ProSLIC® API and Embedded Linux® Integration

21 January 2016

TITLE	ProSLIC®/Embedded LINUX® Integration
SECURITY	**0 – Released under NDA**

Revision History

Version	Description of changes to this document	Date
006	Corrected a few grammar errors. Added more details in section 4.	21 JAN 2016
005	Expanded information on the Linux kernel demo & added a section on h/w integration with an embedded system.	03 NOV 2015
004	Updated to include information about spidev & Linux kernel driver.	02 OCT 2015
003	Updated after additional code review/bug fixes.	16 MAR 2010
002	Minor Updates – elapsed time updated and added more details.	22 FEB 2010
001	Original	5 FEB 2010

Table of Contents

1. Introduction

The purpose of this document is to present to the user a method of integrating the ProSLIC API software into a recent Linux kernel based system. The main focus is in the "Control Wrapper" as mentioned in the ProSLIC API Specification document. This layer is broken into two main categories:

- System Timer Interfaces
- System Control Interfaces

This document does not cover the following:

- How one would integrate the ProSLIC API into an application – including configuration management, interrupt control, hook state machine, etc.
- How one implements a SPI bus master under the Linux kernel.
- How one implements a PCM bus interface.

We present two different methods of implementing a system:

- Using the Linux "spidev" interface, having a minimal footprint in the Linux kernel.
- Using the Linux SPI API inside the Linux kernel, having the ProSLIC API in the Linux kernel. This does expose at minimum the System Controller interfaces to the GPL v2 requirements since most kernel implementations have export_gpl for the SPI functions.

1.1 Assumptions Made

The following assumptions are made in this document:

- Familiarity with Linux command line
- Knowledge of C & GCC
- Knowledge of how to build a Linux Kernel and associated modules
- Use of at least Linux Kernel 2.6.16 or newer for the SPI bus framework.
- A SPI bus master has been implemented. For the spidev interface implementation, the spidev module and sysfs GPIO interface are implemented and enabled.
- Use of ProSLIC API 8.0.0 or newer
- Si3217x, Si3218x, Si3226x, Si3228x or Si3050 chipsets. Please note: the kernel code example only supports FXS functionality.
- Use of GLIBC or equivalent POSIX® library.
- Control Stream is via SPI vs. GCI. ISI should work, but the implementation will depend upon the SOC vendor's software.
- PCM bus is implemented by the user. The software assumes a valid PCLK and FSYNC have already been established.

The system used for internal testing for ProSLIC API release 8.0.0 is based upon 3.8.13 for SPIDEV and for the Linux kernel driver, 3.10.

1.2 Disclaimers

INFORMATION provided is "AS IS" and makes no warranty of any kind. Customers should review the implementation to see if the approach meets their legal understanding of the GPL and the Silicon Labs Software license agreement related to this. If this approach contradicts your organization's GPL understanding, then you may not use this approach and need to determine the best method of approaching your implementation and understanding.

The only intellectual property with the Linux kernel module approach presented here are the functions and register sets mentioned in the source code demo/linux_kerernel/proslic_sys. No ProSLIC API implemented functions are exposed in this code. Customers are allowed to GPL this source code under version 2 of the GPL or MPL v2.0 . All other functions and intellectual properties are subject to the normal license agreement with Silicon Labs. These two licenses are available at

https://www.gnu.org/licenses/old-licenses/gpl-2.0.en.html

and

https://www.mozilla.org/en-US/MPL/2.0/

The other two modules presented here do not use any export_gpl symbols and the userspace uses generic C library system calls. The exact license of the C library will depend on your system's development environment. Care should be used in selecting a C library that does not cause an infraction to the Silicon Labs Software license agreement.

