Microsemi AcuEdge™ Software Development Kit for the Timberwolf Series Reference Guide

Document# 160191
September 2017

The intent of this guide is to provide an overview of the Microsemi AcuEdge™ Software Development Kit and how the software can be used for rapid development with the Microsemi Timberwolf devices.

Microsemi only warrants that its products, once released to production, will substantially conform to their published specifications, in accordance with Microsemi's standard sales terms and conditions. All other parameters, specifications, designs, enhancements, additions and other modifications thereto, whether to the products themselves or to any related device, module or system, are the sole responsibility of the customer, its OEMs, its subcontractors and other third parties acting on behalf of the customer. Any application support provided by Microsemi in connection with the product, including without limitation, system design recommendations and review, is provided "as is", without any warranty, representation, condition or liability whatsoever.

Contents

Revision History

Revision Number	Revision Date	Description
1	October, 23, 2017	(Initial draft)

Abbreviations

HBI	Host Bus Interface
API	Application Programmer Interface
SPI	Serial Peripheral Interface
I2C	Inter Integrated Circuit
I2S	Inter IC Sound
VPROC	Voice Processing
VPD	Voice Processing Device
SDK	Software Development Kit
IC	Integrated Circuit
OS	Operating System
ALSA	Advanced Linux Sound Architecture
SSL	System Service Layer
HAL	Hardware Abstraction Layer
RAM	Random Access Memory
DAPM	DAI Power Management
DAI	Digital Audio Interface
CS	Chip Select
MPI	Micro-Processor Interface

Typographical Conventions

The SDK guide uses the following formatting conventions

File names/paths	in italic
`C-code Functions`	in Courier New Fonts
`C-code Variables`	in Courier New Bold Fonts
`Terminal commands`	in Courier Fonts

Introduction

This document describes the Microsemi Software Development kit (SDK), identified as the ZLS38100 It is used to control Microsemi's Voice Processing Devices (VPDs). This chapter highlights the document structure and conventions and summarizes the SDK Architecture and features.

Chapter Overview

This user's guide consists of the following chapters:

- Chapter 1: ZLS38100 Software Development Kit Overview
- Chapter 2: Application Programmer Interface
- Chapter 3: Hardware Abstraction Layer
- Chapter 4: System Service Layer
- Chapter 5: Debug Functions

Frequently used Terms

- *Within this document, the terms VPROC SDK and ZLS38100 SDK or ZLS38100 Software package are used interchangeably*
- *The term ZL380xx and VPD refer to all the devices included in the Timberwolf device portfolio*

References

The following are documents you may want to refer to when using this guide.

- ZLS38100 Porting Guide
- ZLK38xx Firmware Manual

ZLS38100 Software Development Kit Overview

The VPROC SDK library is a C source code module that provides a standard software interface for controlling the Microsemi Voice Processing Devices over a Host Bus Interface through SPI or I2C, and exchanging audio with Microsemi Voice Termination Devices through PCM or I2S. The VPROC SDK hides the details of controlling Microsemi VPDs and allows software developers to focus on the application instead of the hardware.

Features

Listed below are some of the key features of the VPROC SDK library.

- Provides an abstract, uniform software interface for any combination of Microsemi VPD products.
- Can be used in any OS or non-OS environment.
- Full example Linux/Core Android implementation of device drivers.
- Fits into common driver static/dynamic modules
- Full example demo applications for controlling every aspects of the VPD
- Supports both Big and Little Endian Micro-Controllers.
- Implemented in C code that is efficient, portable, and ANSI C compliant.
- Loads VPD's firmware and configuration dynamically or statically on at runtime or boot time

Architecture

Figure 1–1 on page 3 illustrates a typical software block diagram of a system incorporating the VPROC SDK. The VPROC SDK API provides services to the Application Layer. The VPROC SDK requires the System Services Layer and Hardware Abstraction Layer to operate correctly. This document describes the interfaces between the VPROC API and those software modules implemented by the user. The following sections describe each of the blocks shown in Figure 1–1.

