

---

# **Microsemi AcuEdge™ Software Development Kit for the Timberwolf Series User Guide**

## **Document# PD-000275617 September 2018**

---

The intent of this guide is to provide the steps to porting the Microsemi AcuEdge™ Software Development Kit to a Host Platform.

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 .....	iv
Abbreviations .....	iv
Typographical Conventions.....	v
Introduction .....	1
Other References .....	1
ZLS38100 Software Development Kit Contents .....	2
Software Design Flow .....	3
The SDK main components in brief.....	6
Platform independent Layer .....	7
Platform dependent Layer .....	7
First Steps.....	9
SDK Basic Functions .....	9
OPEN .....	9
WRITE.....	9
READ.....	10
CLOSE .....	10
Linux Platform setup requirements: .....	10
Host Platform naming .....	12
Timberwolf devices .....	12
User Application Instances.....	13
Compile/Development environment computer .....	14
Windows host Computer minimum Requirements .....	14
Compile/Development platform Network Architecture.....	14
Host Windows Computer Setup .....	16
Guest Linux Workstation Setup and build .....	17
Porting the VPROC SDK on Linux Platforms .....	22
Linux Basics .....	22
Porting Examples .....	22
Porting the SDK to a Raspberry Pi platform.....	23

Raspberry Pi platform info .....	23
Porting the SPI driver into the Pi.....	24
Porting the ALSA driver into the Pi .....	29
Porting the SSL into the Pi.....	32
SSL_lock_create .....	32
SSL_lock.....	33
SSL_unlock .....	33
Compile the SDK drivers.....	33
Compile the SDK demo Apps .....	33
Compile MiTuner Bridge Server .....	33
Compile the Firmware Converter Tool .....	34
SDK Testing and Debug .....	35
Install the kernel modules and configure the Pi .....	35
Install the Demo Apps.....	36
Install the Firmware Converter Tool .....	36
Testing the SDK .....	36
ZL380XX access over SPI using the Demo Apps .....	36
ZL380XX access over SPI using the procfs.....	38
open_device.....	38
close_device.....	38
write_reg.....	39
read_reg.....	39
load_fw .....	39
cfgrec.....	39
flash_save_fwrcfgrec .....	39
start_fw .....	40
flash_load_fwrcfgrec.....	40
flash_erase .....	40
Play and Record Audio with the ZL380xx.....	41
Record a wav file.....	41
Play a wav file.....	41

Troubleshooting.....	43
Compilation Debug .....	43
Possible Compilation error 1.....	43
Possible Compilation error 2.....	43
Possible Compilation error 3.....	44
Possible Compilation error 3.....	44
Loading the driver/apps debug.....	44
Possible Issue 1 .....	44
Possible issue 2 .....	45
Possible issue 3 .....	45
Possible issue 4 .....	45
Audio Playback/Recording Debug.....	46
Possible Issue 1: .....	46
Possible Issue 2: .....	47
Possible Issue 3: .....	47
SPI/I2C Communication error .....	47
Possible error 1 .....	47

## Revision History

Number	Revision	Description
	Date	
1	September 2017	Initial Release
2	September 2018	Added MiTuner Bridge Server paragraph

## Abbreviations

HBI	Host Bus Interface
SPI	Serial Peripheral Interface
I2C	Inter Integrated Circuit
I2S	Inter IC Sound
VPROC	Voice Processing
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

## Typographical Conventions

<i>File names/paths</i>	in italic
C-code Functions	in Courier New Fonts
<b>C-code Variables</b>	in Courier New Bold Fonts
Terminal commands	in Courier Fonts

## Introduction

The ZLS38100 Software Development Kit (SDK) is a collection of software, tools, code examples, and documents that allow rapid application development with the Microsemi Timberwolf device series. With the ZLS38100 Software Package, little or no knowledge of the low-level control of Timberwolf ICs is needed to fully utilize the chipset. The ZLS38100 software is designed to simplify implementation and reduce customers' time to market.

The SDK is written in C language, and it supports all the devices included into the Timberwolf device series (ZL3804x, ZL3805x, ZL3806x, ZL38080, ZL38090, etc., where x: 0, 1, 2, 3 or greater). SDK releases P2.x or greater do not include support for the Galileo devices. Galileo devices are supported in the version P1.0.0 of the SDK.

This Guide provides an overview of the ZLS38100 SDK and must be served as a guide to porting the SDK to a host platform. The SDK supports both Linux (Android) based and Non-OS based platforms. This guide will focus on the steps required to porting the SDK to a host using the Linux OS. The SDK can be compiled natively to the host platform or compiled via a cross-platform.

**Note:**

- *Within this document, the terms VPROC SDK and ZLS38100 SDK or ZLS38100 Software package are used interchangeably*
- *Readers of this document are assumed to be familiar with at least the basics of the Linux OS*
- *The term ZL380xx or VPD refers to all the devices included in the Timberwolf device portfolio*

This Guide covers the following topics:

- [Chapter 1](#): An overview of the components of the ZLS38100 SDK
- [Chapter 2](#): A description of the basic first steps prior to porting the code to an host platform
- [Chapter 3](#): An overview of the porting environment and Operating System basics
- [Chapter 4](#): The steps to port the SDK to a desired host platform
- [Chapter 5](#): The steps to verify that the SDK is properly ported into that host platform and troubleshooting

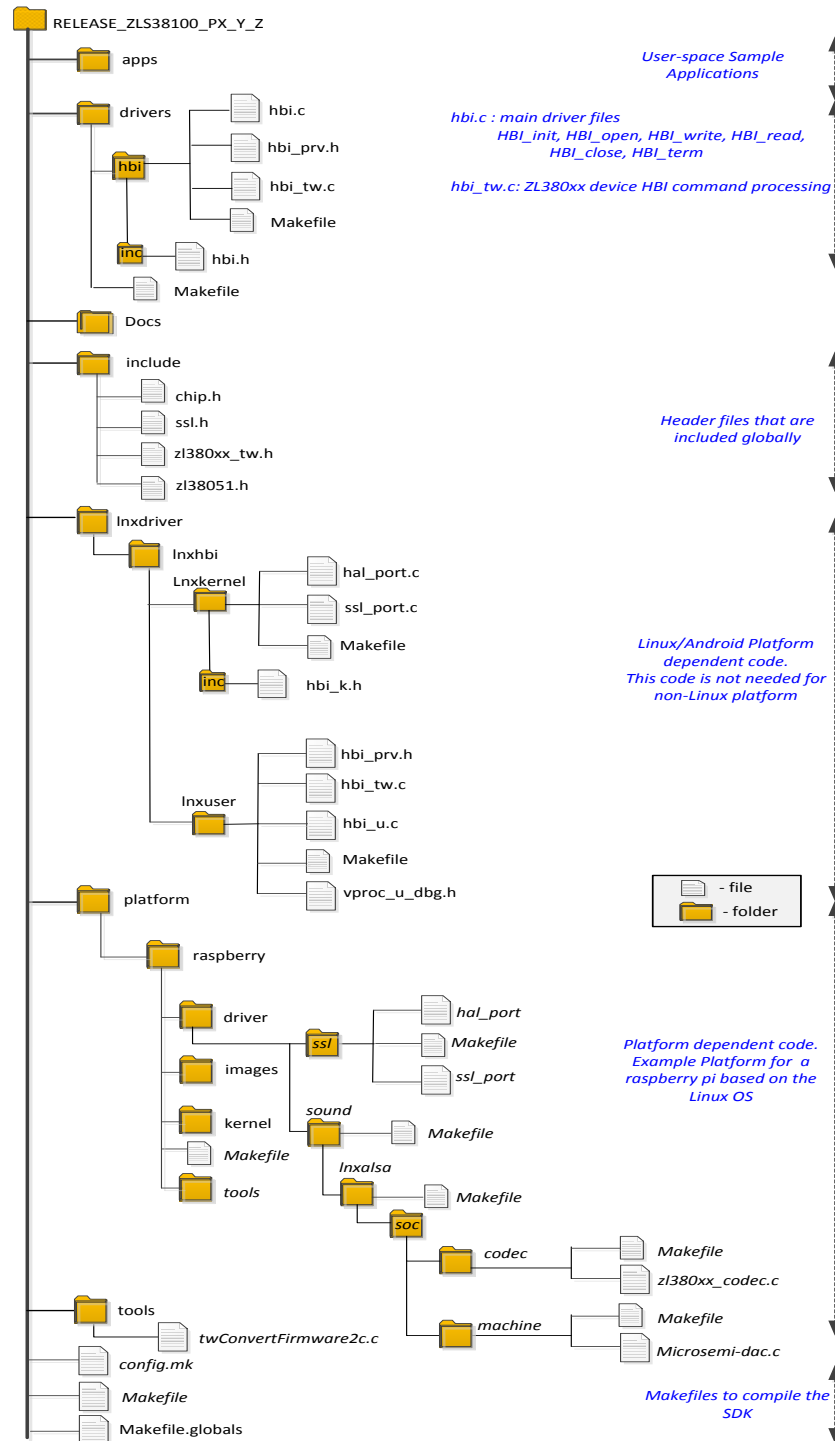
## Other References

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

- ZLS38100 Reference Guide
- ZLK38xx Firmware Manual
- ZLS38508 MiTuner User Guide  
<http://www.microsemi.com/products/audio-processing/design-resources/mituner>
- GNU make Documentation  
<https://www.gnu.org/software/make/manual/make.html>
- Linux Device Tree Source Documentation  
[http://elinux.org/Device\\_Tree\\_Reference](http://elinux.org/Device_Tree_Reference)
- ALSA Documentation  
<http://www.alsa-project.org/alsa-doc/alsa-lib/>
- Linux Kernel Documentation  
<https://www.kernel.org/doc/>

## ZLS38100 Software Development Kit Contents

The ZLS38100 Software Package contains tools, code, documentation, and example applications for developing products based on the Timberwolf chipset. The following is a list of components distributed in the SDK.





## Software Design Flow

The following diagram and table below introduce the prominent elements of the VPROC SDK software architecture. It is recommended that the user reads the VPROC SDK Reference Guide for more precise details on these components.