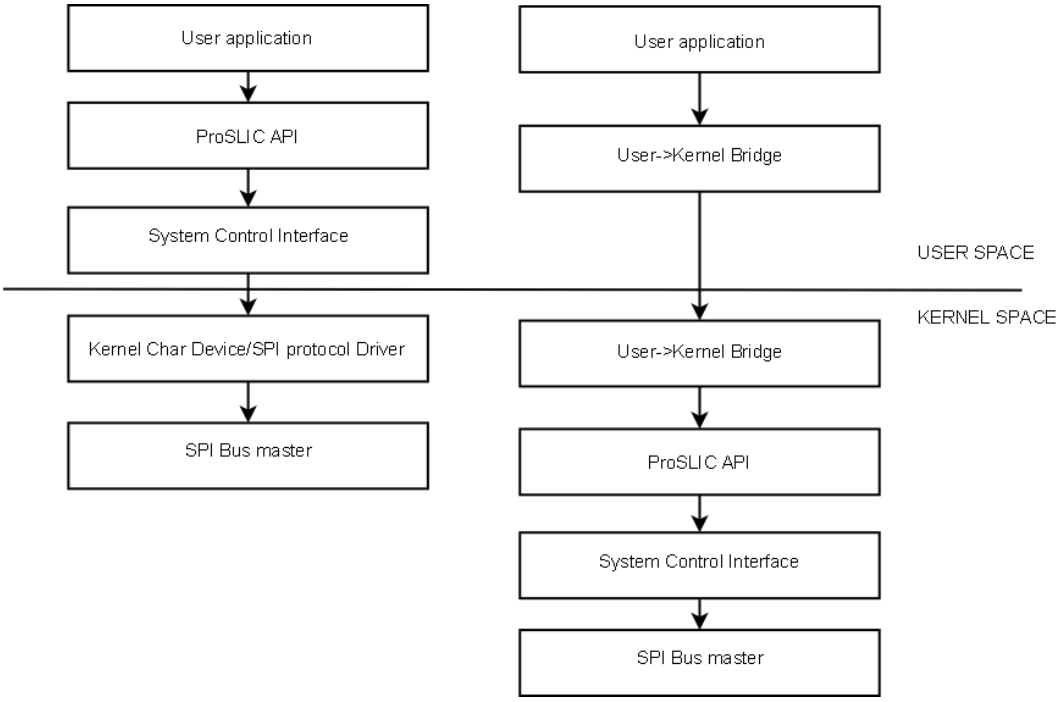
2. Spidev implementation Design Philosophy

The example “spidev” integration philosophy is:

1) Use standard API’s as much as possible – from timer management to hardware access.
2) Use the SPI framework introduced in 2.6.12 and newer. This includes the spidev character device.
3) Use standard character based interfaces – meaning use of open(), close(), read(), write() and ioctl(). In the example code provided, we also use the sysfs gpio interface for reset control. Please see: https://www.kernel.org/doc/Documentation/gpio/sysfs.txt for reference.
4) Minimize the number of system calls to communicate with the SPI devices. This philosophy drives one to send several bytes of data in one write call vs. several write calls to send few bytes of data. This is done to reduce the typical overhead between transitioning from user to kernel space function calls.
5) Do not modify the industry proven ProSLIC API, outside of the normally expected customization. The ProSLIC API assumes a synchronous API to the hardware, therefore no attempt to implement asynchronous operations is made.
6) Have the majority of the software in “userspace” vs. “kernel space”. This is done for several reasons: more accessible debug tools, memory page management, and robustness of the system as a whole, easier integration with a user’s application and avoidance of certain licensing issues.
7) In addition, having the ProSLIC in “userspace” should allow one to eliminate the variety of calls that need to be made into the kernel. That is, if one were to implement with the ProSLIC API in the kernel, a translation layer (possibly IOCTL’s) would be needed for each API that is to be called by the user application/signaling client. For example: if a simple signaling client were to be implemented, at minimum after initialization the following capabilities are needed:
    a. Ring on/off
    b. Line state control (polarity reversal, OSI, OHT, etc.).
    c. Hook state detection

    Other capabilities such as supporting provisioning (gain control, impedance settings, tone settings, FSK generation, etc.) may also be needed, depending on requirements. One may also need to take advantage of the FSK generation (Caller ID), tone generation, and other features of the ProSLIC. Each of these capabilities would require an IOCTL or similar bridge if the ProSLIC were to be placed in the kernel.

    The diagram below shows the two possible places for the ProSLIC API. The version on the left is the spidev version and the on one the right is the Linux Kernel bridge approach discussed in the next section.