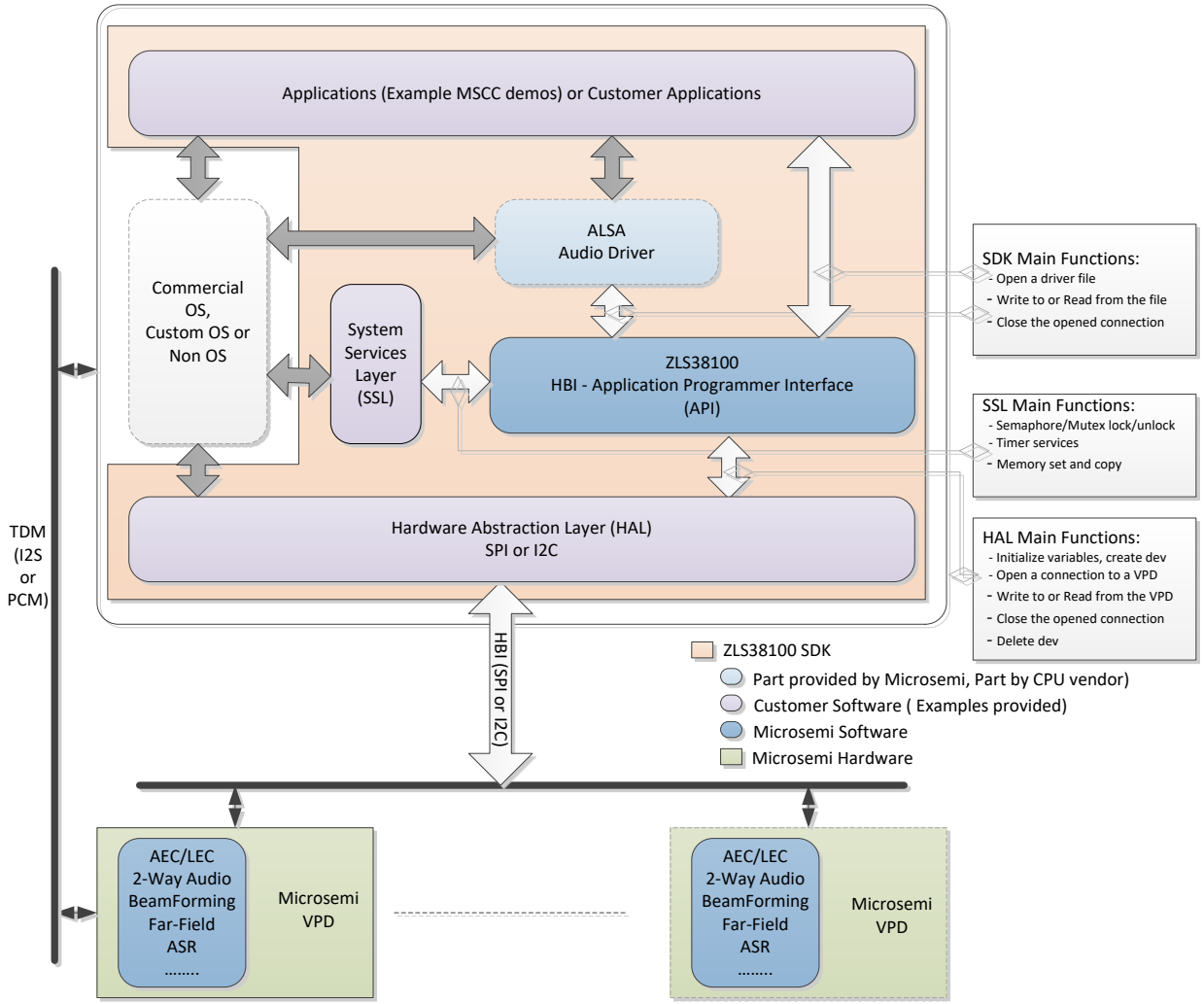

***Figure 1-1**: Software Block Diagram*

VPROC SDK HBI - API

The HBI – API is very light API consisting of only 4 main functions (OPEN, READ, WRITE, CLOSE), and 4 other extra functions that used the 4 main functions to perform more complex tasks such as send a command to the VPD to save a firmware to flash, load a firmware from flash, put the VPD to sleep, etc... The API is the core component of Microsemi's VPROC Software Development Kit (SDK). This software module runs on the host microprocessor that controls one or more Microsemi VPDs. This code is supplied by Microsemi and do not require modification by the application developer.

VPROC HBI API function Summary

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

Basic API Functions

- `HBI_open()` – Opens a connection instance to the device communication driver file associated with one particular VTD, and returns a file handle that must be used as input argument to all other functions of the SDK that wants to communicate with that VTD.

- `HBI_read()` – Performs HBI read transactions. Can read up to 128 registers of the device in a single access.
- `HBI_write()` – Performs HBI write transactions. Can write up to 128 registers of the device in a single access.
- `HBI_close()` – Closes a communication instance previously opened with HBI_open().

Complex API Functions

- `HBI_set_command()` – Executes an HBI command as defined by `hbi_cmd_t`
- `HBI_sleep()` – Puts the VPD to sleep mode
- `HBI_wake()` – Wakes the VPD up from sleep mode
- `HBI_reset()` – Resets the VPD to one of the supported reset modes

Applications

This block represents the user's custom applications that perform tasks such as initializing the system, loading a firmware and related configuration image into the VPD. Or, third party applications such as ALSA aplay, arecord, amixer and speaker-test that perform tasks such as send or receive audio samples to/from the VPD, etc. These functions may be distributed across a large and complex system, but they are shown as one block in Figure 1–1 for convenience. Microsemi provides example implementations of this layer as part of the VP-SDK.

VPROC Demo Application Summary

This section provides a brief overview of each of the demo applications provided with the SDK.

- `hbi_test` – Provides examples demonstration codes on how to use the 4 main functions of the API to access every features of the underlying VPD.
- `hbi_load_firmware` – Provides example demo code that can be used by the customer applications to load a VPD firmware and related configuration from the customer host platform into the VPD internal memory and optionally save the pair to a flash device controlled by the VPD.
- `hbi_load_grammar` – Provides example code on how to load a converted Sensory grammar file from the host internal memory into the VPD.

Operating System

This block represents the operating system (if any) that the user is running on the host microprocessor. The core HBI API of the VPROC SDK does not directly utilize any operating system resources. However, the System Services Layer and Hardware Abstraction Layer may utilize operating system facilities depending on the application.

Hardware Abstraction Layer (HAL)

The Hardware Abstraction Layer (HAL) provides access to Microsemi devices through the Micro-Controller's Host Bus Interface HBI. The HAL software is platform-dependent and must be implemented by the VPROC SDK user. However, Microsemi provides example HAL source code with the VPROC-SDK for a Linux platform that requires little to no change to port it to other Linux platforms. Refer to *Hardware Abstraction Layer*, on page xx for further details.

VPROC HAL functions Summary

This section provides a brief overview of each of the HAL prototype functions.

- `hal_init()` – Performs platform-specific initialization of the HBI and enable the device for access.
- `hal_term()` – Undoes platform-specific initialization performed by `hal_init` and disables access to the device
- `hal_open()` – Opens an instance of the device driver for access to a specific VPD
- `hal_close()` – Closes a previously opened instance
- `hal_port_rw()` – Performs HBI write or read transactions.

System Services Layer (SSL)

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

VPROC SSL functions Summary

This section provides a brief overview of each of the SSL prototype functions.

- `SSL_lock_create()` – create a custom lock by giving it specific name
- `SSL_lock()` – applies a lock
- `SSL_unlock()` – Releases a lock
- `SSL_lock_delete()` – deletes a previously created lock
- `SSL_delay()` – Implements delay in unit of milliseconds
- `SSL_memset()` – Initializes a block of memory to specific value
- `SSL_memcpy()` – Copies the content of a block of memory

Supported Microsemi VPDs

The SDK supports all of the VPDs included in the Timberwolf device portfolio. The SDK supports any number and combination of these devices. The devices in the Timberwolf series are identified in the SDK under the type TW. The devices supported by the TW device type are:

ZL38040, ZL38042, ZL38050, ZL38051, ZL38052, ZL38060, ZL38062, ZL38063, ZL38067, ZL38080, ZL38090.

