TX (line to PCM) paths. . A test failure is determined by checking the limits set in pTstin->audioTest. Typical values are accurate to 1 dB of configured values.

Function:

```
ProSLIC_testInAudio
```

9.3.5.6 Simulated Hook state

This test simulates an offhook state change – testing if the higher level software detects this change. Only one channel can run this test at a time due to caching requirements. Unlike other functions, this test function is meant to be called twice – once to set the state and another to restore it with the upper layer code interrupt/hook state function injected between the two calls.

Please refer to the API demo software on example usage. **NOTE:** code assumes LCR interrupt is enabled in order for it work correctly.

Function: ProSLIC_testLoadTest

9.4 Inward Test Code Example

The following section is an example implementation. This code example presumes the ProSLIC channels have all been properly initialized, enabled, and are in the FWD Active linefeed state. It is also assumed that the user has implemented a PCM loopback test to validate the DTX data stream is identical to what is sourced on DRX. It is called *User_PCM_Test()* and returns RC_TEST_PASSED or RC_TEST_FAILED.

```

/*
** Necessary includes in user's files that call into the inward test APIs
*/
#include "proslic.h"
#include "proslic_tstin.h"
#include "proslic_tstin_limits.h" /* Generated by the proslic_api_config_tool */

chanState ports[MAX_NUMBER_OF_CHAN];
int testResult;
/*
** Allocate memory for inward test data structures
*/
ProSLIC_createTestInObjs(&(ports.pTstin), ports[0].numberOfChan);

/*
** Initialize test objects. Pointers to test limits defined in limits
** file that was generated by the proslic_api_config_tool.
*/
for(i=0;i<ports[0].numberOfChan;i++)
{
    ProSLIC_testInPcmLpbkSetup(ports[i].pTstin,&ProSLIC_testIn_PcmLpbk_Test);
    ProSLIC_testInDcFeedSetup(ports[i].pTstin,&ProSLIC_testIn_DcFeed_Test);
    ProSLIC_testInRingingSetup(ports[i].pTstin,&ProSLIC_testIn_Ringing_Test);
    ProSLIC_testInBatterySetup(ports[i].pTstin,&ProSLIC_testIn_Battery_Test);
    ProSLIC_testInAudioSetup(ports[i].pTstin,&ProSLIC_testIn_Audio_Test);
}
/*
** Call each inward test sequentially, OR'ing results.

```

Proprietary Information - No dissemination or use without prior written permission from Silicon Labs.

```
*/
for(i=0;i<ports[0].numberOfChan;i++)
{
    testResult = RC_TEST_PASSED;
    /* PCM Test */
    ProSLIC_testInPcmLpbkEnable(ports[i].ProObj, ports[i].pTstin);
    testResult |= User_PCM_Test();
    ProSLIC_testInPcmLpbkDisable(ports[i].ProObj, ports[i].pTstin);
    /* DC Feed Test */
    testResult |= ProSLIC_testInDCFeed(ports[i].ProObj, ports[i].pTstin);
    /* Ringing Test */
    testResult |= ProSLIC_testInRinging(ports[i].ProObj, ports[i].pTstin);
    /* Battery Test */
    testResult |= ProSLIC_testInBattery(ports[i].ProObj, ports[i].pTstin);
    /* Audio Test */
    testResult |= ProSLIC_testInAudio(ports[i].ProObj, ports[i].pTstin);

    if(testResult == RC_TEST_PASSED)
    {
        LOGPRINT("\nALL TESTS PASSED!!\n");
    }
    else
    {
        LOGPRINT("\nTEST FAILED!!\n");
    }
}
```

10.0 ProSLIC API Demo Applications

As part of this release, we provide the following applications:

- 1) A basic platform validation program. This program can be used to check if the system services the platform must provide to the ProSLIC API. At present, this checks register and RAM access and that the reset function can be executed.
- 2) A simple register & RAM read/write application. This may be useful to debug any issues found in the first application. This application does not initialize the chipsets and does not probe to identify which chipset is being used.
- 3) A simple demo application is included with the ProSLIC API to allow one to try the API with various evaluation boards. The demo will initialize the given evaluation board, allow one to try different constant presets, check interrupt behavior at a coarse grain (it's over USB for), and generate tones.
- 4) A basic utility to read the EEPROM from ProSLIC. This utility comes with a shell script to interpret the results to suggest which Makefile build option should be used by the API demo program.
- 5) A simple PBX demo that will generate dialtone, collect digits (DTMF or pulse), ring the corresponding phone, and perform an audio cross-connect. When used with the Caller-ID framework, it will generate type-1 Caller-ID (North American).

All the applications can be built either for Linux® desktop/userspace, Cygwin, or MinGW. The makefile detects which OS make is running from, so no additional arguments are needed. Please refer to the appropriate “quick start” guides for details on environment setup and initial executable generation.

For embedded Linux® we now provide an example of how to use the “spidev” interface and a Linux® kernel driver. This is covered in a separate document. The same build instructions found here can be used for the spidev interface demo code.