In the version on the left we have the following functions required: open, close, read, write, and 1 IOCTL (depending if sysfs gpio is used or not). The poll operation is optional and will depend on if interrupt support is needed and how one implements them in the kernel. If using a gpio sysfs interface for reset, then this too would need to be implemented. For the most part these API's could be implemented with little code since the "heavy lifting" is done in the SPI bus master, needed in both implementations.

NOTE: At present the spidev example code is not thread safe – the main area is in the RAM access and in the register read – where we perform at least 2 or more accesses. If your system needs this, you would need to add the necessary locking mechanisms.

2.1 Spidev differences from VMB2 demo code…

The spidev code is implemented under the demo/platform/linux/spi directory. The timer code is implemented under demo/platform/posix/timer. The demo code is implemented assumes:

1) sysfs GPIO interface for the reset (default: gpio50)
2) The GPIO is already configured (see readme_linux_spidev.txt for details located in the build directory for each application). Configured means enabled and set for output. For systems with variable voltage families (1.8V, 3.3V, 5V, etc.), a 3.3V logic level is required for all digital signals going to a
3) The audio subsystem is configured for PCLK and FSYNC.

Various configuration options are available in the platform configuration header file "proslic_platform_cfg.h". In this file one can configure the name of the spidev character device (default: /dev/spidev1.0), the sysfs path for the reset GPIO (defaults to gpio50), the various SPI bus parameters, and the number of loop counts for the ram_wait() function.

The SPI bus parameters include:

- SPI clock rate
- Number of bits per word – 8, 16, or 32 bit. Depending on the SPI bus master implementation, changing from 8 bit to a 32 bit setting may improve data transfer rates.
- Enabling the RAM write function to send the register write requests as either 1 or 6 write() function calls. Enabling this feature may improve data transfer times due to the reduced amount of context swapping that would occur on the system.
- Byte length – number of bytes to read/write in one I/O call to the SPI bus master. This would be either 1 or 2 or 4 bytes. Increasing this parameter may improve the data transfer rates by reducing the number of I/O calls to the SPI bus master.

Customers should make changes for their particular platform to ensure proper operation.

After making the necessary parameter settings, one should run the platform validation tool as described in the ProSLIC API specification, demo section. To build the platform validation tool, type the following from your shell (under demo/pform_test/build):

```
make SPIDEV=1
```

It is highly recommended that the program be executed a few times to ensure consistency on the target system.

NOTE: The application assumes that another program has already configured the PCM bus to use the ProSLIC or DAA. Namely, PCLK and FSYNC **MUST** be up and running prior to running the executable and **MUST** be running until program termination. A failure to do so will cause the master status register read test to fail – indicating a system issue.

The platform validation tool checks that the ProSLIC device has detected a valid PCLK and FSYNC as one of its tests (master status register). If this test passes, then the minimum timing requirements for the PCM bus have been met for these two signals.

If the program is being cross-compiled, then one should set CC to point to the cross-compiler being used. For example:

```
export CC=arm-linux-gcc
```

2.1.1 Building the API Demo

Silicon Labs provides the spidev interface build option as part of the normal ProSLIC API demo. To build for a Linux spidev, one would set the normal parameters for chipset, configuration, debug enabled, MLT, etc. The only difference is to use SPIDEV=1 instead of VMB2=1. For example, for the Si3226x Rev C, Flyback design one would type from a bash shell:

```
make SI3226X_C_FB=1 SPIDEV=1
```

This would build an executable that will use the spidev interface. All the normal functionality mentioned in the ProSLIC API specification is supported.

3. Linux Kernel example software

The current ProSLIC API release provides an example Linux kernel driver. This example is located under demo/linux_kernel. This driver is broken into following components:

- System services module – implements the functions mentioned in the "Customer implemented functions". Licensed under GPLv2 and MPL. The code is located under proslic_sys directory.
- Platform test module – implements the equivalent of the platform test application but runs under kernel space. Uses the system services module.
- ProSLIC API core module – contains a simple character driver that interfaces to the ProSLIC API functions. Uses the system services module.
- Userspace demonstration program. This interfaces to the ProSLIC API core module and demonstrates how to communicate with the kernel module. This demo program is not as extensive as the spidev or VMB1 or VMB2 version at this time.