Basic VPROC-SDK Data Types

Table 1–1 lists the basic data types used extensively throughout the VPROC SDK. These types are defined in the `typedes.h` header file under the platform folder and must be reviewed and if necessary modified by the customer to verify that the definitions are correct on the target platform. Many other types are defined within the VPROC SDK, but the user should never redefine any VPROC SDK types other than those in `typedes.h`

Type	Description
dev_addr_t	HBI API dependent VPD register
tw_device_id_t	Application-dependent VPD device ID, user defined type.
ssl_port_handle_t	HBI-dependent device configuration info, user defined type
ssl_lock_handle_t	SSL-dependent opened lock handle, user defined type
ssl_dev_cfg_t	Device bus interface and name
ssl_drv_cfg_t	SSL, HAL dependent configuration options, user defined type
uint8_t	8-bit unsigned integer
uint16_t	16-bit unsigned integer
uint32_t	32-bit unsigned integer

Table 1-1: *VPROC SDK user defined data types*

VPROC HBI API function Return Type

The VPROC HBI API functions return a result code indicating whether the function executed successfully, and if not, what type of error occurred. The enumeration type **hbi_status_t** is defined for this purpose. All **hbi_status_t** codes are listed in the table below.

Code	Description
HBI_STATUS_NOT_INIT	HAL Driver initialization failed.
HBI_STATUS_INVALID_ARG	One or more arguments to the function are invalid. No command is issued to the VPD.
HBI_STATUS_BAD_HANDLE	Bad device handle i.e. the handle passed to the function is not associated with an open instance of the driver file
HBI_STATUS_INTERNAL_ERR	HBI communication with the device failed.
HBI_STATUS_RESOURCE_ERR	Resources required to perform the requested function are not available.
HBI_STATUS_BAD_IMAGE	Invalid firmware image number. No compatible firmware image found in the flash.
HBI_STATUS_FLASH_FULL	Not enough space on the flash to save the VPD firmware
HBI_STATUS_NO_FLASH_PRESENT	No flash is attached to the master SPI of the VPD

HBI_STATUS_COMMAND_ERR	Function execution failed due to unspecified error.
HBI_STATUS_INCOMPAT_APP	Incompatible VPD firmware image
HBI_STATUS_SUCCESS	Function executed successfully
HBI_STATUS_INVALID_STATE	Function not supported by the driver

***Table 1-2**: HBI Error codes*

APPLICATION NOTES

The VPROC SDK is designed to allow a host application to communicate with one or any combination of the Microsemi VPDs. Every application based on the SDK must follow the following function call sequences below

1. `HBI_open()` /*repeat for as per the set desired number of instances to open for that VPD*/
2. `HBI_read()` or `HBI_write()` /*depending on the desired transaction*/
3. `HBI_close()` /*To close a no longer needed opened instance of the driver*/

Hello World Application

An example Hello world host application using the VPROC SDK. This example application assumes that the HAL and SSL layers have been implemented and ported into the platform accordingly (See the ZLS38100_Porting Guide).

This example Application writes two bytes 0x12, 0x34 into register 0x00E of the VPD. Then, reads back the data from the device.

```
#include <stdio.h>

/*VPROC SDK mandatory includes*/
#include "typedefs.h"
#include "chip.h"
#include "hbi.h"

#define VPD1_DEVICE_ID 0 /*device ID of the VPD*/

void main (void)
{
    hbi_dev_cfg_t devcfg;
    hbi_handle_t handle;
    reg_addr_t reg = 0x00E;
    user_buffer_t buf[] = {0x12, 0x34};
    hbi_status_t status = HBI_STATUS_SUCCESS;
    int i = 0;

    devcfg.deviceId = VPD1_DEVICE_ID;
    devcfg.pDevName = NULL;

    /*Open one instance of the VPD identified by VPD1_DEVICE_ID */
    status = HBI_open(&handle,&devcfg);
    if(status != HBI_STATUS_SUCCESS)
    {
         printf("HBI ERROR: %d, HBI_open() \n", status);
```

```
        return;
    }

    /*Write the content of buf into the VPD register defined in reg*/
    status  = HBI_write(handle,  reg, buf, sizeof(buf));
    if status != HBI_STATUS_SUCCESS) {
        printf("HBI ERROR: %d, HBI_write() failed\n", status);
        HBI_close(&handle);
        return;
    }

    /*Read the VPD register defined in reg and store the result in buf*/
    status  = HBI_read(handle,  reg, buf, sizeof(buf));
    if status != HBI_STATUS_SUCCESS) {
        printf("HBI ERROR: %d, HBI_read() failed\n", status);
        HBI_close(&handle);
        return;

    }

    /*Print the read results stored in buf*/
    for(i=0;i<sizeof(buf);i++)
        printf("0x%02x\t", buf[i]);

    /*Close the opened instance*/
    HBI_close(&handle);
    return;
}
```

Expected Result:

```
0x12  0x00
```

Note: *This register 0x00E of the device is a special register; it is referred to as the wink register. Whenever a 16-bit value is written to that register, if the VPD device is functioning properly, it will zero out the LSB. This register can be used as a quick way to verify the VPROC SDK and the platform.*

TECHNICAL SUPPORT

For technical support, logon to the issue tracker in the Microsemi Software Delivery System (SDS) at the following URLs:
http://sds.microsemi.com/software.php
http://sds.microsemi.com/issues.php

Application Programmer Interface

The VPROC HBI API supports the following key features:

- A single host microprocessor can control multiple VPDs as per the bus limitation of that microprocessor. The number of VPD that must be supported by the API is defined as a precompile option in the Makefile of the SDK.
- The SDK supports both single and multi-threading. The applications can open as many instances to each VPD included in that design. The number of instances is defined as a precompile option in the Makefile of the SDK.

The VPD supports very complex features that require a thorough understanding of the device in order to use these features. This complexity is handled by the API, by providing the host application the option of sending just a single command to exercise any one of such features.

All the functions supported by the API are described in this chapter.