10.1 *Demo directory structure*

The demo software is installed under:

`<ProSLIC API Installation Root>/prosllic_api/demo`

Under this directory we have:

<code>api_demo/</code>	Simple demo application to try the ProSLIC API
<code>id_evb/</code>	Utility to read and identify which EVB is connected to the VMB1 or VMB2 (spidev is not supported).
<code>pform_test/</code>	Basic platform services test program
<code>platform/</code>	Platform (Cygwin, Linux, MinGW) specific support files
<code>rdwr/</code>	Simple register & RAM read/write application directory.
<code>prosllic_api_core.mk</code>	Generic makefile include file (chipset/convert dependencies)
<code>prosllic_api_options.mk</code>	Makefile to convert command line options to arguments used in <code>prosllic_api_core.mk</code>

Each demo has the following structure:

<code>build/</code>	Build directory (location of Makefile and temp build files)
<code>src/</code>	Application specific source code
<code>inc/</code>	Application specific include files
<code>custom/</code>	ProSLIC API configuration files (if needed)

10.2 *Linux desktop VMB2 support*

On the Linux desktop, the VMB2 (blue-green PCB) is only supported and assumes the Silicon Labs VCP driver is already available as either as a kernel module or as part of the kernel. When plugging in the VMB2 to your Linux desktop system, you should see via `dmesg` the following message (or something similar):

```
usb 8-1: cp210x converter now attached to ttyUSB1
usbcore: registered new interface driver cp210x
cp210x: v0.09: Silicon Labs CP210x RS232 serial adaptor driver
```

Proprietary Information - No dissemination or use without prior written permission from Silicon Labs.

In the above, a device inode of /dev/ttyUSB1 was created. You will need to ensure you are part of the correct group to access this device (typically the dialout group). Please refer to your Linux distribution for how to add yourself to the group and/or how to include the CP210x driver. Testing was done on:

- Linux Mint 14.0 3.2.0-60-generic kernel #91-Ubuntu system (x86_64)
- Linux Mint 18.3 4.4.0-109-generic #132-Ubuntu SMP
- Debian 7.11 3.8.13-bone80

The demos for Linux assume /dev/ttyUSB0 for the COM port. This can be overwritten with the following shell script command:

```
export SIVoice_SPI_IF=/dev/<device name>
```

For example:

```
export SIVoice_SPI_IF=/dev/ttyUSB1
```

Using the wrong COM port may have the application hang waiting on data from the VMB2. Just type CTRL-C to quit and correctly configure the environment variable if this occurs.

Besides GCC or Clang and GNU make, GDB (including GDB server for your target system), cscope or ctags can be useful for trying out the various demos under Linux. An IDE such as Eclipse can simplify your work as well with these applications (at least on the desktop).

10.2.1 *Debugging with the Linux Desktop*

The makefile does support building the executables with debug symbols (see section 10.9.4). The default debugger is GDB. If GDB command line is a bit intimidating for you, you can use various GDB front ends – from DDD to Eclipse. Please refer to your Linux distribution documentation on how to install your debugger.

10.3 *Cygwin VMB1 & VMB2 support*

For Cygwin, we support both the Voice Motherboard (VMB) 1 (red PCB) or 2 (blue-green PCB). The user must add to their path, the DLL directory located under <ProSLIC API Installation
Proprietary Information - No dissemination or use without prior written permission from Silicon Labs.

Root>/proslc_api/demo/platform/cygwin/bin. To add to the path, one can either set it relative to the build directory for each demo, or set it to absolute path. For a relative path, the following command can be used:

```
export PATH=$PATH:../../platform/cygwin/bin/
```

or for absolute path:

```
export  
PATH:$PATH:/cygdrive/c/Silabs/ProSLIC/proslc_api_8.0.0/proslc_api/demo/  
platform/cygwin/bin
```

10.3.1 *Installing Cygwin*

Please refer to the “Quick Start Guide – Windows_Cygwin” for details for installation and use.

There are several web sites mentioning that one can use IDE's such as Eclipse with Cygwin, if that is your preference. The provided makefile does not require an IDE to build or run the software.

10.3.2 *Debugging with Cygwin*

The makefile does support building the executables with debug symbols (see section 10.9.4). The default debugger is GDB. If GDB command line is a bit intimidating for you, you can install Cygwin-X and run DDD – Data Display Debugger which is a GUI front end to GDB. Please see <http://x.cygwin.com/> for installation instructions for this environment. After installation, you will need to make sure you also install DDD from the Cygwin setup program.

The alternative to Cygwin-X & DDD is the Eclipse IDE environment. If using this alternative, it is advised to install Eclipse-CDT – which is Eclipse with the C Developers Toolkit (CDT) already included. Please see <http://www.eclipse.org/home/index.php> for downloading this package.

The advantage of Eclipse is it's a full IDE, but the setup is more complicated. You can import an existing Makefile project into Eclipse and build/debug from there. The following web sites may be useful:

http://www3.ntu.edu.sg/home/ehchua/programming/howto/eclipsecpp_howto.html

Proprietary Information - No dissemination or use without prior written permission from Silicon Labs.

Copyright © 2018 Silicon Labs

<http://wyding.blogspot.com/2009/04/setup-cygwin-toolchain-in-eclipse-cdt.html> - the big take away is path mapping from this site.