3.1 Known limitations

The example software does not interface to the standard kconfig menu based system at this time as it is typically system and Linux distribution specific. In the future we may provide an example kconfig file for the different options available.

The software has only been tested on the Si3226X chipset at this time. While it should work on other chipsets, given the underlying code is the ProSLIC API, only this chipset was tested during development.

FXS IOCTLs have been implemented. FXO IOCTLs have not been implemented for bridging to the userspace application.

Power management was left out. The implementer will need decide if the SLIC/FXO needs to support power management. Part of the tradeoff is with the static power consumed in "free run mode", allowing the part to run without a PCLK/Framesync vs. running in "low power mode", which needs PCLK/FSYNC to be externally generated.

3.2 Environment variable/build options

All the kernel modules need to have the following environment variables set:

- ARCH – which system architecture is this module being built for?
- CROSS_COMPILE – which cross compiler prefix to use?
- KDIR – location of the Linux kernel being compiled against
- PATH set to have your cross-compiler in the path.
- CC will need to be set to build the userspace demo to use your cross-compiler.
- PROSLIC_API_DIR – location of the ProSLIC API. Defaults: relative path.

You may have additional build options in each module. Please refer to the Makefile for that module for current details.

3.3 System Services module

This kernel module uses the documented SPI interface found at:

https://www.kernel.org/doc/Documentation/spi/spi-summary

The functions are normally export_gpl'd so the calling module MUST be GPLv2 or greater. As mentioned earlier, the code in this module is licensed under GPLv2 or MPL 2.0. This module does NOT refer directly to any ProSLIC API proprietary header or source file.

The configuration options are located under proslic_sys_cfg.h and include similar options found in the spidev implementation:

- BITS per word – 8, 16 or 32
- SPI clock rate – please refer to your part's datasheet for timing.
- MAX ram wait loop count
- Reset GPIO # (this can be overwritten with a device tree entry)
- Maximum number of channels to support
- RAM write block operations (vs. register access) – this may improve total system performance enabling this option.
- Default debug level – for initial development one may want to leave it enabled, but after the system services module has been tested, it should be disabled or set to give less information.
- Threshold to call mdelay() vs. msleep(). Users may want to adjust this parameter to determine when to call mdelay() and when to call msleep(). mdelay() on systems tested seems to be more accurate, but may cause "busy waits". msleep() schedules the task to sleep by the scheduler, but has more overhead in setup. Therefore msleep() is better for longer delays on system performance.
- msleep() setup overhead time (found on some systems)
- Bytes to send per spi call – 1, 2 or 4 (with exception to the RAM write block setting). It is suggested to set this to 4, if possible. This reduces the number of blocks transferred, but does increase the number of bytes sent. At 1 MHz SPI clock on the system tested, we saw an overall throughput improvement on SPI writes.

Other settings are handled by the general Linux kernel settings:

- CONFIG_OF – if device trees are supported or not

Source code assumes that the SPI bus master has already been configured. This software release has only been tested against a system with device trees and may not be complete without device trees. For those systems, a board file modification may be needed.

This example is not thread safe – RAM read/write and register reads are not protected at this time. Depending on settings (such as "block mode" for RAM write, and 4 byte register write mode), this issue can be reduced down to RAM read/write since the register read/write operators could be reduced down to 1 SPI master request instead of multiple requests.

3.3.1 Device tree Example

This example device tree configures the ProSLIC systems driver to run at 1 MHz on the first chip select (<reg>) with the SPI mode set to 3 with 2 channels.

```
proslic_spi0: proslic_spi0@0 {
        compatible = "silabs,proslic_spi";
        status = "okay";
        spi-max-frequency = <1000000>;
        reg = <0>;
        spi-cpha = <1>;
        spi-cpol = <1>;
        number_channels = <2>;
        debug_level = <7>;
      };
```

The number of channels is used by the probe() routine to see if it can communicate with the given number of ProSLIC channels. **NOTE**: A Si32178/9 chipset should set this parameter to 1 since the probe() routine does not understand how to configure the DAA side of the ProSLIC.