HBI_open()

SYNTAXE

```
hbi_status_t
HBI_open(
    hbi_handle_t *pHandle,   /*Pointer to opened device instance*/
    hbi_dev_cfg_t *pDevCfg) /*Device Configuration, device Id*/
```

DESCRIPTION This function performs the task of verifying that the device exists, and then opening that instance of the driver file related to that particular VPD device ID as defined by `deviceId`.

hbi_dev_cfg_t data types:

```
typedef struct
{
    hbi_device_id_t deviceId;   /*a number from 0 to MAX_NUM_DEVS-1*/
    uint8_t  dev_addr; /*Optional device address. I2C address or SPI CS*/
    uint8_t *pDevName; /*Optional pointer to device name */
    uint8_t  bus_num;   /*Optional Host bus number */
    ssl_lock_handle_t dev_lock; /*lock to serialize HBI access */
}hbi_dev_cfg_t;
```

The driver supports multiple VPDs in a single design. Therefore, the SDK is structured so that each one of these VPDs is assigned a distinct device ID. When the host needs to access a particular device, it must first ask the driver to provide access to that Timberwolf device by passing it the device ID of that VPD.

The number of VPDs and the number of device instances that can simultaneously be opened by the SDK are defined within the Makefile of the SDK.

Notes :

1. *If this function does not return HBI_STATUS_SUCCESS, then the application must not use the device handle created by this function for any subsequent API functions calls.*
2. *No other API function must be called before invoking this function.*
3. *The pDevCfg must be initialized with the device ID instance of the VPD. The device ID must be of type* ***hbi_device_id_t***

POSSIBLE RETURNS

```
HBI_STATUS_NOT_INIT
HBI_STATUS_INVALID_ARG
HBI_STATUS_RESOURCE_ERR
HBI_STATUS_INTERNAL_ERR
HBI_STATUS_SUCCESS
```

VPD TYPES TW

HBI_read()

SYNTAXE

```
hbi_status_t
HBI_read(
    hbi_handle_t handle,/*handle to opened device instance*/
    reg_addr_t reg,      /*VPD 16-bit register to start reading from*/
    user_buffer_t *pOutput, /*Pointer to stored results buffer*/
    size_t length) /*The number of bytes to read from the VPD */
```

DESCRIPTION This function reads the number of requested bytes as specified by length starting from the VPD register defined by **`reg_addr_t`**.

The `handle` argument should be a reference to the handle returned by the HBI_open call. The handle is a reference to how to access the device.

```
typedef uint32_t  hbi_handle_t;
```

The `reg` argument should specify the VPD register in unsigned 16-bit integer.

```
typedef uint16_t reg_addr_t;
```

The `pOutput` argument should point to the user application allocated buffer to a size of at least as specified by length to where to store the results in bytes read from the VPD.

```
typedef unsigned char user_buffer_t;
```

The `length` argument should specify the number of bytes to read from VPD register.

Notes :

1. *The handle passed to this function must be from a currently opened instance of the driver.*
2. *The buffer to where to store the result must be pre-allocated by the user application to a size of at least `length` number of bytes.*

POSSIBLE RETURNS

```
HBI_STATUS_NOT_INIT
HBI_STATUS_INVALID_ARG
HBI_STATUS_INTERNAL_ERR
HBI_STATUS_SUCCESS
```

VPD TYPES TW

HBI_write()

SYNTAXE

```
hbi_status_t
HBI_write(
    hbi_handle_t handle,/*handle to opened device instance*/
    reg_addr_t reg,      /*VPD 16-bit register to start writing to*/
    user_buffer_t *pInput, /*Pointer to the data buffer to write*/
    size_t length) /*The number of bytes to write to the VPD */
```

DESCRIPTION This function writes the number of requested bytes as specified by length starting from the VPD register defined by **reg_addr_t**.

The `handle` argument should be a reference to the handle returned by the HBI_open call. The handle is a reference to how to access the device.

```
typedef uint32_t  hbi_handle_t;
```

The `reg` argument should specify the VPD register in unsigned 16-bit integer.

```
typedef uint16_t reg_addr_t;
```

The `pInput` argument should point to the user application allocated buffer that has the number of data bytes as specified by length to write into the VPD register.

```
typedef unsigned char user_buffer_t;
```

The `length` argument should specify the number of bytes to read from VPD register.

Notes :

1. *The handle passed to this function must be from a currently opened instance of the driver.*
2. *The buffer to where the data to send to the VPD is stored in the host must be pre-allocated by the user application to a size of at least `length` number of bytes, and must be initialized with the actual data bytes.*

POSSIBLE RETURNS

```
HBI_STATUS_NOT_INIT
HBI_STATUS_INVALID_ARG
HBI_STATUS_INTERNAL_ERR
HBI_STATUS_SUCCESS
```

VPD TYPES TW

HBI_close()

SYNTAXE

```
hbi_status_t
HBI_close(
    hbi_handle_t handle)/*Opened handle to device instance to close*/
```

DESCRIPTION This function closes an instance of the driver previously opened by HBI_open(). This function must be called only if there are no other API functions currently using the handle.

The `handle` argument should be a reference to the handle returned by the HBI_open call. The handle is a reference to how to access the device.

Notes :

1. *No API function calls must be called with this instance of the handle once this function is called.*

POSSIBLE RETURNS

```
HBI_STATUS_NOT_INIT
HBI_STATUS_INVALID_ARG
HBI_STATUS_INTERNAL_ERR
HBI_STATUS_SUCCESS
```

VPD TYPES TW

HBI_set_command()

SYNTAXE

```
hbi_status_t
HBI_set_command(
    hbi_handle_t handle,/*handle to opened device instance*/
    hbi_cmd_t cmd)      /*VPD 16-bit register to start reading from*/
    void *pcmdArgs)     /*Pointer to data to pass the API*/
```

DESCRIPTION This function sends a command to VPD.

The `handle` argument should be a reference to the handle returned by the `HBI_open` call. The handle is a reference to how to access the device.

```
typedef uint32_t  hbi_handle_t;
```

Of the 4 main API basic functions described above or mentioned in the Overview chapter are created more complex API functions. These functions perform tasks such as the ones described in the **`hbi_cmd_t`** type definition below.