We do suggest adding FLUSH_STDOUT=1 build option to cause the software to flush stdout prompts when executing the program. When debugging, it seems the prompts do not show up at the right times, even with the suggested fixes. Please see: <http://wiki.eclipse.org/CDT/User/FAQ> and search for “Eclipse console does not show output on Windows” for information about why this option is needed for Eclipse.

The “Quick Start Eclipse MinGW” has some useful information on how to incorporate the ProSLIC API demo with Eclipse that can also apply to Cygwin.

10.4 MinGW VMB1 & VMB2 support

For MinGW, we support both the Voice Motherboard (VMB) 1 (red PCB) or 2 (blue-green PCB). The user must add to their path, the DLL directory located under *<ProSLIC API Installation Root>/proslc_api/demo/platform/cygwin/bin* (we use the same DLL's as Cygwin). To add to the path, one can either set it relative to the build directory for each demo, or set it to absolute path. For a relative path, the following command can be used:

```
export PATH=$PATH:../../platform/cygwin/bin/
```

or for absolute path:

```
export PATH=$PATH:/c/Silabs/ProSLIC/proslc_api_9.0.0/proslc_api/demo/
platform/cygwin/bin
```

10.4.1 Installing MinGW

Please refer to the “Quick Start Guide – Eclipse_MinGW” for installation instructions.

For your console, make sure you have the following settings (add this in your ~/.profile file):

Environment variable name	Setting
CC	gcc
PERL_PATH	/bin/perl.exe

Example .profile file:

```
export CC=gcc
export PROSLIC_INSTALL=/c/SiLabs/ProSLIC/proslic_api_8.1.0/proslic_api/
alias goproslicapi='cd $PROSLIC_INSTALL'
alias goproslicdemo='cd $PROSLIC_INSTALL/demo'
```

NOTE: ~/.profile is under: c:\MinGW\msys\1.0\home\<your windows id>\.profile – assuming the default install directory is used.

To start the MinGW shell, you can create a shortcut to c:\MinGW\msys\1.0\msys.bat on your desktop and start a new shell.

10.4.2 *Debugging with MinGW*

MinGW supports gdb and Eclipse. For a “GUI” experience, it is suggested that Eclipse should be used. Please refer to the “Quick Start Guide Eclipse MinGW” for how to configure Eclipse to be used with the ProSLIC API demo program.

10.5 *Remote SPI (RSPI)*

Starting in ProSLIC API 9.2.0, the remote SPI (RSPI for short) was introduced. This new feature allows the developer to use the ProSLIC demo software to control a ProSLIC on a remote system, such as a target system to debug or do initial bring up of the ProSLIC.

The RSPI code uses UNIX standard TCP/IPv4 socket I/O on a portable “server” that can be embedded into a system with a network connection. The protocol mimics the “systems layer” of the ProSLIC API and is agnostic to the chip family used. The protocol used is meant for an internal network and should not be used on the internet.

Building for RSPI

For all demos provided, we provide a new build option RSPI=1. This can be used in lieu of VMB1=1, VMB2=1 or SPIDEV=1 for these demos.

The server software is located under demo/server. As with other demos, we have a build directory with a makefile. This makefile accepts the normal platform targets of VMB1=1, VMB2=1, or SPIDEV=1. It will not accept RSPI=1. The resulting executable will be called rspi_server[.exe].

Configuring the RSPI client & server

RSPI can be configured at build time or at run time. The build time settings are located in the custom folder of each demo in the file `rspi_client_cfg.h`. In here you have two network parameters: `rspi_ip_address` and `rspi_port_num`. These parameters refer to the server address and port number.

In addition, you have a plain text user name and password. These are `rspi_username` and `rspi_password`. User name and password are limited to 20 characters.

Alternatively, you can have a plain text file that stores the same parameters in a file called `rspi_client_cfg.txt`. In order to enable use of this file, you need to compile with the option `RSPI_TXT_CFG=1`.

On the server end you can have more than 1 user associated with the server by changing `RSPI_NUM_USERS`. Parameters for the server are located in the custom folder for the server under `rspi_server_cfg.h`.

Platforms tested

This new feature has been tested under Cygwin, MinGW, and user space Linux (running on a few different platforms).

Proxy setup

If your network has your target systems on a separate subnet, you could use iptables under Linux (and other UNIX style OS's). On a modern Linux distribution, this can be done with the following commands (the example has 192.168.7.2 as the "server" and ports are 7890):

```
sysctl net.ipv4.ip_forward=1 #systemd based systems need this!
iptables -t nat -A PREROUTING -p udp --dport 7890 -j DNAT --to-destination 192.168.7.2
iptables -t nat -A PREROUTING -p tcp --dport 7890 -j DNAT --to-destination 192.168.7.2
iptables -t nat -A POSTROUTING -j MASQUERADE
```

10.6 Platform test tool

This application is installed under the demo directory `pform_test`. This program can be used prior to integrating the ProSLIC API into your target system to check that the system services the ProSLIC API needs are implemented correctly. At present, this program checks the basic SPI operations:

- Register Read/Write
- RAM Read/Write
- PCM/Framesync are valid
- Reset function is operational
- For multiple channel implementations, checks for control word channel encoding

In addition, the validation program does a basic timer test and then if the SPI & timer tests pass, provides a rough throughput value.