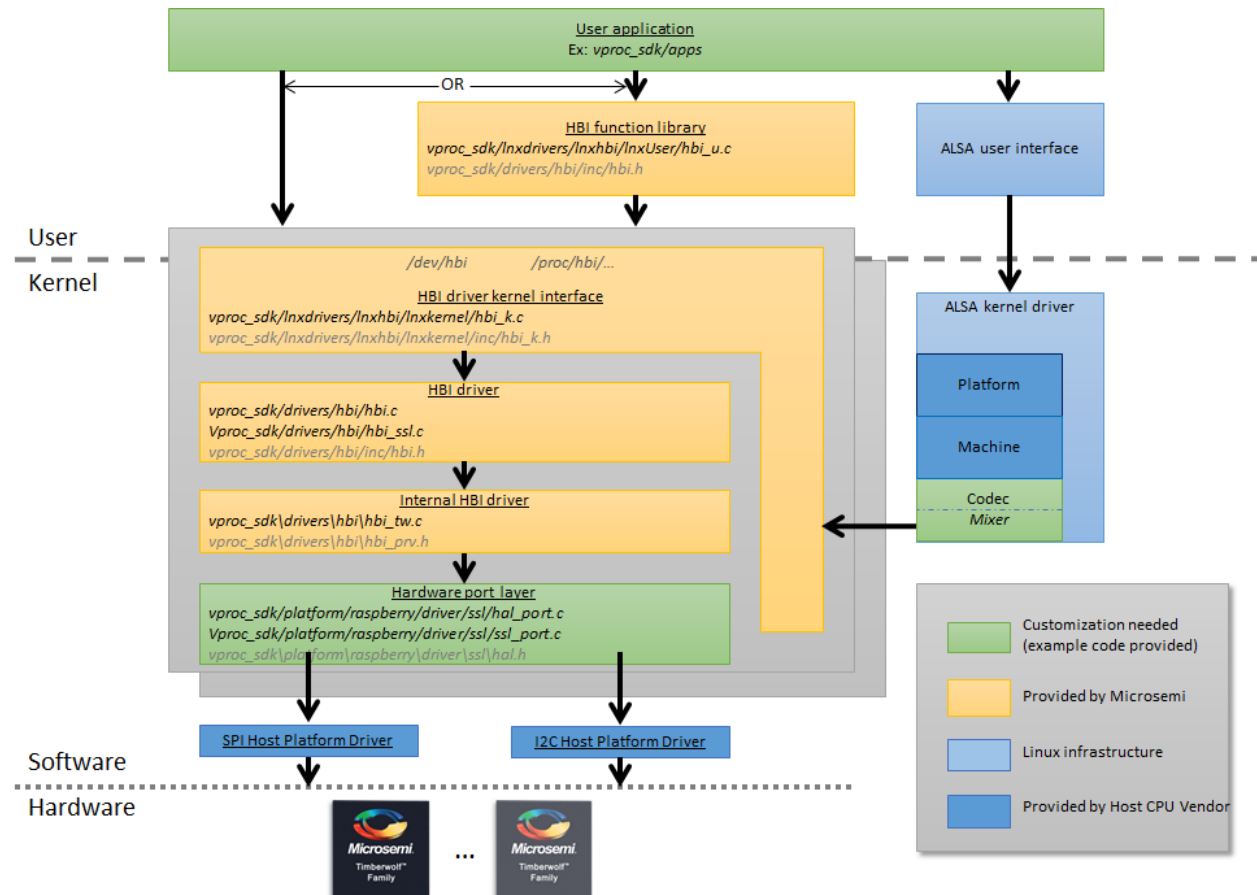


Figure 1: VPROC SDK Software Architecture for the Timberwolf devices

### Software Architecture Conventions:

- Blocks in **yellow** are provided by Microsemi. The implementation of these codes is complete and must not be modified by the SDK user.
- Blocks in **green** are codes that must be implemented by the SDK user in order to port the SDK to a desired platform. However, Microsemi provides tested and working example codes that can be ported to the customer's platform with little to no changes. These changes will be discussed in the [porting](#) chapter of this document.
- Blocks in **blue** are codes that are specific to the host platform. Some of these codes are provided by the CPU vendor, and some are already included in the platform OS.

SDK Directory Structure and Files		Description
<b>RELEASE_ZLS38100_PX_Y_Z/</b>		
<b>apps/</b>	<i>hbi_load_firmware.c</i>	Example user-apps to load a desired Timberwolf image (firmware (*.s3) and/or configuration(*.cr2)) into the device
	<i>hbi_load_grammar.c</i>	Example user-apps to load a desired Sensory grammar+ search files into the Timberwolf device
	<i>hbi_test.c</i>	Example user-apps to read, write one or up to 256 registers of the device. Load image stored from an external flash into the device RAM, erase specific image or the entire external flash
<b>drivers/</b>	<i>hbi.c</i>	Implements the 5 basics main functions of the SDK (Open, Read, Write, Reset, Close)
	<i>inc/hbi.h</i>	The SDK variables and prototypes for the functions defined within the hbi source code
	<i>hbi_tw.c, hbi_prv.h</i>	The layer that performs the translation from register+data into HBI command format
<b>include/</b>	<i>hal.h, zl380xx_tw.h, ssl.h, chip.h</i>	Contains the header files that must be included in applications that use the SDK
<b>lnxdrivers/lnxhbi/</b>	<i>lnxkernel/</i>  <i>hbi_k.c</i>  <i>inc/hbi_k.h</i>	Required only if the SDK is ported to a Linux platform. It implements the Linux kernel aspect of the SDK that binds the HBI layer to the platform port layer of the SDK
	<i>lnxuser/Hbi_u.c</i>	Required only if the SDK is ported to a Linux platform. It implements the Linux user-space aspect of the SDK that binds the HBI layer to the user-space

			application layer of the SDK
<b>Platform/</b>	<b>Ambarella/</b>	<i>driver/ssl/hal_port.c</i>	SPI or I2C slave device driver code for a Linux based platform
		<i>driver/ssl/ssl_port.c</i>	System Service Layer code. This code is Operating System Specific
		<i>driver/sound/lnxalsa/soc/codecs/</i> <i>zl380xx_codec.c</i>	ALSA codec driver for the Timberwolf device
		<i>driver/sound/lnxalsa/soc/machine/</i> <i>s2l_zl380xx_audio.c</i>	ALSA machine driver for the Ambarella S2L in compatibility with the Timberwolf device
		<i>include/typedefs.h</i> <i>include/vproc_dbg.h</i>	Platform specific C-code variable data type definition (typedef). Platform specific debug print function definition
		<i>Kernel/</i>	Repository for the Ambarella patched Linux kernel need to cross-compile the SDK
		<i>tools/</i>	Repository for the Ambarella cross-compiler toolchain needed to compile the SDK
	<b>NxpJN516x/</b>	<i>driver/ssl/hal_port.c</i>	SPI or I2C slave device driver code for the NXPJN516x ZigBee series
		<i>driver/ssl/ssl_port.c</i>	System Service Layer code. This code is Operating System Specific. Current implementation pertains to a platform with no Operating System.
		<i>include/typedefs.h</i> <i>include/vproc_dbg.h</i>	NxpJN516x specific C-code variable data type definition (typedef). NxpJN516x specific debug print function definition
	<b>Raspberry/</b>	<i>driver/ssl/hal_port.c</i>	SPI or I2C slave device driver code for a Linux based platform
		<i>driver/ssl/ssl_port.c</i>	System Service Layer code. This code is Operating System Specific. Example implementation is for a Linux based platform

		<i>driver/sound/lnxalsa/soc/codec/ zl380xx_codec.c</i>	ALSA codec driver for the Timberwolf device
		<i>driver/sound/lnxalsa/soc/machine/ microsemi-dac.c</i>	ALSA machine driver for the Raspberry Pi in compatibility with the Timberwolf device
		<i>include/typedefs.h include/vproc_dbg.h</i>	Platform specific C-code variable data type definition (typedef). Platform specific debug print function definition
		<i>Kernel/</i>	Repository for the Ambarella patched Linux kernel for cross-platform compiling of the SDK
		<i>tools/</i>	Repository for the Ambarella cross-compiler toolchain needed to compile the SDK
	<b><i>tools/</i></b>	<i>twConvertFirmware2c.c</i>	A utility user application to convert the Timberwolf device firmware and configuration into a binary file (*.bin) or a C-Code inline header file (*.h) files. The utility can be compiled on a Linux or Windows platform.
	<b><i>config.mk</i></b>		Configuration file where global variables defined within the Makefile.globals are passed to the SDK during compilation
	<b><i>Makefile.globals</i></b>		Definition of global compile-time variables used by the SDK
	<b><i>Makefile</i></b>		Main SDK GNU make file. It defines the main target rules used to compile every aspects of the SDK

Table 1: VPROC SDK Software Components

### The SDK main components in brief

The VPROC SDK code is divided in two layers. A platform independent layer which includes codes that execute in user space, and codes that execute in kernel space. A Platform dependent layer which includes codes that run solely in kernel space.

### Platform independent Layer

This Layer implements the basic aspects of a driver such as OPEN a connection to the underlying device, WRITE to or READ from a register of the device, and CLOSE that connection when finish.

Since the implementation of this layer is fully complete and must not be modified by the SDK user, then this document will not go into the details on how these tasks are performed, this will be discussed in details in the VPROC SDK reference guide.

The codes implemented by this Layer are labelled by the blocks “HBI Driver” and “Internal HBI Driver” in the block diagram of [figure 1](#). The Microsemi provided example user space Applications are platform independent as well.

### Platform dependent Layer

This layer implements the HAL, SSL, and for a Linux based platform ALSA drivers that extend the capability of a host platform in order to communicate with the ZL380xx devices over a SPI or I2C bus, and exchange audio samples over an I2S bus.

#### Hardware Abstraction Layer (HAL)

The HAL layer is the code that implements a slave SPI or I2C driver. The host will use this code whenever it wants to read from or write to a register of the ZL380xx device.

#### System Service Layer (SSL)

The SSL layer is the code that implements a mutual exclusion. Meaning, since only one instance of the host user application can communicate with the ZL380xx device at a time, then it is important for the SDK to provide a mechanism to prevent an-ongoing HBI transaction from being perturbed. Therefore, the SSL must implement locking and unlocking mechanism to prevent this.

During HBI transaction the HBI bus will be locked via the SSL, any other simultaneously transaction will be queued and executed once the current HBI transaction is completed and the HBI bus unlocked.

The SSL only matters for platforms that are based on a multi-task, multi-thread OS. Otherwise the implementation for the SSL functions can be a simple `return 0;` statement.

#### Advanced Linux Sound Architecture Layer (ALSA)

ALSA is a very sophisticated software API for sound card driver development. It is available for the Linux platform. This Library can be used to create simple to complex sound device drivers for a Linux system.

A typical ALSA driver consists of 3 layers. It includes the following 3 sub-drivers:

- **Platform driver:** implements the audio DMA engine driver, digital audio interface (DAI) drivers (e.g. I2S, AC97, PCM) and any audio DSP drivers specific to that platform.
- **Machine driver:** acts as the glue that describes and binds the Platform and the Codec component drivers together to form an ALSA “sound card device”. It handles any machine

specific controls and machine level audio events (Ex: turning on an amp at start of playback, etc.).

- **Codec driver:** is platform independent and contains audio controls, audio interface capabilities, codec Dynamic Audio Power Management (DAPM) definition and codec read/write access functions.

Like any Linux based SDK, ALSA includes a library that is strictly for ALSA kernel device driver development, and a related Library that is strictly for user-space ALSA application development. The 3 drivers layers mentioned above are ALSA kernel drivers. The Platform driver is provided by the host CPU vendor, while the codec driver is provided by the codec vendor. The Machine driver can be developed by either the CPU or the codec vendor. But typically, an example implementation of the Machine driver is always included with the SDK provided by the CPU vendor.

## First Steps

Prior to porting the SDK to the particular platform, certain variables within the main Make files at the root path of the SDK need to be set in accordance to the design and the host platform. Each of these settings will be discussed below.

This document will not focus on make files; it is assumed that the user of this document is familiar with the concept of make files. For more info on make files please see the GNU Make documentation.

<https://www.gnu.org/software/make/manual/make.html>

## SDK Basic Functions

In order to understand how these Make file variables relate to the SDK, first here is a basic overview of the functional behavior of the SDK.

The main driver of the SDK is based on the following four mains functions:

OPEN, WRITE, READ, CLOSE

These four functions are used by the user applications in order to control every aspect of the SDK and the underlying Timberwolf device.

### OPEN

The driver supports multiple Timberwolf devices. Therefore, the SDK is structured so that each one of these Timberwolf device is assigned a distinct device ID. When the host needs to access a particular Timberwolf device, it must first ask the driver to provide access to that Timberwolf device by passing it the device ID of that Timberwolf. The OPEN function is referred in the SDK as `HBI_open()`. It performs the task of verifying that the device exists, and then opening that instance of the driver file related to that particular Timberwolf device ID.

The number of Timberwolf devices and the number of device instances that can simultaneously be opened by the SDK are defined within the `Makefile.global` of the SDK.

### WRITE

The host application must use this function (referred in the SDK as `HBI_write()`) to write to one or up to 128 registers of the Timberwolf device in a single access. The registers of the Timberwolf device are documented in the Timberwolf Firmware manual.

The buffer size needed to store data to write to the device and also to store important runtime variables of the SDK must be specified by the user of the SDK. This buffer size is defined within the `Makefile.global` of the SDK.

## READ

The host application must use this function (referred in the SDK as `HBI_read()`) to read from one or up to 128 registers of the Timberwolf device in a single access. The registers of the Timberwolf device are documented in the Timberwolf Firmware manual.

Both the READ and WRITE functions must be passed the handle (reference) value returned by the OPEN.

## CLOSE

When the host no longer wants to communicate with a particular Timberwolf device instance, then the device instance opened during the OPEN must be closed, in order to release resources allocated by the OPEN. The CLOSE function is referred in the SDK as `HBI_close()`.

## Linux Platform setup requirements:

Let's assume that the ZLS380100 SDK is already downloaded into your cross-platform development Computer. This computer can be a Windows, Linux or Mac OS based computer. From the root folder of the SDK, there is a file named `Makefile.globals` which contents are described in the table below.