Some of these commands are defined as conditionally compiled options. For example, if the host system doesn't have a flash device interfaced to the VPD master SPI port, then the flash related commands can be disabled by de-selecting the respective directive at compile time. (See the ZLS38100 Porting Guide)

```
typedef enum
{
    HBI_CMD_LOAD_FWR_FROM_HOST,
    HBI_CMD_LOAD_CFGREC_FROM_HOST,
    HBI_CMD_LOAD_FWR_COMPLETE,
    HBI_CMD_LOAD_FWRCFG_FROM_FLASH,
    HBI_CMD_SAVE_FWRCFG_TO_FLASH,
    HBI_CMD_ERASE_WHOLE_FLASH,
    HBI_CMD_ERASE_FWRCFG_FROM_FLASH,
    HBI_CMD_START_FWR,
    HBI_CMD_SAVE_CFG_TO_FLASH,
    HBI_CMD_END
}hbi_cmd_t;
```

The table below summarizes a list of commands that are supported by the VPD and their respective descriptions. Some of the commands may require additional input for successful execution. Please see Timberwolf device firmware manual for required input w.r.t commands listed below.

Command	Description
HBI_CMD_LOAD_FWRCFG_FROM_FLASH	Instructs the VPD to load a specified firmware and associated configuration record from the flash, into the VPD internal memory. `pcmdArgs` must point to an unsigned 16-bit value that specifies the index of the image on flash to load.
HBI_CMD_SAVE_FWRCFG_TO_FLASH	Instructs the VPD to save the currently loaded (not running yet) firmware and configuration record to flash. `pcmdArgs` must point to NULL.
HBI_CMD_SAVE_CFG_TO_FLASH	Instructs the VPD to save the currently running firmware and configuration record to a specified firmware currently stored on the flash. `pcmdArgs` must point to an unsigned 16-bit value that specifies the index of the image on flash for which to save the configuration record.
HBI_CMD_ERASE_WHOLE_FLASH	Instructs the VPD firmware to erase the entire content of the flash. `pcmdArgs` must point to NULL
HBI_CMD_ERASE_FWRCFG_FROM_FLASH	Instructs the VPD to erase a specific firmware and associated configuration record from the flash. `pcmdArgs` must point to an unsigned 16-bit value that specifies the index of the image on flash to erase.
HBI_CMD_LOAD_FWR_FROM_HOST	Instructs the **HBI driver** to load a firmware image from the host to the VPD internal memory. `pcmdArgs` must point to NULL
HBI_CMD_START_FWR	Instructs the VPD to start the execution of a firmware previously loaded into its internal memory. `pcmdArgs` must point to NULL
HBI_CMD_LOAD_CFGREC_FROM_HOST	Instructs the **HBI driver** to load a configuration record from the host to the VPD internal memory. `pcmdArgs` must point to an initialized instance of the structure `hbi_data_t`

***Table 3-1**: HBI Commands*

The `pcmdArgs` argument must be in accordance to the command. If the command does not require any argument, then `pcmdArgs` must point to NULL. Otherwise, `pcmdArgs` must point to an unsigned 16-bit integer value.

The hbi_data_t structure

```
typedef struct
{
   unsigned char *pData; /* pointer to user data buffer */
   size_t        size; /*length of the buffer in bytes */
}hbi_data_t;
```

Notes :

1. *The handle passed to this function must be from a currently opened instance of the driver.*
2. *See the `hbi_load_firmware` user application for the example use of these commands.*

POSSIBLE RETURNS

```
HBI_STATUS_NOT_INIT
HBI_STATUS_INVALID_ARG
HBI_STATUS_INTERNAL_ERR
HBI_STATUS_SUCCESS
HBI_STATUS_NO_FLASH_PRESENT
HBI_STATUS_RESOURCE_ERR
HBI_STATUS_FLASH_FULL
HBI_STATUS_BAD_IMAGE
HBI_STATUS_COMMAND_ERR
HBI_STATUS_INCOMPAT_APP
```

VPD TYPES TW

HBI_sleep()

SYNTAXE **hbi_status_t**

HBI_sleep(

hbi_handle_t handle)/*Opened handle to device instance*/

DESCRIPTION This function puts the VPD referenced by handle in to sleep mode.

The `handle` argument should be a reference to the handle returned by the `HBI_open` call. The handle is a reference to how to access the device.

Notes :

1. *No API function calls other than `HBI_wake()` must be called to get the device out of that state once the device entered the sleep mode.*

POSSIBLE RETURNS
```
HBI_STATUS_NOT_INIT
HBI_STATUS_INVALID_ARG
HBI_STATUS_INTERNAL_ERR
HBI_STATUS_SUCCESS
```
VPD TYPES TW

HBI_wake()

SYNTAXE **hbi_status_t**

HBI_sleep(

hbi_handle_t handle)/*Opened handle to device instance*/

DESCRIPTION This function wakes up the VPD referenced by handle from sleep mode.

The `handle` argument should be a reference to the handle returned by the HBI_open call. The handle is a reference to how to access the device.

POSSIBLE RETURNS
```
HBI_STATUS_NOT_INIT
HBI_STATUS_INVALID_ARG
HBI_STATUS_INTERNAL_ERR
HBI_STATUS_SUCCESS
```
VPD TYPES TW

HBI_reset()

SYNTAXE

```
hbi_status_t
HBI_reset(
    hbi_handle_t handle, /*Opened handle to device instance*/
    hbi_rst_mode_t mode) /*Reset mode*/
```

DESCRIPTION This function resets the VPD referenced by handle to one of the reset modes defined by the `ZL380xx_RST_MODE` enumeration below.

The `handle` argument should be a reference to the handle returned by the `HBI_open` call. The handle is a reference to how to access the device.

```
typedef enum
{
      RST_HARDWARE_RAM, /*Hard Reset the device*/
      RST_HARDWARE_ROM, /*Reset the VPD, and load firmware from flash*/
      RST_SOFTWARE,     /*Issues a soft-reset to the VPD*/
      RST_AEC,          /*Resets the AEC*/
      RST_TO_BOOT       /*Puts the VPD in boot mode*/
}ZL380xx_RST_MODE;
```

Notes :

1. `RST_HARDWARE_ROM` reset mode performs a hard reset of the VPD, then loads the first firmware it found in the slave flash attached to the VPD master SPI port.
2. `RST_HARDWARE_RAM` reset mode performs a hard reset of the VPD; no firmware is loaded into the device.