The software assumes to be running in “userspace”, however one could modify the code to run in kernel space for Linux. To test multiple channels, specify the number of channels to test at the command line. For example:

```
$ ./pform_tester 2
```

```
Basic platform validation tool version 0.1.0
```

```
Copyright 2015-2016, Silicon Laboratories
```

```
-----
```

```
Demo: Linking function pointers...
```

```
Demo: VMB2 VCP Connected via COM8
```

```
Demo: VMB2 Firmware Version 1.8
```

```
Change Default VMB2 Settings (y/n) ?? ->n
```

```
Demo: Resetting device...
```

```
Testing channel 0
```

```
spiReadRevIDTestCommon      : PASSED
```

```
spiBasicWriteTestCommon     : PASSED
```

```
spiMasterStatusTest        : PASSED
```

```
spiBasicRAMReadTest        : PASSED
```

```
spiBasicRAMWriteTest       : PASSED
```

```
spiResetTestCommon         : PASSED
```

```
Testing channel 1
```

```
spiReadRevIDTestCommon      : PASSED
```

Proprietary Information - No dissemination or use without prior written permission from Silicon Labs.

Copyright © 2018 Silicon Labs

```

spiBasicWriteTestCommon      : PASSED
spiMasterStatusTest          : PASSED
spiBasicRAMReadTest          : PASSED
spiBasicRAMWriteTest         : PASSED
spiResetTestCommon           : PASSED

SpiMultiChanTest             : PASSED

timerElapsed_1Sec            : PASSED
timerElapsed_10mSec          : PASSED
timerElapsed_2015mSec        : PASSED
timerElapsed_5mSec           : PASSED

Testing channel 0
spiReadTimedTestCommon       : measured: 2729 mSec, for 10000 runs.  Had 0 errors
spiReadTimedTestCommon       : PASSED
spiWriteTimedTestCommon      : measured: 2691 mSec, for 10000 runs.
spiWriteTimedTestCommon      : PASSED
Testing channel 1
spiReadTimedTestCommon       : measured: 2747 mSec, for 10000 runs.  Had 0 errors
spiReadTimedTestCommon       : PASSED
spiWriteTimedTestCommon      : measured: 2672 mSec, for 10000 runs.
spiWriteTimedTestCommon      : PASSED

```

Please refer to the source files to see how each test works.

To build the application, from your shell:

```

cd <ProSLIC API Installation Root>/proslic_api/demo/pform_test/build

make <VMB1|VMB2|SPIDEV>=1 [DAA=1]

```

The resulting executable will be called `pform_test` (Linux) or `pform_test.exe` (Cygwin) for the ProSLIC build and for the DAA (Si3050) build, `pform_test_daa` (Linux) or `pform_test_daa.exe` (Cygwin).

To port this code to your target system, the main code changes would be in adding in your SPI and timer functions to the makefile, changing the file `pform_test.c` to refer to your system services `init/teardown` functions and using a cross-compiler, as needed. The outputs are all defined via macros found in `pform_test.h`. No memory allocation is used as part of the test code. Only the VMB functions interact directly with the user for various settings.

Proprietary Information - No dissemination or use without prior written permission from Silicon Labs.

10.7 Building the rdwr example code

The rdwr program is a simple program to access the ProSLIC or VDAA. It provides a simple command line to read & write register and RAM locations and a write/verify SPI bus command. This application is installed under the demo directory rdwr. To build the application, from your shell:

```
cd <ProSLIC API Installation Root>/prosllic_api/demo/rdwr/build

make <VMB1|VMB2|SPIDEV>=1
```

When you start the application, the program has the following commands:

```
rd <channel> <register>
wr <channel> <register> <value>
rrd <channel> <RAM ADDRESS>
rwr <channel> <RAM ADDRESS> <value>
wrv <channel> <register> - send a simple pattern to a particular channel and
verify the write.
reset <reset mode> - reset all device(s) 0 = out of reset, 1 = reset

exit - quit program
```

The program does not use the ProSLIC API higher level functions – instead it is designed as a low level tool for debugging any hardware integration issues.

10.8 Building & using the EVB ID utility

Located in:

```
<ProSLIC API Installation Root>/prosllic_api/demo/id_evb/build>
```

The EVB identification program, which reads a programmed EEPROM on the EVB and prints the pertinent information to help identify which EVB is installed on the VMB1 or VMB2.

To build this utility, cd to the build directory and type either:

```
make VMB1=1
```

Proprietary Information - No dissemination or use without prior written permission from Silicon Labs.

Copyright © 2018 Silicon Labs

or

```
make VMB2=1 [ISI=1]
```

The typical output is as follows:

```
0x14:Si3217xFB-EVB:1.0:Si3251:B
```

When executed on a board that does not have the EEPROM programmed or a Si3050 board, the following output will be displayed (the last parameter may differ):

```
0xFF:UNKNOWN:UNKNOWN:UNKNOWN:A
```

The ISI=1 option is needed to inform the utility that the EVB has an ISI interface. So for instance for the Si3219x family of SLICs one would need to use this:

```
make VMB2=1 ISI=1
```

A shell script is provided that maps a valid output to a string that can be used by the Makefile for the API demo. The shell script is called map.sh. When executed, the following is displayed:

```
./map.sh
Reading EEPROM string - this will take a few seconds
Rev = B
SI3217X_B_FB
```

The last line shows for this particular board, it believes that the SI3217X_B_FB is the correct value to use in the ProSLIC API demo makefile.