`Makefile.globals` :

Variables	Description	Options
PLATFORM	A name that identifies the user host platform. The name given must be the exact same name of the folder under <code>RELEASE_ZLS38100_PX_Y_Z/platform/</code>	Ex: for a Raspberry pi platform  PLATFORM=raspberry
TARGET	Identifies Target VPROC device	TW=1
HBI	Identifies the Host to VPROC device Interfacing bus type	I2C=1 SPI=2
HBI_BUF_SIZE	Specifies the maximum buffer size to allocate for driver data.	1024 up to the capability of the platform
HOST_ENDIAN	Identifies host Micro-Processor endianness	little : for little-endian big : for big-endian
VPROC_DEV_ENDIAN	Identifies ZL380xx device endianness	little : for little-endian big : for big-endian
BOOT_FROM_HOST	Option to enable Booting of target device over HBI	yes : for boot to host no : Otherwise



FLASH_PRESENT	Specifies whether a slave flash device is attached to the ZL380xx to store ZL380xx firmware, configuration and ASR images	yes :if a flash is present  no : If no flash
BUILD_TYPE	Indicates the build type	DEBUG, RELEASE
HBI_MAX_INST_PER_DEVICE	Specifies the maximum number of user applications or threads that can simultaneously open an instance on a particular ZL380xx device	1 up to 256
VPROC_MAX_NUM_DEVS	Specifies the maximum number of ZL380xx devices that need to be controlled by the VPROC SDK	1 up to 256
NUM_MAX_LOCKS	Specifies the maximum number of mutual exclusion locks that can be in effect.	User defined value, by default 100
DEBUG_LEVEL	To enable the desired level of debugging information that must be reported from the VPROC SDK.	0: none  0x1 : function Entry/Exit info,  0x2 : informational  0x4 : warning  0x8 : error  0x1F : All
HBI_LOAD_FWR_STATIC	Specifies whether a ZL380xx firmware image (*.h) needs to be statically compiled with the SDK.	yes : if static  no : otherwise
HBI_LOAD_CFGREC_STATIC	Specifies whether a ZL380xx configuration record image (*.h) and support to loading that image at boot time needs to be statically compiled with the SDK.	yes : if static  no : otherwise
HBI_ENABLE_FWR_BIN	Specifies whether support to handling loading of a binary (*.bin) ZL380xx firmware image at boot time must be compiled into the SDK.	yes : if static  no : otherwise
ZL380XX_FIRMWARE_IMAGES_PATH	Specifies the path to static (*.h) ZL380xx firmware and configuration record images	Default path is a folder named images under the particular platform
KSRC	Specifies the path of the Linux kernel headers	

	needed to compile the SDK	
TOOLSPATH	Specifies the path where to find the tool chain (compiler) with which to compile the SDK	Only required for cross-compiling
CROSS_COMPILE	Specifies the compiler name	
ARCH	Specifies the platform(controller) architecture	

*Table 2: Compile-time Make File variables*

For most designs, the variables of the make files that will be changed from the default settings are described below.

#### Host Platform naming

The SDK is provided with full support for Linux based platforms such as the Ambarella S2L, and the Raspberry pi platforms. Also, full support for the NXP (no OS) JN516x ZigBee Micro-controllers.

If the host platform is based on the Ambarella S2L micro-controllers, then set the PLATFORM variable within the Makefile.globals to:

```
PLATFORM=ambarella
```

If instead the host platform is based on the Raspberry pi, then set the PLATFORM variable within the Makefile.globals to:

```
PLATFORM=raspberry
```

If instead none of the platform names currently included within the SDK relates to the actual host micro-controller, then the customer can simply rename either one of the existing platform, and modify the code within that platform accordingly to their design.

Let's say the user of the SDK is using an NXP IMX6 Micro-controller and would like to use the raspberry platform as the starting point for their design. Then, they can simply rename the raspberry platform name to a desired platform name

Example:

```
PLATFORM=imx6
```

Note that the platform name can be any name that the user of the SDK desires, however that name must be without space or special character. That name follows the exact same requirements as any C language variable. It is recommended to use one of the existing platforms as a starting point.

#### Timberwolf devices

The SDK must know the total number of Timberwolf devices that will be controlled by the SDK. That number must be specified by the VPROC\_MAX\_NUM\_DEVS variable of the Makefile.globals.

Example:

If the design needs to support two Timberwolf devices, then that variable must be set to 2.

```
VPROC_MAX_NUM_DEVS=2
```

### User Application Instances

Depending on the customer design requirements, their design may have a requirement that multiple applications run simultaneously, and therefore each one of these application will need to talk to a Timberwolf device. In that case each of the applications will need to keep an open instance to the driver in order to read/write registers of the Timberwolf device.

This case is very common in host platform OS (Ex: Linux, VxWorks, ...) that supports multi-threading. Where each thread of the customer application, may be performing a different task, which requires separate access to the Timberwolf device whenever needed. The desired number of device driver instances that could be opened simultaneously per Timberwolf device must be specified by the `HBI_MAX_INST_PER_DEV` variable of the `Makefile.globals`.

Example:

If the design will need to have two open connections to a Timberwolf device, then that variable must be set to 2.

```
HBI_MAX_INST_PER_DEV=2
```

## Compile/Development environment computer

For compilation and development with the VPROC SDK, a Linux computer is needed. That computer depends on the target platform to which the SDK will be compiled for and ported to. If that target platform is a full blown computer such as the Raspberry Pi etc, where a mouse, a keyboard and monitor screen can be attached. Then the VPROC SDK can be simply copied into that machine and compiled statically. In that case skip the info below and move on to the “SDK Porting Steps” described in the next chapter.

But if the target platform is not a full computer such as a demo development board, then such platform will only provide an interface to load/flash an image into the board. Therefore, a cross-platform Linux machine is needed to compile the VPROC SDK first and then load the resulting drivers into that target platform either with a programmer, or via USB, Ethernet etc. as per the platform requirements.

The document highlights the steps to create a cross-platform Linux machine that must be used to compile the SDK. The steps described in this document assume that the user of the SDK already has a Windows-based desktop computer with the minimum requirements described below. The document will walk the user through the steps to adding a virtual Linux workstation into that existing Windows-based workstation

### Windows host Computer minimum Requirements

CPU: Intel X5 or later

RAM: 2GB or more

HDD: 100GB or more

Ethernet: Dual 100/1000 Mbps Ethernet cards (NICs)

Graphic card: Graphic card with a t least 256MB

OS: Windows 7 or later

Monitor: 14” or bigger

### Compile/Development platform Network Architecture

The Picture below illustrates a cross-compiling system. Where the host Windows OS computer has two network cards (NICs). From the Windows OS a guest virtual Linux workstation is installed via Oracle VirtualBox. The guest Linux OS and the Windows Host OS share the resources (RAM, HDD, Graphic, Monitor mouse and Keyboard) of the computer and appears as two different computers to the external world. Each OS has access to both NICs, but, each NIC is assigned a distinct IP address on each respective OS.

The Linux virtual guest has two shares: share 1 and share 2. Share 2 is a small share; it will be used to share any files that must be accessible via both the target platform system and the Windows Host

system. Share 1 is a more massive shares, it will be used to share any files/repositories (Such as: host SDK, VPROC SDK, Tool Chain, Linux headers etc.) that must be accessible via the Windows Host system.

Example: The VPROC SDK and any other host platform SDK can be stored in share 1; therefore the developer will have the option to use either a code editor on Windows or the Linux workstation for code development. Once the VPROC SDK is compiled, then the compiled object codes can be copied to share 2, for accessibility via the target platform system.

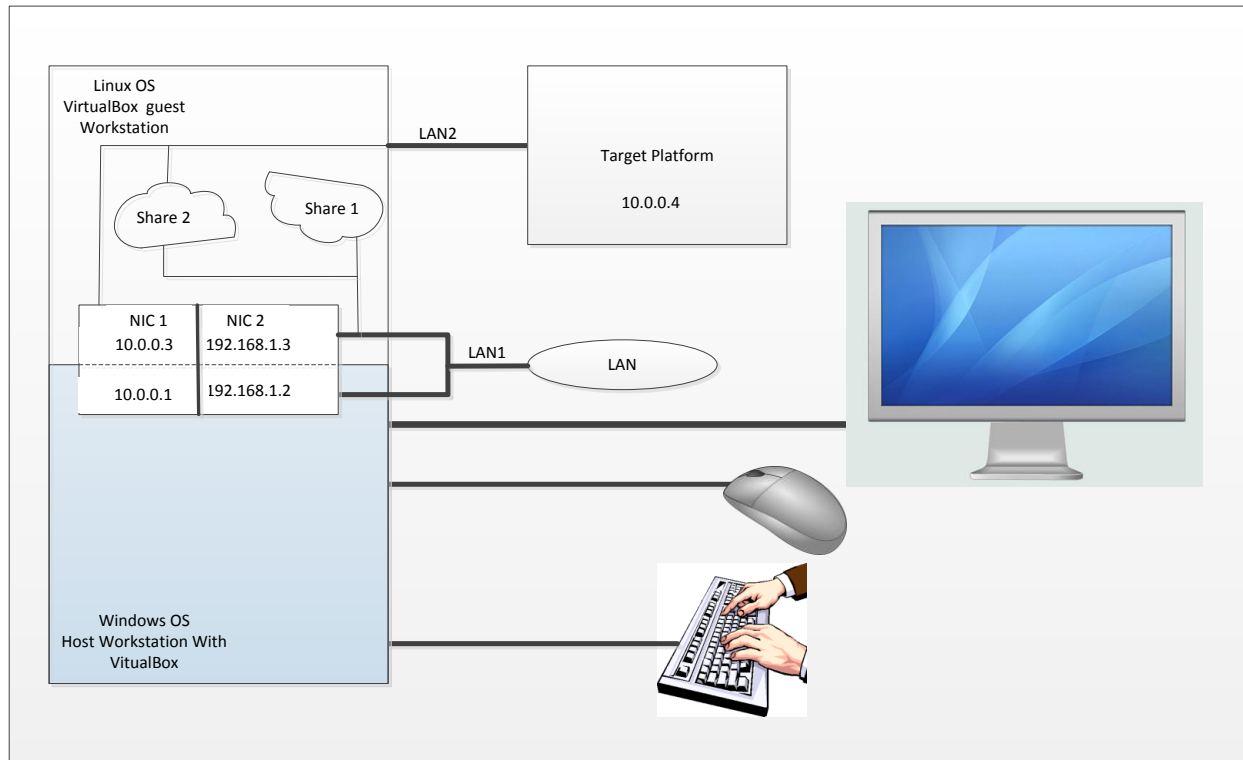


Figure 2: Shared development Environment

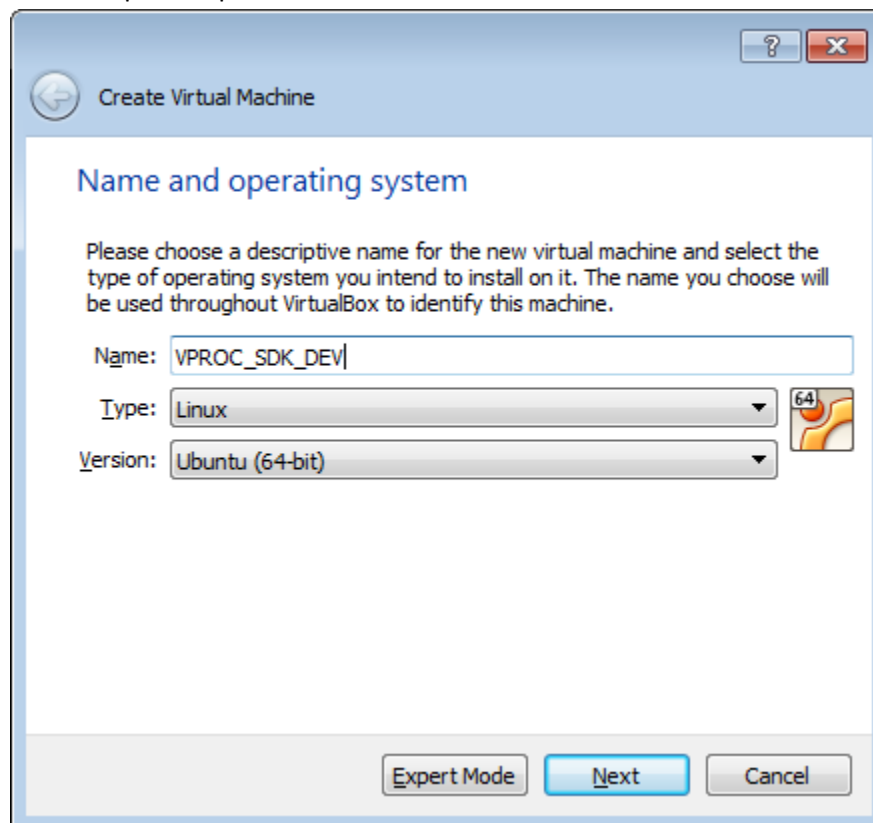
Workstations	IP	Netmask	Gateway
Windows host	10.0.0.1	255.255.255.0	10.0.0.1
	192.168.1.2		192.168.1.0
Linux guest	10.0.0.3 (LAN 2)		10.0.0.1
	192.168.1.3 (LAN 1)		192.168.1.0
Target Host platform	10.0.0.4		10.0.0.1

Table 3: Example IP address assignment

## Host Windows Computer Setup

- Download Oracle VirtualBox for Windows host version is 5.1.28 or later  
<https://www.virtualbox.org/wiki/Downloads>
- Download Ubuntu Desktop version 16.04.3 LTS or later  
<https://www.ubuntu.com/download/desktop>
- Install VirtualBox into the Windows Workstation.
  - Once installation is completed, open VirtualBox and click on New to create a new Virtual guest machine. Make sure the Machine type is set to Linux and version to Ubuntu (64-bit) if you have downloaded the 64-bit version of Ubuntu, 32-bit otherwise. Then Click Next.