POSSIBLE RETURNS

```
HBI_STATUS_NOT_INIT
HBI_STATUS_INVALID_ARG
HBI_STATUS_INTERNAL_ERR
HBI_STATUS_SUCCESS
```

VPD TYPES TW

Hardware Abstraction Layer

The Hardware Abstraction Layer (HAL) defines functions for communicating with a target VPD through the HBI. These functions hide the details of the platform HBI hardware design from the VPROC HBI API. The customer must implement these functions as appropriate for their specific platform. Microsemi provides example implementations of these functions for a Linux core Android based platform. The provided examples implementation requires minimal change if any to port it to a Linux platform. The following functions are included in the HAL:

- `hal_init()` – Performs platform-specific initialization of the HBI and enable the device for access.
- `hal_term()` – Releases resources and disables access to the device
- `hal_open()` – Opens an instance of the device driver for access to a specific VPD
- `hal_close()` – Closes a previously opened instance
- `hal_port_rw()` – Performs HBI write or read transactions

The HAL requires an initialized instance of the **`ssl_dev_info_t`** driver info structure. This structure defines the actual device type, and necessary info needed by the HBI in order to create the device driver file and to properly initialize the device at boot time. If multiple VPD needs to be supported, then the structure must be defined as an array, where the array length must be as per the desired number of Timberwolf devices the driver must support. The **`hbi_device_id_t`** `deviceId` argument that identifies the target VPD, is the index of that array.

Below is an example definition of that structure to register the SPI driver for two Timberwolf slave devices. One is a ZL38063 at SPI bus 0 and chip select 0, and the other a ZL38042 at SPI bus 0 and chip select 1 is provided below

```
static ssl_dev_info_t sdk_board_devices_info[] =
{
    {
        .chip = 38063,     /*Microsemi chip number without the ZL: Ex 38063*/
        .bus_num = 0,      /*SPI or I2C bus number*/
        .dev_addr = 0,     /*SPI chip select or I2C address*/
        .isboot = FALSE,   /*set this TRUE if a device firmware has to be loaded at boot*/
        .pFirmware = NULL,/*a pointer to either the filename without the extension (.bin)
                            if in *.bin format or data array  if in c code format*/
        .pConfig = NULL,   /*a pointer to either the filename if in *.bin format or data
                            array  if in c code format*/
        .dev_lock = 0,     /*lock to serialise device access */
        .imageType = 0,    /*0: for static *.h, 1: for *.bin */
    } ,
    {
        .chip = 38042,
        .bus_num = 0,
        .dev_addr = 1,
        .isboot = FALSE,
        .pFirmware = NULL,
        .pConfig = NULL,
        .dev_lock = 0,
        .imageType = 0,
    }
```

```
};
```

hal_init()

SYNTAXE

```
int
hal_init(
    void)/*no argument*/
```

DESCRIPTION This function prepares the system for communication through the HBI bus as per the configured HBI precompile option. This function takes no argument. This function is not visible at the user application layer, it is used internally by the HBI the first time the application layer issues an HBI_open() call.

POSSIBLE RETURNS This function returns `0` on success or a negative error code if the function fails to create and open that device driver instance.

VPD TYPES All

hal_open()

SYNTAXE

```
int
hal_open(
    void **pHandle,   /*Pointer to the opened device instance*/
    void *pDevCfg)    /*Device Configuration, device Id*/
```

DESCRIPTION This function creates an instance in the existing device file as per the specified host bus interface SPI or I2C, and then opens that instance of the driver file related to that particular VPD device ID as defined by `deviceId`. The data pointed to by `pDevCfg` is an initialized instance of that **`hbi_dev_cfg_t`** as passed by the `HBI_open()` call from the user application, then type casted to `ssl_dev_cfg_t`.

The SDK supports reentrancy. Multiple instances, threads or processes can each open a connection to the same VPD up to the instance per VPD (See the `HBI_MAX_INST_PER_DEVICE` definition in the ZLS38100 Porting Guide) **limit defined in the SDK. The reentrancy of the SDK is possible by the implementation of the `hal_open`. Every time the user application calls this function it will open an instance of the driver for that particular VPD.**

`ssl_dev_cfg_t` data types:

```
typedef struct
{
    tw_device_id_t deviceId;   /*a number from 0 to MAX_NUM_DEVS-1*/
    dev_addr_t dev_addr; /*device address. I2C address or SPI CS*/
    uint8_t *pDevName;    /* pointer to device name */
    uint8_t  bus_num;     /* Host bus number */
}ssl_dev_cfg_t;
```

Notes :

1. *If this function does not return 0, then the application must not use the device handle created by this function for any subsequent API functions calls.*
2. *Other API functions such read write and set command should work after this function is successfully executed.*
3. *The pDevCfg must be initialized with the device ID instance of the VPD. The device ID must be of type* ***tw_device_id_t***

POSSIBLE RETURNS This function returns `0` on success or a negative error code if the function fails to create and open that device driver instance.

VPD TYPES All

hal_port_rw()

SYNTAXE

```
int
hal_port_rw(
    void *pHandle, /*Pointer to opened device instance*/
    void *pDevCfg) /*Device access type data*/
```

DESCRIPTION This function writes a command to a VPD pointed to by pHandle, and depending on the type of the transaction either writes a specified number of data bytes to the VPD or reads a specified number of bytes from the VPD.

The pHandle pointer is a reference to the device to access. It is of the same type as `hbi_handle_t` type

. The data pointed to by `pDevCfg` is an initialized instance of the **`ssl_port_access_t`**

`ssl_port_access_t` data types:

```
typedef struct
{
    void    *pSrc; /* Pointer to source buffer of the data to write */
    void    *pDst; /* Pointer to destination buffer to store read results */
    size_t   nread;/* Number of bytes to read.  Ignored for write operation*/
    size_t   nwrite; /* Number of bytes to write. Ignore for read operation*/
    ssl_op_t op_type;/* Enum: port operation: 'read','write','read/write'*/
}ssl_port_access_t;
```

The port operation is specified by the enum **`ssl_op_t`** data type. This **`enum`** is defined below.

```
typedef enum
{
    SSL_OP_PORT_RD=0x01, /*Read only operation on Port */
    SSL_OP_PORT_WR=0x02, /*Write only operation on Port */
```

```
    SSL_OP_PORT_RW = (SSL_OP_PORT_RD | SSL_OP_PORT_WR) /*both*/
}ssl_op_t;
```

Notes :

1. *pSrc must never point to NULL for this function*
2. *The pDevCfg must be pointed to an initialized instance of* **ssl_port_access_t** type

POSSIBLE RETURNS This function returns 0 on success or a negative error code if the function fails to create and open that device driver instance.