10.9 Building the API Demo

To build a demo application, from a terminal:

```
cd <ProSLIC API Installation Root>/prosllic_api/demo/api_demo/build

make <evb option>=1 [NO_LBCAL=1] <VMB1|VMB2|SPIDEV>=1 [DEBUG=1]
[PROSLIC_CFG_SRC=<your constants file>] [LOG_ALL=1] [NO_VDAA=1] [LUA=1]
```

Proprietary Information - No dissemination or use without prior written permission from Silicon Labs.

Copyright © 2018 Silicon Labs

The *Table 20* enumerates the available EVB options and the corresponding output files created.

For example, to build for the Si3217x Revision C BJT Buck-Boost EVB using VMB2,

```
make SI3217X_C_BB=1 VMB2=1
```

The NO_LBCAL=1 option lets you skip longitudinal balance calibration during initialization.

The LOG_ALL=1 flag will log all debug and trace messages to a file. The debug menu can be used to turn off logging.

The NO_VDAA=1 flag is used in conjunction with the Si3217x chipsets to disable the VDAA detection and menus. For example, if you are using the Si32178/9 chipset and are not using the VDAA option, set this flag.

You can also specify a different constants file from the default one by using the option PROSLIC_CFG_SRC=<fn> where <fn> is the constants file to be built with the demo. Example:

```
make SI3217x_B_FB=1 VMB2=1 PROSLIC_CFG_SRC=my17xConstants.c
```

The default location for the constants file is under proslic_api/demo/api_demo/custom.

Please refer to proslic_api_core.mk and the Makefile for other switches that are available. In addition, you can look at the buildtargets.txt file with the chipset & converter options currently supported.

10.9.1 *Building for ProSLIC + Si350..*

The ProSLIC API demo can support both a ProSLIC and a Si3050 at the same time. In the API demo code we provide example of a Si3226x Rev C with a Flyback converter and a Si350 connected. The information below is the code changes needed to support such a configuration.

Please note: other configurations such as having two different ProSLIC's are not supported in the makefile at this time.

In order to enable this, you will need to modify the following files:

- demo\proslc_api_options.mk – create a new combination setup. We provide an example of a Si3226x FB and Si3050 (search for 26x_3050).
- demo\api_demo\custom\demo_config.h - change DEMO_PORT_COUNT to the number of dissimilar EVB's. For the 26x_3050 setup, this would be 2 instead of 1.
- demo\api_demo\custom\demo_config.h - set the number of devices as needed.
- demo\api_demo\custom\demo_config.h - #define SIVoice_USE_CUSTOM_PORT_INFO 1 – this disables the default port/device mapping in demo\platform\common\demo_common.c.

You will need to implement the following function:

```
void demo_init_port_info(demo_port_t *port, unsigned int port_id)
```

This function takes in the port_id index and fills in the port structure with the device type, etc. The following code snippet is an example for the Si3226x in the first “port” and the second “port” a setup of Si3050's...

```
void demo_init_port_info(demo_port_t *port, unsigned int port_id)
{
    switch(port_id)
    {
        case 0:
            printf("Setting up Si3226x\n");
            port->deviceType = SI3226X_TYPE;
            port->numberOfDevice = SI3226X_NUMBER_OF_DEVICE;
            port->numberOfChan = SI3226X_NUMBER_OF_CHAN;
            port->chanPerDevice = SI3226X_CHAN_PER_DEVICE;
            break;

        case 1:
            printf("Setting up Si3050\n");
            port->deviceType = SI3050_TYPE;
```

```

    port->numberOfDevice = SI3050_NUMBER_OF_DEVICE;
    port->numberOfChan    = SI3050_NUMBER_OF_CHAN;
    port->chanPerDevice   = SI3050_CHAN_PER_DEVICE;
    break;

default:
    printf("Unknown port: %d\n", port_id);
    break;
}
}

```

The above example is in `api_demo.c` function...

10.9.2 *A word on software build resources included in the API*

Included in the release is a BASH script under the `demo/cygwin/build` directory called `buildall.sh` – this script will build for both VMB1 and VMB2 all the different chipset/converters listed under `buildtargets.txt` and places the executables under a `tmp` directory. The script renames the executables to include a suffix for which Voice Motherboard (VMB) it was built for. For example, `si3226x_c_qs_vmb2.exe` instead of `si3226x_c_qs.exe`. After running the script, to run a demo you would type: `tmp/<exe name>`. For example: `tmp/si3226x_c_qs_vmb2.exe` (under Cygwin or `tmp/si3226x_c_qs_vmb2` under Linux)

In addition, we've included under the `proslc_api/demo` directory a makefile fragment called `proslc_api_core.mk`. This file creates a dependency list of which patch, constants, and interface files are needed to build an application/driver based upon which converter and chipset is supported. Please examine the ProSLIC API demo Makefile on how it is used. We suggest customer adopt this file since it is updated for every release. Please note: at present, MULTI-BOM is not supported in this file.