Please refer to your distribution on how to compile and use device trees (if supported).

3.3.2 Functions provided

The systems services exports the following symbols to be used by other modules:

- proslic_spi_if - reset, register read/write, RAM read/write, and semaphore (not implemented).
- proslic_timer_if – delay, time elapsed, get time.
- proslic_get_channel_count - number of channels detected (DAA or ProSLIC). **NOTE**: For the SI32178/9 with a DAA, this count will be 1 not 2. Should be used to see if any device was detected.
- proslic_get_device_type – was the given device a DAA or ProSLIC on a particular channel?
- proslic_get_hCtrl - for a given channel, return the handle to a hardware control (SPI) interface. This is used by the upper layer modules to get the parameter needed to pass to this module.

3.3.3 Debugging

The systems module uses the printk function to print out various messages, depending on the debug level set in either the device tree or set via command line parameter. The printk's are all prefixed with "PROSLIC_API" to help users search for this particular module's messages.

Example messages (1 Si3226x installed, configured to probe 4 channels):

```
dmesg |grep PROSLIC_API
[    8.010000] PROSLIC_API TRC: proslic_reset(418): in_reset = 0
[    8.010000] PROSLIC_API TRC: proslic_read_register_private: 0 11 0x60 0xB
[    8.010000] PROSLIC_API TRC: proslic_read_register_private(176): chan = 0 reg = 11 data = 0x05
[    8.010000] PROSLIC_API DBG: proslic_detect_type(461): channel = 0 data = 0x5
[    8.010000] PROSLIC_API TRC: proslic_write_register(223): chan = 0 reg = 12 data = 0x00
[    8.010000] PROSLIC_API TRC: proslic_read_register_private: 0 12 0x60 0xC
[    8.010000] PROSLIC_API TRC: proslic_read_register_private(176): chan = 0 reg = 12 data = 0x00
[    8.010000] PROSLIC_API TRC: proslic_write_register(223): chan = 0 reg = 12 data = 0x5A
[    8.010000] PROSLIC_API TRC: proslic_read_register_private: 0 12 0x60 0xC
[    8.010000] PROSLIC_API TRC: proslic_read_register_private(176): chan = 0 reg = 12 data = 0x5A
[    8.010000] PROSLIC_API DBG: proslic_detect_type(472): channel = 0 is_proslic
[    8.010000] PROSLIC_API TRC: proslic_read_register_private: 1 11 0x70 0xB
[    8.020000] PROSLIC_API TRC: proslic_read_register_private(176): chan = 1 reg = 11 data = 0x05
[    8.020000] PROSLIC_API DBG: proslic_detect_type(461): channel = 1 data = 0x5
[    8.020000] PROSLIC_API TRC: proslic_write_register(223): chan = 1 reg = 12 data = 0x00
[    8.020000] PROSLIC_API TRC: proslic_read_register_private: 1 12 0x70 0xC
[    8.020000] PROSLIC_API TRC: proslic_read_register_private(176): chan = 1 reg = 12 data = 0x00
[    8.020000] PROSLIC_API TRC: proslic_write_register(223): chan = 1 reg = 12 data = 0x5A
[    8.020000] PROSLIC_API TRC: proslic_read_register_private: 1 12 0x70 0xC
```

```
[    8.020000] PROSLIC_API TRC: proslic_read_register_private(176): chan = 1 reg = 12 data = 0x5A
[    8.020000] PROSLIC_API DBG: proslic_detect_type(472): channel = 1 is_proslic
[    8.020000] PROSLIC_API TRC: proslic_read_register_private: 2 11 0x68 0xB
[    8.020000] PROSLIC_API TRC: proslic_read_register_private(176): chan = 2 reg = 11 data = 0x00
[    8.020000] PROSLIC_API DBG: proslic_detect_type(461): channel = 2 data = 0x0
[    8.020000] PROSLIC_API DBG: proslic_detect_type(486): channel = 2 is_unknown
[    8.020000] PROSLIC_API TRC: proslic_read_register_private: 3 11 0x78 0xB
[    8.020000] PROSLIC_API TRC: proslic_read_register_private(176): chan = 3 reg = 11 data = 0x00
[    8.020000] PROSLIC_API DBG: proslic_detect_type(461): channel = 3 data = 0x0
[    8.020000] PROSLIC_API DBG: proslic_detect_type(486): channel = 3 is_unknown
[    8.020000] PROSLIC_API DBG: proslic_spi_setup(661): spi driver registered
[    8.020000] PROSLIC_API TRC: proslic_spi_setup(669): completed
```