See Example setup below



- Setup the resources' for the guest Linux machine.

Example

Memory Size: 512MB or 1024MB depending on how much Memory your Windows host PC has. Keep in mind that your computer resources will be shared between the host and the guest machines. Then Click Next,

You will be asked to select hard disk type. Select "Create a Virtual Hard disk" option. Then click Create. Select "VHD" Disk Type for next option. Then make sure to set the Virtual hard disk size to either "**Dynamic**" or "**Fixed**", either one is fine. However I recommend that it is set to Fixed and set the amount to something like 20GB or more

depending on the size of files you will be storing on into that virtual machine. Click Create, and then Finish.

- Configure the newly created Virtual machine
  - Click **Settings > Storage > Empty** from the Main VirtualBox window.
  - Then, click on the DVD icon next to the “Optical Drive” and Choose “**Virtual Optical Disk File**” then browse to the location on your Windows PC drive to where the Ubuntu ISO image was downloaded and load it into that virtual Optical drive.
  - From the Main VirtualBox window click **Settings > Network**
    - Click on the **Adapter 1** tab. Select “**Enable Network Adapter**”
    - Set “**Attached to**” to “**Bridged Adapter**”.
    - Set “**Name**” to the network card that is interfaced to LAN 1. The one that provides Internet or access to the Local Lan
    - Click on **Adapter 2** tab and repeat the settings above. Make sure the “**Name**” is set to the network card that is interfaced to LAN2

- Install Ubuntu into the virtual Guest machine, by clicking on “**Start**” from the main VirtualBox window

Note:

- Follow the Ubuntu installation steps.
- Once installation is completed, shut down the Virtual machine, then from the main Virtual Box window remove the image from the virtual optical drive  
**Settings > Storage** then click on the **CD/DVD** icon next to the **Optical Drive**, and select “**Remove disk from virtual drive**”
- Install the Guest Additions
  - This will install drivers for the mouse/keyboard and monitor inside the guest machine.
    - Click **Settings > Storage > Empty** under **Controller: IDE**. Click on the **CD/DVD** icon next to **Optical Drive** and browse to the **VboxGuestAdditions.iso** file. It is normally located here, or where ever you had chosen to install VirtualBox.  
`C:\Program Files\Oracle VM VirtualBox\ VBoxGuestAdditions.iso`
- Start the Guest Linux Machine by click on **Start** from the Main VirtualBox window.
  - Open a terminal and change to the directory where the CD-ROM is mounted  
Ex:  
`cd /media`  
`sudo ./VBosLinuxAdditions.run`

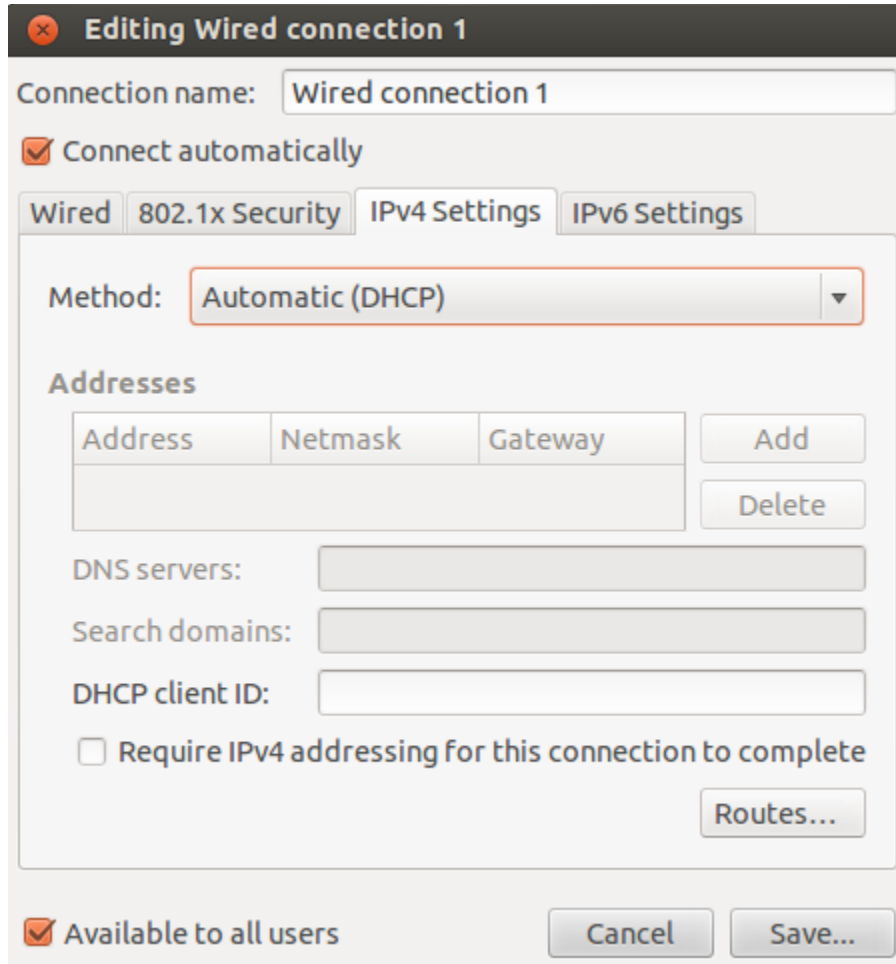
## Guest Linux Workstation Setup and build

- Create a share 1 folder (see next section on how to create a Samba or NFS share).
- Configure and Create a Network share inside the Linux guest machine.  
From the Ubuntu main screen, click on the “**System Settings**” icon, then click on **Network**. You should notice two wired networks. Make sure to identify to which network card each network is linked to by comparing the **Hardware address** found for that network with the **Mac address** from the **VirtualBox > Setting > Network > Adapter 1 > Advanced > Mac Address**

If that network matches the LAN1 network card, then click on Options and set the **IPv4 Settings** to **Automatic DHCP** or set to Manual with desired IP address depending on your LAN network DHCP setting

Ex: If the main LAN uses DHCP, then the DHCP server will assign an IP address to the Linux Guest machine as well as to the Windows machine.

Below is an example where LAN uses DHCP



**Editing Wired connection 1**

Connection name:

☒ Connect automatically

Wired | 802.1x Security | **IPv4 Settings** | IPv6 Settings

Method:

**Addresses**

Address	Netmask	Gateway

DNS servers:

Search domains:

DHCP client ID:

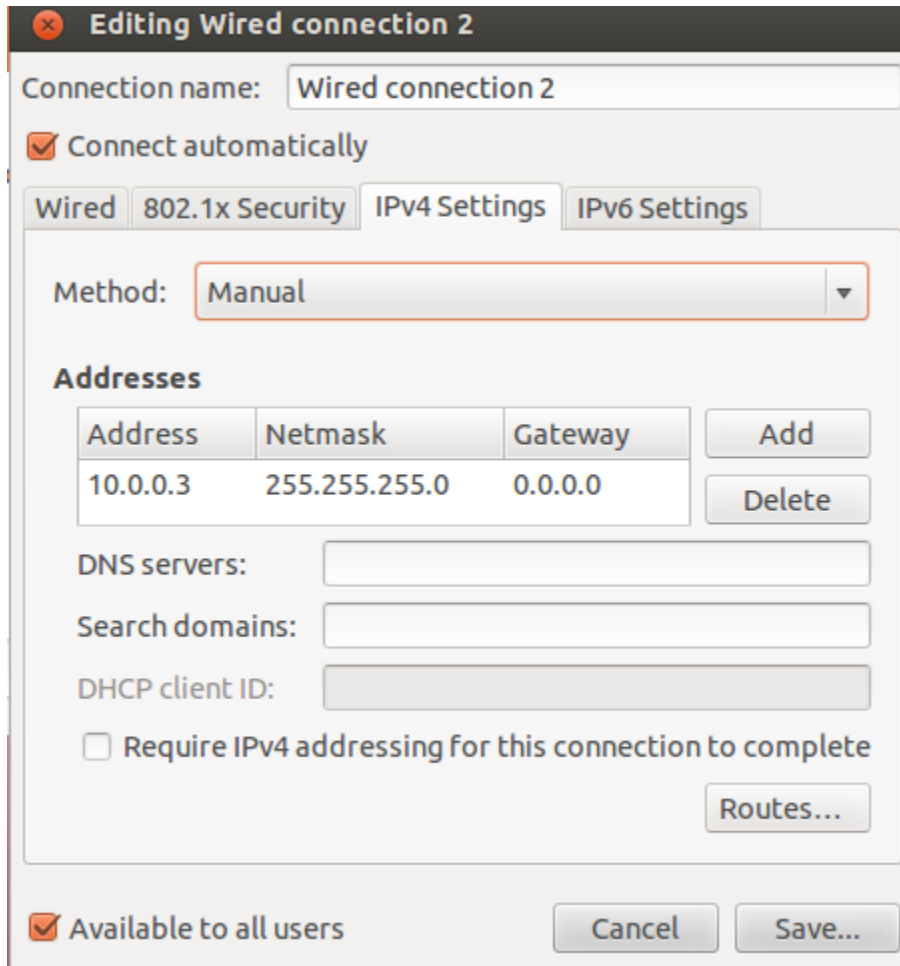
☐ Require IPv4 addressing for this connection to complete

☒ Available to all users

Repeat the above for the second wired network and second adapter. Click on options and set the IPv4 security to Manual and enter desired IP as per the LAN2 network.

Example: for assigning an IP address of 10.0.0.3 to the second network (LAN2) of the Linux Guest virtual machine.





**Editing Wired connection 2**

Connection name:

☒ Connect automatically

Wired 802.1x Security IPv4 Settings IPv6 Settings

Method:

**Addresses**

Address	Netmask	Gateway
10.0.0.3	255.255.255.0	0.0.0.0

DNS servers:

Search domains:

DHCP client ID:

☐ Require IPv4 addressing for this connection to complete

☒ Available to all users

- Install the Samba file system on the Linux Virtual machine
  - Open a terminal window in the Linux Virtual machine and issue the commands sequence below

```
sudo apt-get update
sudo apt-get install samba samba-common-bin
sudo cp /etc/samba/smb.conf /etc/samba/smb.conf.old
```
  - Create a directory that will be shared with other computers on the network  
Ex: to create a directory named shares under the Linux machine user's home directory  
Open a new terminal and issue the command below

```
pwd
```

That command will return the path for the home user currently logged into that Linux machine

Example: Let's say a user named xyz is currently logged then the command returns  
/home/xyz

**Note:**

*user in the path above is whatever user that is currently logged in the Linux machine. The idea is that we want to create the shared folder under the current user home directory. Therefore, make sure to replace user as per the home path returned by the pwd command*

Issue the command sequence below to create that share.

```
mkdir -m 777 /home/xyz/shares
sudo chown -R root:users /home/xyz/shares
sudo chmod -R ug=rwx,o=rx /home/xyz/shares
```

- Edit the Samba config file as per below or edit it with your preferred Linux editor. The nano editor is used in this case.  
`sudo nano /etc/samba/smb.conf`

Under the **##### Authentication #####** section add the line below if it does not already exist.

```
security = user
```

Under the [homes] section find the line

```
read only = yes
```

change it to

```
read only = no
```

Then at the end of the smb.conf file add the following lines

```
[shares]
comment = Public Storage
path = /home/xyz/shares
guest ok = yes
create mask = 0777
directory mask = 0777
read only = no
browseable = yes
```

- Save the smb.conf file if using the nano editor by pressing **CTRL+x** keys on your keyboard.
- Restart Samba  
`sudo /etc/init.d/samba restart`

That share created above is the share 1 mentioned in the block diagram. Repeat the above steps to create more shared directories if desired. That share will be visible and accessible by the Windows machine. In your Windows machine, open the file/directory explorer and enter the LAN1 IP address

of the Linux machine at the file browser bar. You should see the directory shares of the Linux machine in your Windows File/directory explorer.