We also provide a file called `proslc_api_options.mk` – this maps the supported converter and chipset types into what `proslc_api_core.mk` understands. This may be useful for any applications on your target system.

10.9.3 *Running the Demo*

To run a demo application connect power and USB to the ProSLIC EVB, then from a Cygwin or BASH shell:

```
./<exe filename>
```

Device	Revision	DC-DC Converter	EVB Option	Executable Created (Linux drops the .exe extension)
SI3050	Any	N/A	SI3050	si3050.exe
Si3217x	B	Flyback	SI3217X_B_FB	si3217x_b_fb.exe
		BJT Buck-Boost	SI3217X_B_BB	si3217x_b_bb.exe
		PMOS Buck-Boost	SI3217X_B_PBB	si3217x_b_pbb.exe
		Low-Cost QCUK	SI3217X_B_LCQC3W ¹	si3217x_b_lcqc3w.exe
	C	Flyback	SI3217X_C_FB	si3217x_c_fb.exe
		BJT Buck-Boost	SI3217X_C_BB	si3217x_c_bb.exe
		PMOS Buck-Boost	SI3217X_C_PBB	si3217_c_pbb.exe
		Low-Cost QCUK	SI3217X_C_LCQC3W	si3217x_c_lcqc3w.exe
		Low-Cost QCUK	SI3217X_C_LCQC6W	si3217x_c_lcqc6w.exe
		Low-Cost QCUK	SI3217X_C_LCQC7P6W	si3217x_c_lcqc6w.exe

¹ For EVB's with DAA option installed, you would want to add to the makefile parameters the following:
PROSLIC_CFG_SRC=si3217x_LCQC3W_DAA_constants.c

		Low Cost Ultra-Boost	SI3217X_C_LCUB	si3217x_c_lcup.exe
		Low Cost Flyback	SI3217X_C_LCFB	si3217x_c_lcfb.exe
Si3226x	C	Flyback	SI3226X_C_FB	si3226x_c_fb.exe
		BJT Buck-Boost	SI3226X_C_BB	si3226x_c_bb.exe
		Low-Cost QCUK	SI3226X_C_LCQC3W	si3226x_c_lcqc3w.exe
		Low-Cost QCUK	SI3226X_C_LCQC7P6W	si3226x_c_lcqc7p6w.exe
		Low-Cost QCUK	SI3226X_C_LCQC6W	si3226x_c_lcqc6w.exe
		CUK	SI3226X_C_CK	si3226x_c_ck.exe
		QSS	SI3226X_C_QS	si3226x_c_qs.exe
		Low Cost Ultra-Boost	SI3226X_C_LCUB	si3226x_c_lcup.exe
		Low Cost Flyback	SI3226X_C_LCFB	si3226x_c_lcfb.exe
Si3218x	A	Low-Cost Capacitive Boost	SI3218X_A_LCCB	si3218x_a_lccb.exe
			SI3218X_A_LCCB110	si3218x_a_lccb110.exe
Si3219x	A	Low-Cost Capacitive Boost	SI3219X_A_LCCB	si3219x_a_lccb.exe
SI3228x	A	Low-Cost Capacitive Boost	SI3228X_A_LCCB ¹	si3228x_a_lccb.exe
			SI3228X_A_LCCB110 ¹	Si3228x_a_lccb110.exe

¹ For EVB's with Si32280-Si32283, you would want to add to the makefile parameters the following:
PROSLIC_CFG_SRC=si3228x_LCCB_constants.c.

Table 20. ProSLIC API Demo Build Options

10.9.4 *Lua Script Interpreter (Experimental)*

Lua is a lightweight scripting language that can be embedded in other applications. A Lua script can be used to prototype certain sequences without resorting to recompiling the API demo.

The ProSLIC API demo has a build option (LUA=1) if the development environment supports it.

NOTE: You should NOT enable LUA for Si3050 builds since it will have linker issues.

The interpreter can be accessed from the debug menu under “Execute LUA script” option. For more information on the Lua language, please visit www.lua.org. **Please note:** This is an experimental feature at this time.

The following ProSLIC API functions have been exported to the Lua script interpreter:

Function Name	Purpose	Arguments
SiVoice_WriteReg	Write to a Register	Channel number, address, value
SiVoice_ReadReg	Read a Register	Channel number, address
ProSLIC_WriteRAM	Write to a RAM location	Channel number, address, value
ProSLIC_ReadRAM	Read a RAM location	Channel number, address
ProSLIC_ToneGenSetup	Wrapper for API function of same name	Channel number, index to tone setting.
ProSLIC_ToneGenStart	Wrapper for API function of same name	Channel number
ProSLIC_ToneGenStop	Wrapper for API function of same name	Channel number
ProSLIC_RingSetup	Wrapper for API function of same name	Channel number, index to ring setting.
ProSLIC_RingStart	Wrapper for API function of same name	Channel number
ProSLIC_RingStop	Wrapper for API function of same name	Channel number

¹ The release includes a constants file for Si32284-Si32289, for other variants, you will need to create a constants file and place it in the custom folder for the demo + refer to it in the makefile command line.