This module does not provide any direct register/RAM read/write functionality since that may cause some conflict in the other modules using it.

A good initial step is to see if this module detected a ProSLIC or DAA. This would confirm the SPI bus is marginally functional. This step does not replace the platform test module mentioned in the next section.

On the system tested, one can see the sysfs entries under /sys/bus/spi/devices/spi0.0/driver/module. One can also see the parameters set and adjust the debug level from this location. **NOTE**: It is suggested that the debug level be changed to 0 once the system has been verified since it will print out messages on a per register/RAM read/write basis which could impact system behavior.

3.4 Platform test module

This module provides the equivalent of the platform test application mentioned in the ProSLIC API specification. It should be executed during initial system development to confirm that the SPI and PCM bus timing are sufficient to integrate the ProSLIC API on your platform. Once the tests pass, you can discard this module as it does not add any real value on a production system.

In order to build this module, the following symbolic links are needed:

- src – platform demo code
- custom – platform custom presets
- api_src – ProSLIC API src directory
- platform – ProSLIC API platform directory

A helper script is provided called mklinks.sh to create these symbolic links for you.

When this module loads, it will take several seconds to run and produce numerous printk messages that have a prefix of ProSLIC_CORE. Once installed, it will stay loaded until unloaded. At this time, it is suggested to unload the module once loaded as it does not provide any services to any other module.

3.4.1 Example output

Below is an example run of the test module. Note the rmmod at the end:

```
insmod proslic_pform_test.ko
dmesg |grep ProSLIC_CORE
246712.420000] ProSLIC_CORE:  ProSLIC API platform test module loaded, version: 0.0.1
[246712.420000] ProSLIC_CORE:  Copyright 2015, Silicon Laboratories
[246713.920000] ProSLIC_CORE:  spiReadRevIDTestCommon          : PASSED
[246713.920000] ProSLIC_CORE:  spiBasicWriteTestCommon         : PASSED
[246713.920000] ProSLIC_CORE:  spiMasterStatusTest             : PASSED
```

```
[246713.920000] ProSLIC_CORE:  spiBasicRAMReadTest             : PASSED
[246713.960000] ProSLIC_CORE:  spiBasicRAMWriteTest            : PASSED
[246714.460000] ProSLIC_CORE:  spiResetTestCommon              : PASSED
[246715.460000] ProSLIC_CORE:  timerElapsed_1Sec               : PASSED
[246715.470000] ProSLIC_CORE:  timerElapsed_10mSec             : PASSED
[246717.490000] ProSLIC_CORE:  timerElapsed_2015mSec           : PASSED
[246717.490000] ProSLIC_CORE:  timerElapsed_5mSec              : PASSED
[246718.330000] ProSLIC_CORE:  spiReadTimedTestCommon          : measured: 840 mSec, for 10000
runs.  0 error(s) detected.
[246718.330000] ProSLIC_CORE:  spiReadTimedTestCommon          : PASSED
[246719.010000] ProSLIC_CORE:  spiWriteTimedTestCommon         : measured: 680 mSec, for 10000
runs.
[246719.010000] ProSLIC_CORE:  spiWriteTimedTestCommon         : PASSED
rmmod proslic_pform_test
```

NOTE: It is recommended to keep the debug setting for the system services module set to the lowest possible setting as the default level prints out several messages per SPI access.

The main takeaway from the above example output is that all tests should pass prior to trying to integrate the ProSLIC API "core" module. The SPI access tests look for consistency with the data and the timing tests look for accuracy within a certain timing tolerance. On the system tested, the tests take < 7 seconds to run – this will depend on your SPI bus clock and if the tests passed or failed. The current tests stop on first failure for SPI and timing. If either test blocks fail, the timed SPI tests do not run.