VPD TYPES TW

hal_close()

SYNTAXE

```
int
hal_close(
    void *pHandle)/*pointer to opened device handle*/
```

DESCRIPTION This function closes an instance of the driver previously opened by hal_open(). This function must be called only if there are no other HAL functions currently using the handle.

The pHandle pointer is a reference to the device to access. It is of the same type as hbi_handle_t type.

POSSIBLE RETURNS This function returns 0 on success or a negative error code if the function fails to create and open that device driver instance.

VPD TYPES All

hal_term()

SYNTAXE

```
int
hal_term(
    void)/*no argument*/
```

DESCRIPTION This function releases system resources that were allocated during the hal_init(). This function is not visible by the user application, it is invoked internally by hal_close() when the user application issues an HBI_close() function call.

POSSIBLE RETURNS This function returns 0 on success or a negative error code if the function fails to create and open that device driver instance.

VPD TYPES All

System Service Layer

The System Services layer provides critical section, commonly known as mutual exclusion. These functions are system-dependent and must be implemented specifically for each platform on which the VPROC HBI API is used. The SSL functions below are included in the System Services layer.

- `SSL_lock_create()` – create a custom lock by giving it specific name
- `SSL_lock()` – applies a lock
- `SSL_unlock()` – Releases a lock
- `SSL_lock_delete()` – deletes a previously created lock
- `SSL_delay()` – Implements delay in unit of milliseconds
- `SSL_memset()` – Initializes a block of memory to specific value
- `SSL_memcpy()` – Copies the content of a block of memory

As mentioned in the HAL layer function description, the SDK supports not only multiple VPDs in a single design, also supports reentrancy. Therefore, since the HBI of the VPD can only be accessed by one application or instance, thread of an application at a time, then the SDK must provide the mean to enforce mutual exclusion. The mutual exclusion puts a lock on the HBI to prevent other instances of the application from accessing the HBI while there is an HBI transaction in progress. All other transactions must be queued and executed one at a time once the HBI lock is removed.

SSL Status Return Types

The VPROC SSL functions return a result code indicating whether the function executed successfully, and if not, what type of error occurred. The enumeration type **`ssl_status_t`** is defined for this purpose. All **`ssl_status_t`** codes are listed in the table below.

.

Status	Description
SSL_STATUS_NOT_INIT	Indicates the SSL_init() was not executed
SSL_STATUS_INTERNAL_ERR	Indicates that the HBI API calling functions resulted in an error.
SSL_STATUS_INVALID_ARG	Indicates some or all of the input parameters are invalid
SSL_STATUS_BAD_HANDLE	Indicates that the lock ID passed to the call is invalid.
SSL_STATUS_RESOURCE_ERR	Indicates that the API fails to acquire needed resources
SSL_STATUS_OK	Indicates that the function executed successfully.
SSL_STATUS_TIMEOUT	Indicates that the SSL_lock() timed out waiting to acquire a lock.
SSL_STATUS_FAILED	Indicates that the function failed to execute properly.

***Figure 4-1**: SSL Error Codes*

The SSL Function descriptions:

ssl_lock_create()

SYNTAXE

```
ssl_status_t
ssl_lock_create(
    ssl_lock_handle_t *pLock,   /*Pointer to the created lock*/
    const char *pName, /* Optional name for the lock or NULL*/
    void * pOptions)/*Optional pointer to lock info*/
```

DESCRIPTION This function creates a lock for synchronization purpose. User can implement it as a binary, or a semaphore, or a mutex. Any information required to create a semaphore or a mutex for the critical section lock can be passed into a data structure pointed by `pOptions`.

The created lock `pLock` **points to a** `ssl_lock_handle_t` data type. This data type is a user defined type, therefore must be defined in accordance to the user implementation of this layer.

Microsemi provides an example implementation of this layer. The pLock points to the type casted value of the structure `ssl_lock` **defined below.**

```
struct ssl_lock {

      uint8_t  name[SSL_LOCK_NAME_SIZE]; /* name of the lock */
      struct mutex lock; /* lock type */
      bool    inuse; /* flag indicating current usage*/
};
```

Notes :

1. *If this function does not return SSL_STATUS_OK, then the application must not use the lock created by this function for any subsequent SSL functions calls.*
2. *No other SSL function must be called before invoking this function.*

POSSIBLE RETURNS

```
SSL_STATUS_OK
SSL _STATUS_INVALID_ARG
SSL _STATUS_RESOURCE_ERR
SSL _STATUS_INTERNAL_ERR
```

VPD TYPES TW

SSL_lock()

SYNTAXE

```
ssl_status_t
ssl_lock(
```

```
    ssl_lock_handle_t lock_id,   /*Id of the lock*/
    ssl_wait_t wait_type)  /*Wait type for in use lock*/
```

DESCRIPTION This function protects the SDK from reentrant execution. It acquires a lock whenever it is called. This function must be implemented to prevent other thread from accessing while an HBI transaction is in progress. In the example implementation provided by Microsemi, if the lock is in use, this call would behave according to the **ssl_wait_t** parameter. If the **ssl_wait_t** is not equal to SSL_WAIT_NONE and SSL_WAIT_FOREVER, then this function will return an SSL_STATUS_INVALID_ARG error. The implementation of the **ssl_wait_t** parameter is optional and dependent upon the lock type being implemented.

The lock_id is created by a successful SSL_lock_create() call.

The **ssl_wait_t** is defined below.

```
typedef enum
{
    SSL_WAIT_NONE, /* Return immediately, if failed to get lock*/
    SSL_WAIT_FOREVER /* Wait until lock is attained. Note this
                     * may block the call
                     */
}ssl_wait_t;
```

POSSIBLE RETURNS

```
SSL_STATUS_OK
SSL _STATUS_INVALID_ARG
SSL _STATUS_INTERNAL_ERR
SSL _STATUS_BAD_HANDLE
SSL _STATUS_STATUS_FAILED
```

VPD TYPES TW

SSL_unlock()

SYNTAXE

```
ssl_status_t
ssl_unlock(
    ssl_lock_handle_t lock_id)   /*Lock Id*/
```

DESCRIPTION This function releases a previously acquired lock. All locked resources must be restored to the same state prior to the last application of the lock.