ProSLIC_FSKSetup	Wrapper for API function of same name	Channel number, index to FSK setting.
ProSLIC_EnableCID	Wrapper for API function of same name	Channel number
ProSLIC_DisableCID	Wrapper for API function of same name	Channel number
ProSLIC_SetLinefeedStatus	Wrapper for API function of same name	Channel number, numerical value of line state (0 = open, etc.)
ProSLIC_AudioGainSetup	Wrapper for API function of same name	Channel number, rxgain, txgain, index to Impedance preset.
ProSLIC_Ring_N_Times	Rings phone N times, assumes interrupt enable RING_T1 has not been disabled.	Channel number, number of times to ring the channel.
si_sleep	Sleep process for N seconds.	Number of seconds to pause.

In addition, the following global variables are set when running in the interpreter:

Variable name	Purpose
PROSLIC_NUM_CHAN	Number of channels
PROSLIC_VERSION	Version of ProSLIC API

Example Lua scripts are provided under the build\lua directory. When specifying a Lua script, please include the relative directory in your input. For example: lua/wrv.lua instead of wrv.lua.

10.10 uPBX demo

The uPBX demo application is a light weight application that implements a limited PBX system. It uses the same build infrastructure as found in the ProSLIC API. It can be used with the Caller ID framework software to generate type 1 Caller ID. Please refer to the Makefile for further information.

Main features:

- Supports ALL ProSLIC chipsets that are supported by the API Demo
- DTMF & Pulse dial detection supported.
- With the Caller ID framework enabled, generate type 1 CID

It does not have:

- Support VDAA
- VoIP support

Proprietary Information - No dissemination or use without prior written permission from Silicon Labs.

- Type-2 CID support
- Support for VMB2 firmware earlier than 1.7. Please see pbx_demo.c for what to do in the case of older firmware.

10.10.1 *Configuring the uPBX demo*

In the “inc” directory, we have following files for configuring the demo:

- api_demo.h - number of devices, channels, etc.
- pbx_demo.h – timeouts for various states

10.10.2 *Building the uPBX demo*

To build a demo application, from a terminal:

```
cd <ProSLIC API Installation Root>/proslic_api/demo/uPBX/build

make <evb option>=1 [NO_LBCAL=1] <VMB1|VMB2|SPIDEV>=1 [DEBUG=1]
[PROSLIC_CFG_SRC=<your constants file>] [LOG_ALL=1] [NO_VDAA=1] [LUA=1]
[ENABLE_CID=1]
```

Where evb_option is the same options documented in the API demo in “Table 20. ProSLIC API Demo Build Options “

For example:

```
make SI3226X_C_FB=1 VMB2=1 ENABLE_CID=1
```

Please note: you will need to modify the Makefile to point to the location you have installed the CID framework.

10.10.3 *Example execution*

To start, the executable from the build directory, just type ./<exe> name. Below is an example run (this was done for a 4 channel build):

```
*****
***** micro-PBX demonstration software. Version 0.1.0 *****
*****
Copyright 2010-2017, Silicon Labs, Released under NDA
ProSLIC API: 9.0.0
```

PBXDEMO:Connecting to Host -> VMB2 ...

PBXDEMO:Initializing system timer...

PBXDEMO:Linking function pointers...

Demo: VMB2 VCP Connected via COM8

Demo: VMB2 Firmware Version 1.8

Change Default VMB2 Settings (y/n) ?? ->n

Demo: Resetting device...

PBXDEMO:Allocating memory

PBXDEMO:Initializing ProSLIC...

Si3226x_Init_with_Options(565) size = 4 init_opt = 0

PBXDEMO:Starting Longitudinal Balance Calibration...

port: 0 Number of devices: 2 Number of channels: 4

PBXDEMO:Channel 0 phone number = 5551200

PBXDEMO:Channel 1 phone number = 5551201

PBXDEMO:Channel 2 phone number = 5551202

PBXDEMO:Channel 3 phone number = 5551203

Demo is now ready to start - perform a hookflash to change settings

State changed from Idle to Dial Tone for chan 0

State changed from Dial Tone to Digit Collect for chan 0

DBG: digits dialed for chan 0 = 55

DBG: digits dialed for chan 0 = 555

DBG: digits dialed for chan 0 = 5551

DBG: digits dialed for chan 0 = 55512

DBG: digits dialed for chan 0 = 555120

DBG: digits dialed for chan 0 = 5551201

State changed from Idle to CID for chan 1

State changed from Digit Collect to Ring Back for chan 0

State changed from CID to Connect for chan 1

State changed from Ring Back to Connect for chan 0

State changed from Connect to Disconnect for chan 0

State changed from Connect to Idle for chan 1

State changed from Disconnect to Idle for chan 0

Proprietary Information - No dissemination or use without prior written permission from Silicon Labs.