3.5 ProSLIC API "core" module

This module is intended to be used in your deployed production system. It should not be instantiated until the system services module and platform test module have been run and found that the basic infrastructure seems reliable. This module assumes another subsystem has configured the PCLK, FSYNC, and audio data.

A basic character based IOCTL interface has been implemented for demonstration purposes. Customers may discard this for their specific needs. A simple makefile is provided. None of the ProSLIC API functions are marked as exported, given the specific requirements will differ from customer to customer.

Customers should review the prosclic_ioctl_api.h header file for what has been implemented to communicate with the userspace demo program. The module does all of the general initialization when open() is called – so it may take several seconds to open a file handle. At present, the implementation has 1 file handle per device chain.

The Makefile for this module needs the following symbolic links setup:

- src – points to the ProSLIC API source directory
- patch_files – points to the ProSLIC API patch file directory.

The module has its own directory for presets/config files and configuration header files.

NOTE: Audio presets and timeslot assignment has not been implemented for this module for this release. Customers would need to add this code in the open() function in the character driver C file.

3.6 ProSLIC API userspace demo

To build the demo application have the environment variable CC and PROSLIC_API_DIR set as needed. There are no build time configuration options at present – other than the number of file handles to open. The chipset specific code is all in the “core” kernel module.

This application has a text based menu to allows the customer to exercise the various IOCTL calls provided by the ProSLIC API core module.

The demo provides the following (subject to change):

1) Initializes the part with the default presets (index 0) from the given constants file in the “core” module.
2) Debug menu – read/write registers/RAM locations, create a dump file of registers and RAM.
3) Linefeed menu – read/set linefeed state, load DC feed preset
4) DC feed menu – power up/down converter
5) Ring menu – start/stop ringing, load ring preset.
6) Audio menu – load zsynth preset (impedance setting), PCM bus preset, enable/disable PCM bus, set timeslots.
7) Tone menu – load tone preset, start tone with and without cadence timers enabled, disable/stop tone.

The demo uses IOCTL’s to call functions in the character device driver to call the ProSLIC API functions. Only open(), close() and ioctl() are used currently.

4. Using the Silicon Labs EVB with an embedded system

Depending on your embedded system, you may want to have the VMB provide a starting point of migrating over to your embedded system. The VMB1 and VMB2 provide power, PCLK, FSYNC, reset, and a SPI bus controller. One strategy is to migrate the SPI interface with the reset to be driven by the embedded target and have the PCM signals come from the VMB. After that interface has been verified, have the PCM bus migrated to the target along with the power supply.

NOTE: While the VMB1 could be used for the starting point for migration, it does require a DSP firmware image to be downloaded for the PCLK & FSYNC to work correctly. This would require at minimum, the DSP image to be downloaded via the USB cable from a PC using custom software. Therefore, a VMB2 is recommended since it has some default PCLK & FSYNC values without connecting it to a PC via USB (default= 512 KHz for PCLK).

The pins on the VMB2 that need to be isolated for SPI are:

- JS12(SPI) – MOSI_OUT – use pin 1. NOTE: if you EVB has an SDI Thru header, it is not connected to the bottom pins, so connecting wires to the top header is not equivalent to connecting to the bottom pins.
- JS7(SPI) – INT, CS0, SCLK, MISO_OUT
- JS10(PCM + Reset) – PCLK_OUT, DRX_OUT, DTX_OUT, FSYNC_OUT, RST_OUT
- JS9 (Power)

Exact connector references cannot be given from the EVB perspective since these may differ.

Caution: Care should be taken to avoid damaging the lower pins since not all the pins are through hole and can be easily damaged if bending them.

If doing this in stages, do not forget to have a common ground to have correct signaling occur. Please also make sure your voltage I/O is compatible with the EVB in question. Most EVBs use 3.3V for digital I/O, not all SOCs are compatible with this. Please consult your datasheets for both the SOC and the Silicon Labs part being used.