The IP address of the Linux machine can be read via a Linux terminal inside the Linux machine using the command:

```
ifconfig
```

- Download the host platform SDK, tool chain and VPROC SDK into the Linux workstation shares directory.
- Install the tool chain into the Linux workstation. Then, update the VPROC SDK Makefile.globals related TOOLPATHS make file variable with the path to the toolchain, and cross-compiler name. as discussed in chapter 3 of this document

## Porting the VPROC SDK on Linux Platforms

The VPROC SDK as per latest version is approximately 350K bytes. The SDK can be ported to OS based platforms such as Linux, Android, VxWorks, etc. or non-OS based platforms. The VPROC SDK is not fully platform independent. The code under the /platform folder of the SDK is specific to that platform and must be implemented by the SDK user. The term “Porting” really is in reference to the code within that platform folder.

The platform folder as illustrated in the table 1 and figure 1 on chapter 2 of this document pertains to the Hardware Abstraction (HAL), System Service Layer (SSL), and for a Linux platform the Advanced Linux Sound Architecture (ALSA) Layer drivers that must be developed in accordance to that platform.

### Linux Basics

Linux is a Unix-like OS which is widely used on embedded system such as Phones, IP camera, embedded computer such as the Raspberry Pi etc. Linux source code is under GNU General Public Licence (GPL) and open to the public. Therefore, it is not necessary to write your program from scratch, code written by a Linux user under that License can be freely distributed to any Linux user. Microsemi provides working example codes for the VPROC platform specificity mentioned above that require little change if any to port it to a particular Linux platform.

Linux is a flexible real-time OS which can perform tasks with latency as low as 1ms. It supports multi-task and multi-user operations. Both of these operations can be implemented to run in either the Linux user space or kernel space.

The Linux OS itself is based on a Linux kernel, the Android OS itself is a Linux based OS, therefore code written or developed on a Linux platform can actually run on an Android platform that is based on that same Linux kernel. The kernel is a large complex code base, which is fully customizable to include/exclude specific features, as well as extendable to include support for new hardware such as the Microsemi ZL380xx devices.

The VPROC SDK is divided into two sections. The code under the platform folder of the SDK implements Linux kernel device drivers that are used to expand the capability of Linux so that a specific host processor interfaced to the Microsemi ZL380xx devices can communicate with the device over a SPI or I2C bus, exchange audio samples over an I2S bus.

### Porting Examples

The section of this document will cover an example of porting the SDK on a Linux platform. That platform is the popular Raspberry Pi platform flashed with the latest Raspbian (Jessie or Stretch) image.

The SDK has been ported and verified on multiple platforms with Linux kernels ranging from version 2.6.x to 4.9.x. (The SDK was last tested on kernel 4.9.41-v7+)

## Porting the SDK to a Raspberry Pi platform

The Raspberry Pi platform is a full Linux based platform. For those not familiar with the Raspberry Pi platform please refers to the link below.

<https://www.raspberrypi.org/>

As per the writing of this document, the latest Raspian image for the Raspberry Pi is Raspian Stretch release date 2017-09-07 based on Linux kernel 4.9.41-V7+.

If you have not downloaded the VPROC SDK into your Raspberry Pi yet, please do it now.

Porting the SDK to a Raspberry Pi platform is straight forward; it involves basically compiling the SDK on the Pi in order to create the following 3 Linux kernel modules that must be installed into the PI in order to provide support for the ZL380xx device.

Drivers	Description
hbi.ko	The HBI driver implements the slave aspect of a SPI or I2C slave driver. It allows the SPI or I2C master on the PI (Broadcom bcm2837) to communicate (read, write) with the ZL380xx over SPI or I2C
snd-soc-zl380xx.ko	It implements the codec aspect of an ALSA driver. That is the third layer in the ALSA driver structure that controls/configures the actual ZL380xx device for sound exchange with a Platform ALSA driver.
snd-soc-microsemi-dac.ko	It implements the machine aspect of an ALSA driver. That is the second layer in the ALSA driver structure. It binds the higher layer ALSA platform driver to the lower Layer codec driver in order to create a sound card for the system.

Table 4: VPROC SDK kernel modules

## Raspberry Pi platform info

The Raspberry pi platform is based on a Broadcom BCM2837 micro-controller. That Microcontroller is a 64-bit ARM8 Cortex A53 with 4 cores clocked at 1.2GHz. The Raspberry Pi provides a SPI interface with two dedicated SPI chip selects. One I2S, and two dedicated I2C and plenty of GPIOs that can be used to interface to more devices.

The Raspberry driver framework supports both older and later Linux device driver registration methods. By device driver registration, meaning a kernel device driver can be implemented either as a slave if that device for which this driver is implemented is a slave to another master device or a controller driver if that device for which the driver is implemented is a master device. Therefore, these two devices master and slave must be aware of each other and therefore create a registration procedure so that both drivers act like a single entity for full duplex communication between the related devices.

The Raspberry pi driver framework supports the board info older device registration method or device tree source (dts) based registration method. The example driver codes provided by Microsemi support both registration methods. In order to demonstrate the use of both registration methods, we will register the SPI/I2C driver using the old non dts based method, and the sound drivers using dts based method

### Porting the SPI driver into the Pi

Microsemi provides a sample HAL driver that can be compiled as either a slave SPI or I2C driver. See the file `/RELEASE_ZLS38100_PX_Y_Z/platform/raspberry/driver/ssl/hal_port.c`

This code requires minimal change to compile it and port it into the Raspberry Pi. It implements the lower level codes to read/write SPI data from/to the SPI bus

The only change required to that code is to create and initialize an instance of the `ssl_dev_info_t` driver info structure as per the desired number of Timberwolf devices to support and whether to load or not a firmware and related configuration record into these devices at boot time.

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 serialize device access */
        .imageType = 0,    /*0: for static *.h, 1: for *.bin */
    },
    {
        .chip = 38042,      /*Microsemi chip number without the ZL: Ex 38063*/
        .bus_num = 0,      /*SPI or I2C bus number*/
        .dev_addr = 1,     /*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 serialize device access */
.imageType = 0,     /*0: for static *.h, 1: for *.bin */
}

};
```

**Note:**

- 1) If for example the host has a firmware binary image named `vprocfirmware_zl38063_0.bin` and a binary configuration record named `vproconfig_zl38063_0.bin` that need to be loaded into the 38063 device at boot time, then the following member variables in the above `sdk_board_devices_info` definition must be defined as per below for the 38063 entry in the structure initialization

```
.pFirmware = (uint8_t *) "vprocfirmware_zl38063_0.bin",
.pConfig = (uint8_t *) "vproconfig_zl38063_0.bin",
```

- 2) If for example the host has a firmware \*.h image named `vprocfirmware_zl38063_0.h` and a \*.h configuration record named `vproconfig_zl38063_0.h` that need to be compiled with the SDK and loaded into the 38063 device at boot time, then the following member variables in the above `sdk_board_devices_info` definition must be defined as per below for the 38063 entry in the structure initialization

```
.pFirmware = (uint8_t *) vprocfirmware_zl38063_0,
.pConfig = (uint8_t *) vproconfig_zl38063_0,
```

**Note:** \*.h firmware images and configuration records that must be compiled with the SDK must be located within the directory named *images* within that particular platform

*Example for the raspberry pi platform, these images must be within:*

*/RELEASE\_ZLS38100\_PX\_Y\_Z/platform/raspberry/images/*

### Modify the Pi dts for the SPI

The Raspberry Pi SPI bus and its chip selects are occupied by default by a generic slave SPI driver named `spidev` that is provided with the Linux kernel. In order to make the SPI available for use with another slave SPI driver the dts file on the Raspberry used for driver registration must be modified to remove the assignation of the SPI bus and chip selects from the `spidev` driver, so that the SPI and related CS can be used by the VPROC hbi SPI driver.

This can be done by simply writing a dts overlay file that implements the necessary code to perform the above registration requirements of the driver.

A dts overlay, basically implements only changes to perform to an existing dts file.

An example of the overlay file to free-up the SPI chip selects from the spidev driver is given below

```
/dts-v1/;
/plugin/;
/ {
    compatible = "brcm,bcm2708","brcm,bcm2709";
    fragment@0 {
        target = <&soc>;
        __overlay__ {
            spi0: spi@7e204000{
                status = "okay";
            };
        };
    };
    fragment@1 {
        target = <&spi0>;
        __overlay__ {
            spidev@0{
                status = "disabled";
            };
            spidev@1{
                status = "disabled";
            };
        };
    };
};
```

This file is located at the following path within the SDK.

/RELEASE\_ZLS38100\_PX\_Y\_Z/platform/raspberry/kernel/dts/microsemi-spi-multi-tw-overlay.dts

This dts overlay simply free-up the SPI chip select 0 and 1 of the Micro-controller. Therefore, when the HBI driver is executed it will register to the PI SPI controller not via the device tree method, but via the old device probing and attach method. If optionally the HBI SPI driver device tree registration mode is preferred, then use the example dts below.

That example dts contains the same two fragments in the previous example, but it adds one more fragment in order to force the HBI driver to register using device tree compatibility method.



```
/dts-v1/;
/plugin/;

/ {
    compatible = "brcm,bcm2708","brcm,bcm2709";
    fragment@0 {
        target = <&soc>;
        __overlay__ {
            spi0: spi@7e204000{
                status = "okay";
            };
        };
    };
    fragment@1 {
        target = <&spi0>;
        __overlay__ {
            spidev@0{
                status = "disabled";
            };
            spidev@1{
                status = "disabled";
            };
        };
    };
    /*register two devices hbi0, hbi1 at cs0, cs1 respectively*/
    fragment@2 {
        target = <&spi0>;
        __overlay__ {
            hbi0@0 {
                compatible = "microsemi,z138xx0";
                reg = <0>;
                spi-max-frequency = <20000000>;
                status = "okay";
            };
            hbi1@1 {
                compatible = "microsemi,z138xx1";
                reg = <1>;
                spi-max-frequency = <20000000>;
                status = "okay";
            };
        };
    };
};
```

Basically, as you probably already figured out, the device tree source is a programming language in its own right with its syntaxes closely related to C. The implementation of a device source can be as practical as you want it to be. In the examples dts code above, in the first example there is a compatibility line and two fragments (Sections). The first line in a dts file is typically a compatible line. That line specifies a key name that drivers can include into the code, so that Linux can search for

another compatible master driver with a similar name in its code in order to bind (register) these two drivers together. That name although can be anything, but as a consensus it is a name that closely refers to the actual micro-controller vendor or sometimes CPU name.

The first fragment is to tell Linux which SOC and which one of its SPI buses that you will use.

Ex:

```
target = <&soc>;      /*The SOC pointed by soc defined in the main dts*/
spi0: spi@7e204000    /*The SPI bus 0 at hardware address 7e204000h*/
```

The second fragment is to tell Linux a driver named spidev is currently assigned to the chip selects 0 and 1 of that SPI bus, de-assign it to both.

Ex:

```
spidev@0{
    status = "disabled";
};
spidev@1{
    status = "disabled";
};
```

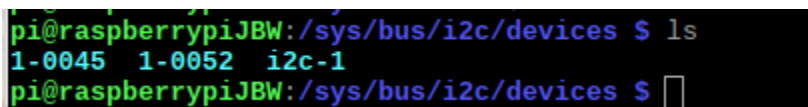
And, optionally depending on the preferred registration method the third fragment is to add device tree compatibility nodes for the desired device drivers to register.

Ex:

```
hbi0@0 {
    compatible = "microsemi,zl38xx0";
    reg = <0>;
    spi-max-frequency = <20000000>;
    status = "okay";
};
```

As you can see above, the dts includes hardware addresses for the different peripherals of the Host platform CPU, therefore, the CPU vendor will always provide a dts file for their platform.

Similar example dts overlay is provided for I2C interfacing. Example picture below shows two devices at addresses 0x45 and 0x52 registered to I2C bus 1 as defined in the provided dts overlay once the driver is compiled as I2C and insmoded.



A terminal window showing the command 'ls' in the directory '/sys/bus/i2c/devices'. The output lists two devices: '1-0045' and '1-0052', both associated with the driver 'i2c-1'.

Example for the Raspberry Pi, the dts file for the Pi3 model B can be found here for kernel 4.9.y.

<https://github.com/raspberrypi/linux/blob/rpi-4.9.y/arch/arm/boot/dts/bcm2710-rpi-3-b.dts>

Once this file is compiled, it will generate an executable file of the same name but with the extension \*.dtbo

dtbo stands for device tree blob or binary overlay.

Where this file must be copied on the host platform depends on the platform. For the raspberry pi, the compiled \*.dtbo executable must be copied into /boot/overlays directory on the Pi.

For more info regarding the device tree, see the link below

[http://elinux.org/Device\\_Tree\\_Reference](http://elinux.org/Device_Tree_Reference)

Porting the ALSA driver into the Pi

The ALSA driver framework for the SDK consists of two sub-drivers. An ALSA Codec class driver and a Machine class driver.

#### *VPROC ALSA Codec driver*

*An ALSA codec is platform independent and contains audio controls, audio interface capabilities, codec Dynamic Audio Power Management (DAPM) definition and codec read/write access functions.*

The Codec driver for the VPROC SDK can be compiled as a simple generic codec driver, which provides no audio control, DAPM or codec control functions. Or, a fairly more complex driver that provides control functions in order to control certain features of the underlying ZL380xx codec device and aspect of the audio.

The compilation method of the driver can be specified via the following variable of the Makefile.globals.

VPROC\_CODEC\_MIXER\_ENABLE=no

If this variable is set to yes, the codec driver will be compiled to include Audio controls and Codec control functions that implements an ALSA mixer. Otherwise, these functions will not be included.

The code is located here:

/RELEASE\_ZLS38100\_PX\_Y\_Z/platform/raspberry/driver/sound/lxalsa/soc/c  
odec/zl380xx\_codec.c

Basically the Pi driver framework needs to be told how to register the ZL380xx codec driver. For the codec driver to be dts registrable, then it must include a matching table that includes the compatibility match. This is specified by the following Linux kernel device API structure struct of\_device\_id

```
static const struct of_device_id zl380xx_of_match[] = {
    { .compatible = "ms,zl380xx", },
    {}
};
```

That compatible line within that structure must be added into the Raspberry Pi dts file so that it knows how to register this Codec driver.

### VPROC ALSA Machine driver

*The machine driver acts as the glue that describes and binds the Platform and the Codec component drivers together to form an ALSA "sound card device". It handles any machine specific controls and machine level audio events (Ex: turning on an amp at start of playback, etc.).*

The machine code for the VPROC SDK is based on example machine code provided with the Raspberry Pi for the bCM2837 platform. Remember, the only necessary changes to that example code for compatibility with the VPROC SDK as per the codec DAI and name defined within the Codec driver are specified below in **red**. Anything else even if the naming refers to microsemi is just renaming of that existing machine driver function for contextual purpose.

```
static struct snd_soc_dai_link snd_microsemi_dac_dai[] = {
    {
        .name                = "Microsemi DAC",
        .stream_name         = "Microsemi DAC ",
        .cpu_dai_name         = "bcm2708-i2s.0",
        .codec_dai_name       = "z1380xx-dai",
        .platform_name        = "bcm2708-i2s.0",
        .codec_name           = "z1380-codec",
        .dai_fmt              = SND_SOC_DAIFMT_I2S | SND_SOC_DAIFMT_NB_NF |
                                SND_SOC_DAIFMT_CBS_CFS,
        .ops                  = &snd_microsemi_dac_ops,
        .init                 = snd_microsemi_dac_init,
    },
};
```

The `.dai_fmt` variable in the above structure specifies whether the ZL380xx codec will be the I2S master or the slave.

If the ZL380xx is to be the I2S master, then set it as per below

```
.dai_fmt          = SND_SOC_DAIFMT_I2S | SND_SOC_DAIFMT_NB_NF |
                    SND_SOC_DAIFMT_CBM_CFM,
```

Otherwise, set it to

```
.dai_fmt          = SND_SOC_DAIFMT_I2S | SND_SOC_DAIFMT_NB_NF |
                    SND_SOC_DAIFMT_CBS_CFS,
```

And for dts registration the following code is added into the Machine driver in order to register it with the PI using dts as per the matching compatibility line "**microsemi,microsemi-dac**". As previously mentioned, that compatibility matching line can be anything as long as you add that same anything in the dts file of the host platform.

```
static const struct of_device_id snd_microsemi_dac_of_match[] = {
    { .compatible = "microsemi,microsemi-dac", },
    {},
};
```

```
};
```

As you may already figure out, since it is said in the definition of a machine driver its main purpose is to bind the ALSA platform driver to the Codec driver. Therefore that dai\_link structure member variable below simply specifies the related callback variables defined within:

The Codec driver :

```
.codec_dai_name = ?  
.codec_name     = ?
```

The Platform driver :

```
.cpu_dai_name = ?  
.platform_name= ?
```

Now, it worth mentioning that the format for specifying the codec\_name differs from older kernel to newer kernel. This will be discussed in the troubleshooting chapter.

The `codec_dai_name` pertains to the naming instance of that ALSA API **structure struct snd\_soc\_dai\_driver** within the Codec driver. Every Codec driver must include an initialized definition instance of that structure. (See the initialized instance of that structure within the `zl380xx_codec.c` code)

The `codec_name` pertains to the name given to that Codec driver. That name is specified within the ALSA driver type structure definition. An ALSA driver can be of type platform, or SPI or I2C. The type I2C ALSA driver type pertains solely to the Codec driver class, since only the Codec driver may need to access the actual codec device over SPI or I2C if it implements ALSA driver features that require access to the underlying device over SPI or I2C.

That name must be specified within the ALSA API structure **struct platform\_driver**, or if using the I2C or SPI type **struct i2c\_driver** or **struct spi\_driver** respectively.

### Modify the dts for the Machine and Codec drivers Registration with the PI

Below is the example dts overlay to modify the related section of that Pi dts file to add the matching compatibility lines defined within the related Codec and Machine driver codes. The order lines within that dts file are standard dts coding to specify which SOC and interface of that SOC the dts modification pertains to.

This simple dts overlay, simply defines the matching compatibilities between the drivers to register with each other. Basically, during the driver insmod any drivers found with these 3 compatible match words: `brcm`, `microsemi` and `ms` will register with each other.

```
/dts-v1/;  
/plugin/;  
  
/ {  
    compatible = "brcm,bcm2708";  
    fragment@0 {
```

```
        target = <&sound>;
        __overlay__ {
            compatible = "microsemi,microsemi-dac";
            i2s-controller = <&i2s>;
            status = "okay";
        };
    };
    fragment@1 {
        target = <&i2s>;
        __overlay__ {
            status = "okay";
        };
    };
    fragment@2 {
        target-path = "/";
        __overlay__ {
            z1380-codec {
                #sound-dai-cells = <0>;
                compatible = "ms,z1380xx";
                status = "okay";
            };
        };
    };
};
```

## Porting the SSL into the Pi

The user of the SDK must implement the following four functions of the SSL. The file for the SSL is located

/RELEASE\_ZLS38100\_PX\_Y\_Z/platform/raspberry/driver/ssl/hal\_port.c

The Implementation of this file is complete for a Linux platform, therefore, there is nothing to be done in the currently implementation of this file in order to port to the Raspberry Pi platform or any other Linux platform.

But, for non-Linux platform the following four functions must be implemented accordingly to the mutual exclusion mechanism provided by the OS of that platform.

## SSL\_lock\_create

The VPROC SDK defined prototype for the lock creation is

```
ssl_status_t SSL_lock_create(ssl_lock_handle_t *pLock,  const char
*pName, void *pOption);
```

The implementation of this can simply be a place holder to initialize the type of mutual exclusion provided by the OS. If no such initialization is required, then the implementation of this function can be a simple return statement as per below

```
return SSL_STATUS_OK;
```

### SSL\_lock

The VPROC SDK defined prototype for the lock creation is

```
ssl_status_t SSL_lock(ssl_lock_handle_t lock_id, ssl_wait_t wait_type);
```

The implementation of this function must include the necessary code to prevent an in-progress HBI transaction from being perturbed. For platform that does not have an OS, then the implementation of this function can be a simple return statement as per below

```
return SSL_STATUS_OK;
```

### SSL\_unlock

The VPROC SDK defined prototype to remove a previous lock is

```
ssl_status_t SSL_unlock(ssl_lock_handle_t lock_id);
```

The implementation of this function must include the necessary code to remove a previously applied lock. For platform that does not have an OS, then the implementation of this function can be a simple return statement as per below

```
return SSL_STATUS_OK;
```

## Compile the SDK drivers

The SDK is now ready to be compiled. To compile the SDK, simply cd into the root folder of the SDK and issue:

```
cd RELEASE_ZLS38100_PX_Y_Z  
make hbilnx HBI=SPI
```

*Note: to compile the SDK for I2C, simply replace SPI with I2C in the above command.*

## Compile the SDK demo Apps

The SDK includes four example demo applications that are not automatically compiled by the compilation of the SDK. Therefore, these apps can individually be compiled using the commands below.

```
make apps HBI_HELLO=1  
make apps HBI_TEST=1  
make apps HBI_LOAD_FIRMWARE=1  
make apps HBI_LOAD_GRAMMAR=1
```

## Compile MiTuner Bridge Server

MiTuner (ZLS38508 and ZL38508LITE) is a PC application running on Microsoft Windows. Historically, MiTuner connected to the Timberwolf via UART but it can now also connect via a network socket if a

MiTuner Bridge Server is running on the host. MiTuner is needed to initially tune the device and not having to wire out the UART helps to save time and preserve the acoustic of the product.

```
make bridge
```

Once compiled, the executable can be found here:

```
/RELEASE_ZLS38100_PX_Y_Z/apps/C/mituner_bridge/mituner_bridge_server
```

*Note: As MiTuner Bridge Server will open its own instance of the HBI driver, you have to make sure that enough instances of the HBI driver can access the Timberwolf. Increase [HBI\\_MAX\\_INST\\_PER\\_DEV](#) in `Makefile.globals` if needed.*

*Note: By default, MiTuner Bridge Server connects to first Timberwolf (device ID 0). The device ID can be changed in `mituner_bridge_server.h` by updating `DEVICE_ID` if needed.*

### Compile the Firmware Converter Tool

As mentioned in earlier chapters of this document, the \*.s3 and \*.cr2 firmware and configuration images included in the ZLS380xx firmware package cannot be loaded into the device as is with the VPROC SDK. These images must first be converted into \*.bin files using the

```
/RELEASE_ZLS38100_PX_Y_Z/tools/twConvertFirmware2c.c
```

Or, the Python equivalent

```
/RELEASE_ZLS38100_PX_Y_Z/apps/Python/tw_firmware_converter.py
```

To compile the C version of the converter tool, simply from a terminal window issue the command sequence below:

```
cd RELEASE_ZLS38100_PX_Y_Z/tools
```

```
gcc twConvertFirmware2c.c -o twConvertFirmware2c
```



## SDK Testing and Debug

If the VPROC SDK is successfully compiled as described in the "Porting" chapter of this document, then the following kernel \*.ko modules and \*.dtbo binaries will be created.