Copyright © 2018 Silicon Labs

The above shows us:

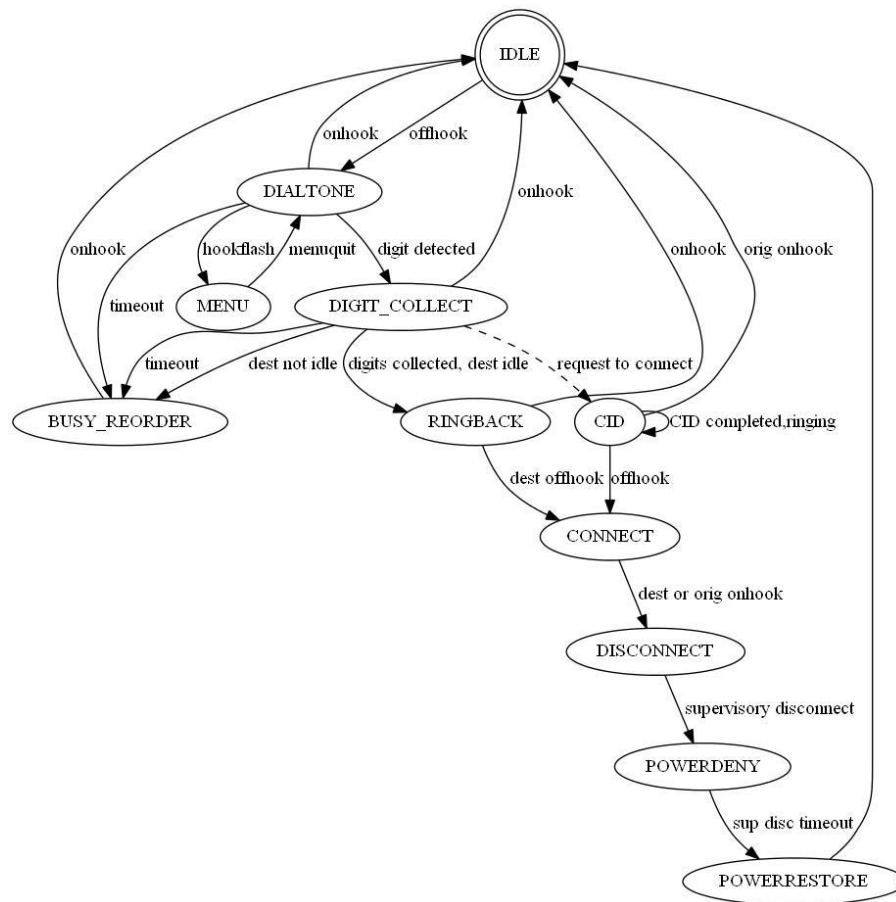
- going off hook on the first channel (0) (Idle->Digit Collect)
- dialing 5551201 (Digit Collect->Ringback)
- Channel 1 waking up to run the CID framework to send type-1 CID (Idle->CID)
- Channel 1 going offhook (CID->Connect)
- Channel 0 going from ringback to connect. At this point we have established an audio path.
- Channel 1 going back onhook (Connect->Idle)
- Channel 0 going to Supervisory Disconnect (Connect->Disconnect)
- Channel 0 going back to Idle, ready for another call...

10.10.4 Code overview

In the src directory we have the following C files:

Name	Purpose
pbx_demo.c	Main entry point. Initializes the ProSLIC API, the CID framework (if enabled), and the uPBX demo.
pbx_demo_dbg_menu.c	Debug menu when hook flash is detected. Change the phone numbers & names. Has the same low level debug menu as found in the API demo.
pbx_demo_dial_plan.c	Performs a simple string match on digits detected.
pbx_demo_exec.c	This polls for interrupts & decodes them. In addition, runs the state machine for the uPBX demo.
pbx_demo_fxs_hook.c	Hook change management.
pbx_demo_fxs_sm.c	This file has the implementation of all the different states the uPBX transitions through.

Below is a graphical representation of the various state transactions:



The uPBX demo has been tested with up to 6 channels. It should scale up to the maximum number of channels the daisy chain allows.

10.11 Debugging the Demo Applications

The provided makefile can be used to build the executable with debug symbols. Installation gdb is needed and rebuild the demo with the additional `DEBUG=1` flag set (after doing a make clean). This will build with the `-g -ggdb` flags enabled allowing one to use gdb to step through the code to see how the various API functions work together to implement the demo functions. The executable name changes to have a `_DBG` suffix added to it.

If you do not have a debugger installed, you can still use the following utility after getting a “core” file:

```
addr2line <address> -e <executable_name>
```

For example:

```
<cut>
```

Stack trace:

Frame	Function	Args
0028CB98	00401C4F	(00000000, 00000001, 00000000, 0028CBC8)
0028CBB8	00401C22	(00000000, 00000001, 0028CBE8, 00417559)
0028CBE8	00414FB1	(800102C8, 00000000, 0028CC6C, 61154D18)
0028CC88	00411F5F	(00000001, 0028CCAC, 80010100, 61008477)
0028CD28	610084DC	(00000000, 0028CD84, 61007520, 00000000)

End of stack trace

You could inspect the stack frame at the time of the crash with:

```
addr2line.exe 00401C22 -e api_demo_si3218x_a_lccb_DBG.exe  
<path>/prosllic_api/src/prosllic.c:377
```

Looking at this version of the C file, you would see that ProSLIC_Init() was called. At this point you can examine the source code to determine the cause of the crash. In this example, the first argument was a 00000000 – a NULL pointer being passed to ProSLIC_Init().