The lock_id created by a successful SSL_lock_create() call.

POSSIBLE RETURNS

```
SSL_STATUS_OK
SSL _STATUS_INVALID_ARG
```

```
SSL _STATUS_INTERNAL_ERR
SSL _STATUS_BAD_HANDLE
SSL _STATUS_STATUS_FAILED
```

VPD TYPES TW

SSL_lock_delete()

SYNTAXE

```
ssl_status_t
ssl_lock_delete(
    ssl_lock_handle_t lock_id)   /*Lock Id*/
```

DESCRIPTION This function deletes a previously created lock.

The `lock_id` created by a successful `SSL_lock_create()` call.

POSSIBLE RETURNS

```
SSL_STATUS_OK
SSL _STATUS_INVALID_ARG
SSL _STATUS_INTERNAL_ERR
SSL _STATUS_BAD_HANDLE
SSL _STATUS_STATUS_FAILED
```

VPD TYPES TW

SSL_delay()

SYNTAXE

```
ssl_status_t
Ssl_delay(
    uint32_t tmsec)   /*Delay time in ms*/
```

DESCRIPTION The HBI API calls this function when it needs to wait for certain amount of time. The `tmsec` argument specifies the wait time in terms of milli-seconds.

The `tmsec` is the desired amount of delay in milli-seconds.

POSSIBLE RETURNS `SSL_STATUS_OK`

VPD TYPES TW

SSL_memcpy ()

SYNTAXE

```
ssl_status_t
Ssl_memcpy (
```

```
    void *pDst, /*Pointer to the memory location to copy into*/
    const void *pSrc, /*Pointer to the memory location to copy from*/
    size_t size)  /*The size of the data to be copied*/
```

DESCRIPTION This function copies a number of bytes specified by size from the memory location pointed to pSrc to the memory location pointed to pDst. The implementation of this function can be a simple call to the standard C equivalent memory copy function, or else implements this function accordingly

The `pDst` points to the destination array where the content is to be copied. `pDst` is type-casted to a pointer of type `void`*.

The `pSrc` points to the source of data to be copied. `pSrc` is type-casted to a pointer of type `void`*.

`size` is the number of bytes to be copied.

POSSIBLE RETURNS

```
SSL_STATUS_OK
SSL _STATUS_INVALID_ARG
SSL _STATUS_INTERNAL_ERR
```

VPD TYPES TW

SSL_memset ()

SYNTAXE

```
ssl_status_t
Ssl_memset (
    void *pDst, /*Pointer to the location of the data to set to val*/
    int32_t val,/*the value to be written into pDst*/
    size_t size)/*The size of the data to initialize*/
```

DESCRIPTION This function initialize a number of bytes specified by size from the memory location pointed to pDst to the value in val. The implementation of this function can be a simple call to the standard C equivalent memory initialization function, or else implements this function accordingly

The `pDst` points to the destination array where the content is to be initialized with the value in `val`.

The `val` is the value with which to overwrite the content pointed by `pDst`.

`size` is the number of bytes to overwrite.

POSSIBLE RETURNS

```
SSL_STATUS_OK
SSL _STATUS_INVALID_ARG
SSL _STATUS_INTERNAL_ERR
```

VPD TYPES TW

Debug Functions

The VPROC HBI API source code includes many different types of debug output which customers can use to isolate problems with their application and/or system. Each type of output can be included/excluded at compile time.

TYPES OF DEBUG OUTPUT

The following types of debug output are supported:

Macro	Bitmask	Description
VPROC_DBG_LVL_NONE	0x00	Disables logging. No Debug message will be reported.
VPROC_DBG_LVL_FUNC	0x01	Prints out every function entry and exit.
VPROC_DBG_LVL_INFO	0x02	Prints out informational messages, such as status information during normal operation of complex function, data to be written and the data read, etc.
VPROC_DBG_LVL_WARN	0x04	Prints out behavior that is not probably what not was intended.
VPROC_DBG_LVL_ERR	0x08	Prints out a message whenever the API function call does not return an HBI_STATUS_SUCCESS.
VPROC_DBG_LVL_ALL	0x0F	Enable all of the above debug macros.

***Table 5-1**: Debug Levels*

The debug is defined as an enum. Below is the definition of this enum.

```
typedef enum
{
    VPROC_DBG_LVL_NONE=0x0,
    VPROC_DBG_LVL_FUNC=0x1,
    VPROC_DBG_LVL_INFO=0x2,
    VPROC_DBG_LVL_WARN=0x4,
    VPROC_DBG_LVL_ERR=0x8,

VPROC_DBG_LVL_ALL=(VPROC_DBG_LVL_FUNC|VPROC_DBG_LVL_INFO|VPROC_DBG_LVL_WARN|
VPROC_DBG_LVL_ERR)
}VPROC_DBG_LVL;
```

DEBUG OUTPUT SELECTION AT COMPILE TIME

Debug output strings, and the code necessary for displaying them, can occupy a non-negligible amount of memory in some applications. Therefore, the SDK provides the ability to exclude unwanted types of debug output at compile time.

Customers may want to compile in all types of debug output during initial development, but include less debug output in the final application. Or, if separate "production" and "debug" builds are maintained, a different selection of debug output may be specified in each.

The selection of debug output types at compile time is specified in the Makefile.globals file in the root folder of the SDK by the makefile options defined below. By default the debug level is set to a bit mask of 0x8 which equals to `VPROC_DBG_LVL_ERR`

```
DEBUG_LEVEL=8
```

VPROC_DBG_PRINT

The VPROC SDK routes all debug output through a customer-defined function: VPROC_DBG_PRINT. Example definition of this function is defined within the RELEASE_*ZLS38100_Px_y_z/platform/PlatformName/include/vproc_dbg.h*,

The example following definition is sufficient:

```
#define VPROC_DBG_SET_LVL(dbg_lvl) (vproc_dbg_lvl = dbg_lvl)

#define VPROC_DBG_PRINT(level,msg,args...)  if(level & vproc_dbg_lvl)
{printk("[%s:%d]"msg,__FUNCTION__,__LINE__,##args);}

#else

#define VPROC_DBG_PRINT(level,msg,args...)

#endif
```