```
RELEASE_ZLS38100_PX_Y_Z/libs/lib/modules/`uname -r`/extra/hbi.ko
RELEASE_ZLS38100_PX_Y_Z/libs/snd-soc-microsemi-dac.ko
RELEASE_ZLS38100_PX_Y_Z/libs/snd-soc-zl380xx.ko
RELEASE_ZLS38100_PX_Y_Z/libs/microsemi-spi-multi-tw-overlay.dtbo
RELEASE_ZLS38100_PX_Y_Z/libs/microsemi-spi-multi-tw-dt-overlay.dtbo
RELEASE_ZLS38100_PX_Y_Z/libs/microsemi-i2c-multi-tw-dt-overlay.dtbo
RELEASE_ZLS38100_PX_Y_Z/libs/microsemi-dac-overlay.dtbo
```

The Apps compilation will generate the following 4 executables

```
RELEASE_ZLS38100_PX_Y_Z/apps/C/hbi_hello
RELEASE_ZLS38100_PX_Y_Z/apps/C/hbi_test
RELEASE_ZLS38100_PX_Y_Z/apps/C/hbi_load_firmware
RELEASE_ZLS38100_PX_Y_Z/apps/C/hbi_load_grammar
```

The firmware converter tool compilation will generate the executable below

```
RELEASE_ZLS38100_PX_Y_Z/tools/twConvertFirmware2c
```

## Install the kernel modules and configure the Pi

Load these modules into the Raspberry Pi platform as described below

1. Copy the 3 \*.ko files into the following location on the Pi

Create a directory to store the \*.ko modules

```
sudo mkdir /usr/local/my_modules
```

```
cd RELEASE_ZLS38100_PX_Y_Z/libs/lib/modules/`uname -r`/extra/
sudo cp hbi.ko /usr/local/my_modules
cd RELEASE_ZLS38100_PX_Y_Z/libs
sudo cp *.ko /usr/local/my_modules
```

Edit the ~/.profile file on the Pi to add the lines below at the end of the file

```
sudo insmod hbi.ko
sudo insmod snd-soc-microsemi-dac.ko
sudo insmod snd-soc-zl380xx.ko
```

And at a terminal type the command below to reboot the Pi.

```
reboot
```

2. Copy the \*.dtbo files into the following location on the Pi  
/boot/overlays

```
cd RELEASE_ZLS38100_PX_Y_Z/libs
sudo cp *.dtb /boot/overlays
```

3. Edit the /boot/config.txt on the Pi as per below

```
sudo nano /boot/config.txt
```

- a. Un-comment (Remove the #) the line below within it to enable the SPI and I2S  
#dtparam=i2s=on

- #dtparam=spi=on
- b. Add the following lines at the end of that file
  - dtoverlay=i2s-mmap
  - dtoverlay=microsemi-spi-multi-tw-overlay
  - dtoverlay=microsemi-dac-overlay
- c. Close and save the file
  - Press **CTRL+x**, then press **y**

- 4. Reboot the Pi.
  - sudo reboot

Once the Pi is re-booted a sound card under the name `sndmicrosemidac` will be created under `/proc/asound`

And two SPI drivers named `hbi0` and `hbi1` will be created under `/dev`

### Install the Demo Apps

Copy the executables of the demo apps into the `/usr/local/bin` directory on the pi

```
cd RELEASE_ZLS38100_PX_Y_Z/apps
sudo cp hbi_test /usr/local/bin
sudo cp hbi_load_firmware /usr/local/bin
sudo cp hbi_load_grammar /usr/local/bin
```

### Install the Firmware Converter Tool

Copy the `twConvertFirmware2c` executable into the `/usr/local/bin` directory on the pi

```
cd RELEASE_ZLS38100_PX_Y_Z/tools
sudo cp twConvertFirmware2c /usr/local/bin
```

### Testing the SDK

With the drivers and apps installation above, both audio playback and recording and SPI access to the ZL380xx devices can be performed as described in the next two sections below.

### ZL380XX access over SPI using the Demo Apps

- 1. Load a firmware and a related configuration image into the ZL380xx
  - First use the `twConvertFirmware2c` tool convert the `*.s3` and `*.cr2` into `*.bin`
  - Simply copy the the `*.s3` and `*.cr2` to a desired location within the Pi. See the example command below on how to use the tool to convert a `*.s3` and `*.cr2` file
  - Example:
    - Let's say I have a firmware `*.s3` file named `Microsemi_ZLS38063.1_E0_10_0_firmware.s3` and a configuration `*.cr2` file

generated from the Microsemi MiTuner tool named  
Microsemi\_ZLS38063.1\_E0\_10\_0\_config.cr2 that are located in a directory named  
/home/pi/my\_images on the Pi. To convert the files to \*.bin issue the following command  
sequence from a terminal on the Pi.

```
cd /home/pi/my_images
twConvertFirmware2c -i Microsemi_ZLS38063.1_E0_10_0_firmware.s3 -o
ZLS38063.1_E0_10_0_firmware.bin -b 16 -f 38063

twConvertFirmware2c -i Microsemi_ZLS38063.1_E0_10_0_config.cr2 -o
ZLS38063.1_E0_10_0_config.bin -b 16 -f 38063
```

The following two binary files will be created within the /home/pi/my\_images  
ZLS38063.1\_E0\_10\_0\_firmware.bin  
ZLS38063.1\_E0\_10\_0\_config.bin

To load these 2 binary images into the ZL380xx device Id 0 without saving them to a slave flash  
controlled by the ZL380xx device, simply at a terminal on the Pi issue the command below  
cd /home/pi/my\_images

```
hbi_load_firmware -d 0 -i ZLS38063.1_E0_10_0_firmware.bin -c
ZLS38063.1_E0_10_0_config.bin
```

To Load both images into the device, and optionally save them to flash.

```
hbi_load_firmware -d 0 -i ZLS38063.1_E0_10_0_firmware.bin -c
ZLS38063.1_E0_10_0_config.bin -s
```

**Note:** the device Id is as per the index of the `sdk_board_devices_info[]` structure array  
defined in chapter “[Porting the SPI driver into the Pi](#)”. To access device at device ID 1, simply  
replace the number 0 following the `-d` into commands above to 1

2. To Read, write specific registers of the ZL380xx device use the `hbi_test` demo apps as per the  
example commands below

Example:

To write the 0x1234, 0x5678 into register 0x00C of the ZL380xx device at Id 0

```
hbi_test -d 0 -w 0x00C 0x1234 0x5678
```

To read 2 16-bit (4 8-bit) words from register 0x00C of that same device

```
hbi_test -d 0 -r 0x00C 4
```

Expected read result from that register 0x00C will be:

```
0x000C = 0x1234
```

0x000E = 0x5600

**Note:** Register 0x000E of the ZL380xx is a special register, whatever a host writes to this register will be zeroed out by the ZL380xx firmware, in order to confirm to the host that the command has been received correctly. Therefore this register is a good way to verify that the host is accessing the device and the device is working properly.

## ZL380XX access over SPI using the procfs

The VPROC SDK implements a PROC file system that includes support for all the features supported by the SDK. The procfs commands implemented by the SDK are described below:

### open\_device

The open\_device command opens an instance of the hbi driver for a specific ZL380xx device. This command call is optional and no longer needed, by default an instance of each of the ZL380xx driver is already opened when the hbi.ko driver is loaded. But, if you wish to close and re-open the instance of the driver using the close\_device and open\_device procfs commands below in a terminal on the host platform.

In brief, in the latest P3.0.0 version of the SDK, you don't need to do a procfs open\_device in order to use the other procfs commands, since by default this is already done when the driver is loaded into the platform. The driver will automatically open and create a dev\_xx instance under /proc/hbi/ for each of the ZL380xx devices.

The first x is the SPI or I2C bust number, the second x is the SPI chip select number or the I2C address in hexadecimal.

So for the example `sdk_board_devices_info[]` structure array defined in chapter "[Porting the SPI driver into the Pi](#)" the following two procfs devices will be created for SPI bus 0 CS 0, and SPI bus 0, CS 1

```
/proc/hbi/dev_00
```

```
/proc/hbi/dev_01
```

Example, to open device at SPI bus 0 with slave select 0  
`echo 0:0 > /proc/hbi/open_device`

The command syntax is as follows:

```
echo bus_num:dev_addr(in hex) > /proc/hbi/open_device
```

### close\_device

To close an opened proc fs instance of the driver for a specific ZL380xx device issue the command below.

Example, to close device at SPI 0, CS 0 enter  
`echo 0:0 > /proc/hbi/close_device`

#### write\_reg

To perform the same register write example performed in the [ZL380XX access over SPI using the Demo Apps](#) section of this document, simply issue the command below:

```
echo 000C 12345678 > /proc/hbi/dev_00/write_reg
```

*Note: the number arguments following the echo are in hexadecimal..*

#### read\_reg

To perform the same register read example performed in the [ZL380XX access over SPI using the Demo Apps](#) section of this document, simply issue the command below

```
echo 000C 4 > /proc/hbi/dev_00/read_reg
```

You can view the results by issuing either one of the command below

```
cat /proc/hbi/dev_00/read_reg
```

Or

```
dmesg | tail -5
```

*Note: the number arguments following the echo are in hexadecimal.*

#### load\_fw

To load a desired firmware image into the ZL380xx device 0 using the procfs, simply issue the command below.

Example: to load the same firmware file discussed in the [ZL380XX access over SPI using the Demo Apps](#) section of this document

```
cat ZLS38063.1_E0_10_0_firmware.bin > /proc/hbi/dev_00/load_fw
```

#### cfgrec

To load a configuration record into the ZL380xx device 0, simply issue the command below.

Example: to load the same configuration file discussed in the [ZL380XX access over SPI using the Demo Apps](#) section

```
cat Microsemi_ZLS38063.1_E0_10_0_config.cr2 > /proc/hbi/dev_00/cfgrec
```

#### flash\_save\_fwrcfgrec

The firmware and optional configuration record currently loaded into the device memory before starting the firmware can be saved into a slave flash controlled by the ZL380xx device using the command below.

```
cat /proc/hbi/dev_00/flash_save_fwrcfgrec
```

**Note:** *this feature is only supported if the device is in boot mode and a firmware image and optional configuration have been loaded into the device memory but a `start_fw` command has not been issued yet to start running the image.*

#### `start_fw`

Once the `load_fw` and the optionals `cfgrec`, and `flash_save_fwrcfgrec` are completed, then the execution of that firmware image currently stored into the internal memory of the ZL380xx device must be stated using the command below.

```
cat /proc/hbi/dev_00/start_fw
```

#### `flash_load_fwrcfgrec`

To load a firmware that is currently stored in a slave flash controlled by the ZL380xx device, simply issue this command below.

Example: to load an image currently stored in index 1 of the flash.

```
echo 1 > /proc/hbi/dev_00/flash_load_fwrcfgrec
```

Example: to load an image currently stored in index 2 of the flash.

```
echo 2 > /proc/hbi/dev_00/flash_load_fwrcfgrec
```

#### `flash_erase`

The memory of the flash can be erased using the following command.

To erase the whole flash

```
cat /proc/hbi/dev_0/flash_erase
```

To erase just a specific index of the flash. Example to erase index 1

```
echo 1 > /proc/hbi/dev_00/flash_erase
```

**Note:** if the flash currently stores multiple images

Ex: images 1, 2, 3 in slots 1,2,3 respectively, deleting the image at flash slot 1, will simply delete the image on the flash, but that slot will not be reusable until the images at slot 2 and 3 are deleted .

Therefore, this feature should only be used for scenario where the image to delete is the last image on the flash.

## Play and Record Audio with the ZL380xx

The ZL380xx ALSA drivers installed above support above recording and playback of audio from sampling rates of 8KHz up to 48KHz. The ZL380xx can be configured as an I2S master or slave. The ZL380xx I2S master slave configuration in the configuration record must be set in accordance to the [.dai fmt](#) configuration in the Machine driver. By default the SDK is configured to have the ZL380xx device behaves as a slave I2S device.

### Record a wav file

Example: to record speech from the Mic at a sampling rate of 16KHz of the ZL380xx device and save it to a file named test16KHz.wav

```
arecord -D hw:sndmicrosemidac,0 -c 2 -f S16_LE -r 16000 test16KHz.wav
```

### Play a wav file

Example: to play a file named test16KHz.wav at a sampling rate of 16KHz

```
aplay -D hw:sndmicrosemidac,0 -c 2 -f S16_LE -r 16000 test16KHz.wav
```

or simply,

```
aplay -D hw:sndmicrosemidac,0 test16KHz.wav
```

As you can see above, both recording are done in stereo, this is specified by the option `-c` in the command above. But, what if you don't want to record/play in stereo, and prefer to record/play only the channel where speech is detected?

What if the Platform I2S driver does not support playing mono audio?

Then, you must configure ALSA so that ALSA will always downmix incoming/outgoing audio to a single channel. You can do that by configuring the ALSA configuration file on the host platform as per below:

Copy the following script into either or both of the following files `~/.asoundrc` or `/etc/asound.conf` on the host platform.

```
# ---ALSA configuration for dmix plugin---

pcm.dmixed {
    ipc_key 1025
    type dmix
    slave {
        pcm "hw:sndmicrosemidac,0"
        channels 2
    }
}

pcm.dsnooped {
    ipc_key 1027
```

```
type dsnoop
slave {
    pcm "hw:sndmicrosemidac,0"
    channels 1
}
}

pcm.asymed {
    type asym
    playback.pcm "dmixed"
    capture.pcm "dsnooped"
}

# make the sndmicrosemidac the default sound card
pcm.!default {
    type plug
    slave.pcm "asymed"
}

ctl.!default {
    type hw
    card sndmicrosemidac
}
#-----END-----
```

The above will force the `sndmicrosemidac` to be the default sound card for that platform. And also will support the downmix of audio to a single audio channel.

So with the above, I can record a single audio channel as per below

```
arecord -c 1 -f S16_LE -r 16000 test16Khz.wav
```

or simply,

```
arecord -f S16_LE -r 16000 test16Khz.wav
```

I can play a single audio channel as per below

```
aplay -c 1 -f S16_LE -r 16000 test16Khz.wav
```

or simply,

```
aplay -f S16_LE -r 16000 test16Khz.wav
```



## Troubleshooting

Although compiling and porting the SDK should be a seamless process, this chapter will go through some steps to help debugging issues that may be observed if any during the porting and the use of the SDK.

### Compilation Debug

Although error generated during the compilation process will at least provide some clues for the root cause of the errors, very often error reported by the compile process will likely be due to incorrect setting of the mandatory pre-compile variables (KSRC, TOOLSPATH, CROSS\_COMPILE, ARCH) within the root Makefile.globals of the SDK or an improper toolchain.

#### Possible Compilation error 1

*"make[x]: \*\*\* No targets specified and no makefile found. Stop."*

X: is an integer number

That error will be succeeded by other errors since the SDK will fail to compile. Since the SDK includes all necessary makefiles, then the only other makefile needed for the VPROC compilation is from whatever kernel source or header with which to compile the SDK. This error is an indication that no kernel headers are found in the path provided to the VPROC SDK to where to find the kernel headers. Make sure that the path on the development machine or the network to where the kernel headers are located is correct.

#### Possible Compilation error 2

*gcc: error: unrecognized argument in option ...*

*or*

*gcc: error: unrecognized command line option ..*

This is an indication that the toolchain being used to compile the SDK is improper for that controller ARCH. Meaning the compiler found in the provided tool chain does not match that CPU architecture defined in the makefile. Make sure to set the TOOLSPATH, CROSS\_COMPILE, ARCH variables in the makefile accordingly and make sure a matching compiler is found for the ARCH for which to compile the SDK. Or, make sure the ARCH is set in accordance to the compiler.

Toolchain and compiler for different architecture can be found here. However, make sure to confirm with the CPU vendor on which toolchain to use to compile their SDK and code that needs to be ported into their platform.

<https://releases.linaro.org/components/toolchain/binaries/5.1-2015.08/>

### Possible Compilation error 3

*make[x]: \*\*\* /vproc\_sdk/platform/xx/: No such file or directory. Stop.*

That's an indication that the make command to build the VPROC SDK was issued not within the root folder of the SDK, but within one of the subfolders of the SDK, or make was issue by a user with not enough permission to compile the SDK. Make sure to be at the root folder of the SDK when issuing the command and the user has the appropriate permission.

### Possible Compilation error 3

You got some warnings such as the one below followed by compilation failure.

*"warning: incompatible implicit declaration of"*

*"Makefile:xx: recipe for target 'xyz' failed"*

*make: \*\*\* [vproc\_sdk] Error 2*

That's an indication that some of the platform dependent typedef variables or includes are not supported by the compiler being used to compile the SDK. Therefore, make sure the typedefs and or standard C/platform library includes defined within the `typedefs.h` file at the path below are in accordance to that platform.

`/RELEASE_ZLS38100_PX_Y_Z/platform/.../include/typedefs.h`

### Loading the driver/apps debug

Once the drivers are compiled, depending on the platform they must be loaded/executed into the platform. As described in the debug steps above, the compilation of the SDK will result into 3 or more executables. These executables must be running into the host platform. Below are info to help in resolving possible issues if any during this process.

### Possible Issue 1

You got a stack limit error with a bunch of numbers which looks like a kernel crash.

```
[ 108.968038] PC is at hbi_drv_init+0x318/0x420 [hbi]
[ 108.968052] LR is at irq_work_queue+0x14/0x90
[ 108.968068] pc : [<7f556318>] lr : [<801f62cc>] psr: 60000013
sp : ab139d68 ip : 00000007 fp : ab139dbc
[ 108.968081] r10: 00000002 r9 : 7f55119c r8 : 00000000
[ 108.968094] r7 : 7f5511a0 r6 : 7f5511a0 r5 : 7f550890 r4 : 7f54e90c
[ 108.968104] r3 : 00000000 r2 : 00000000 r1 : 00000007 r0 : 00000040
[ 108.968118] Flags: nZCv IRQs on FIQs on Mode SVC_32 ISA ARM Segment user
[ 108.968125] Control: 10c5383d Table: 2b1c406a DAC: 00000055
[ 108.968130] Process insmod (pid: 1175, stack limit = 0xab138210)
[ 108.968137] Stack: (0xab139d68 to 0xab13a000)
[ 108.968154] 9d60: 00000000 00000000 00000000 80c6edc0 00000000 00000000
[ 108.968173] 9d80: b6230e00 00000000 80c06900 00000000 00000000 7f54eec0 7f556000 ab092400
```

```
[ 108.968190] 9da0: 7f54eec0 00000001 54f6dfdc 00000000 ab139e3c ab139dc0 80101bf0 7f55600c  
[ 108.968208] 9dc0: 00000000 00000000 ab139e14 ab139dd8 802136a8 393eb000 8024e840 .....
```

This is an indication that the driver s (controller and slave) are not registered together. Therefore, when the slave driver is being loaded, if it can't register with the controller as per the host interfaces specified in the driver, then it will generate a Null driver error as per above. Make sure to set the dts configuration to register that driver accordingly. If the host interfaces are being used by other drivers, then trying to register a driver to that same host interface will generate that error.

#### Possible issue 2

Although the SDK was compiled successfully, the driver generated an error as per below when trying to execute them into the host platform.

*"error could not insert module operation not permitted"*

This error will be generated, if the module was compiled for a different ARCH than the one for which it is being executed. Or, the logged platform user trying to execute the module does not have the right permission to do so.

#### Possible issue 3

Trying to insmod the compiled SDK module resulted in this error below

*insmod: ERROR: could not insert module hbi.ko: Invalid module format*

That error is generated if the kernel on the target platform to which the module is being installed is not the same as the kernel with which the module was built. Make sure the kernel used to compile the SDK is the absolutely exact same kernel version running on the target platform.

Example:

Let's say the SDK was compiled on kernel 4.4.38, but the target platform is running 4.4.38-generic.

Although both are of kernel base 4.4.38 but these are not the same kernel version, because there may be differences in the kernel headers needed to compile modules.

#### Possible issue 4

On my Linux platform, trying to load the ALSA kernel modules resulting from the compilation of the SDK generated the error below when trying to insmod them.

```
[ xxxxxx] snd-microsemi-dac soc snd_soc_register_card() failed: -517
```

This error indicates that a proper ALSA Codec driver was not loaded, or the ALSA driver type structure defined within the Codec driver is not the appropriate type.

As described in the Codec porting section of that guide, the codec driver type structure can be of either **struct platform\_driver**, **struct i2c\_driver** or **struct spi\_driver**. ALSA machine

driver for platform based on older Linux kernel 2.6.x may expect the driver type to be of `struct i2c_driver` or `struct spi_driver`. The reason is the older kernels use a registration method for ALSA drivers that require the `.codec_name` to be passed not only the codec name but also some info regarding the communication interfacing of the Codec to that host controller.

Example: If the Codec is interfaced to the host via SPI using SPI bus 0 and address 0

```
static struct snd_soc_dai_link xyz_dai_link = {
    .name = "ZL38012",
    .stream_name = "ZL38012-STREAM",
    .cpu_dai_name = "ambarella-i2s.0",
    .platform_name = "ambarella-pcm-audio",
    .codec_dai_name = "zl380xx-hifi",
    .codec_name = "spi0.0",
    .init = a5sevk_ak4642_init,
    .ops = &a5sevk_board_ops,
};
```

Or, if the Codec is interfaced to the host via I2C using I2C bus 0 and address 0x45

```
static struct snd_soc_dai_link xyz_dai_link = {
    .name = "ZL38012",
    .stream_name = "ZL38012-STREAM",
    .cpu_dai_name = "ambarella-i2s.0",
    .platform_name = "ambarella-pcm-audio",
    .codec_dai_name = "zl380xx-hifi",
    .codec_name = "zl380xx.0-045",
    .init = a5sevk_ak4642_init,
    .ops = &a5sevk_board_ops,
};
```

Where: "**zl380xx.0-045**": zl380xx is the name given to the codec driver 0-045 is the I2C bus and address

## Audio Playback/Recording Debug

### Possible Issue 1:

Although the driver is loaded properly and sound card is created, but when performing the playback and recording test of the Test section of this document, although the action seems to be performed with no error, but no sound is heard, or no audio is recorded.

That's an indication of an improper device configuration:

- Verify that the ZL380xx codec (SOUT, ROUT) mute function is not activated in the device current configuration
- Gain level (ROUT, SOUT) are not set to low
- Improper ZL380xx cross-point switch causing audio not to be routed to the desired interfaces
- The `.dai_fmt` setting in the ALSA Machine driver specifies that the Codec is the clock master, while Codec is configured as slave. Make sure that the `.dai_fmt` is set accordingly. (See the [ALSA Machine description](#) in the Porting section).

#### Possible Issue 2:

I keep getting *underrun!!! (at least x.xx ms long)* when trying to do *aplay* or *arecord*, why?

That basically means the host or the codec (Depending which device is generating the I2S clock BCLK) is not generating a precise clock. A drifting clock will cause that error. If the platform uses a PLL that uses a divider to provide all the different clocks, make sure that divider is set accordingly for the I2S clock generation. This is set in the Machine driver function that configures the hardware parameters to generate the appropriate clocks, etc.

#### Possible Issue 3:

got a *"Playing WAVE 'xyz.wav' : Signed 16 bit Little Endian, Rate 16000 Hz, Stereo  
aplay: pcm\_write:1737: write error: Input/output error"* when trying to do *aplay*?

That's an indication that you are trying to do something not supported by your Codec driver. If you take a look in the Codec driver code within the VPROC SDK, one of the mandatory definitions within an ALSA codec driver is an initialized instance definition of the structure below

```
struct snd_soc_dai_driver
```

This is where the Codec must specify the type of audio stream (Playback, Capture, minimum and maximum number of channels, sampling rates for each stream) to support. If playback stream support is not defined within this structure then whenever the host issues an *aplay* command that error will be generated.

```
/RELEASE_ZLS38100_PX_Y_Z/platform/raspberry/driver/sound/lnxalsa/soc/c  
odec/zlx380xx_codec.c
```

The imprecision of the I2S clock discussed in previous possible error can cause this error as well.

### SPI/I2C Communication error

#### Possible error 1

```
[HBI_open:84]Opening file /dev/hbi0  
Err 0x2 in HBI_OPEN  
dev open error
```

This is an indication that the application is trying to open a ZL380xx device but the hbi.ko driver is not loaded into the kernel. Or, all instances of the driver for that ZL380xx device are already in use. Make sure to close previously opened instances of the driver for that device.