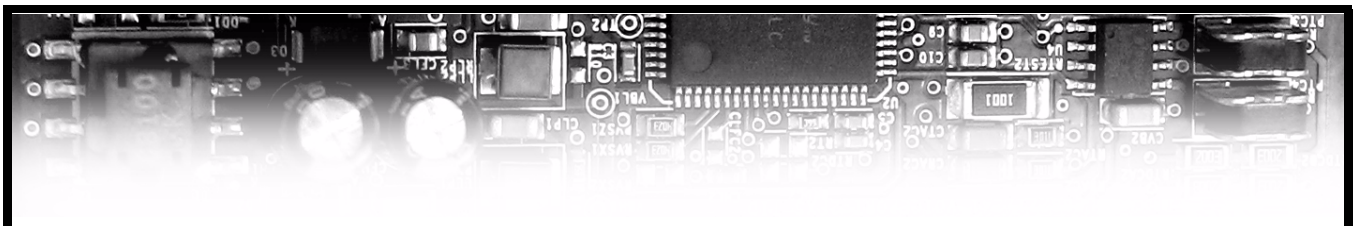


VoicePath™ API-II

Porting to the Linux Kernel

Part Number:	**580, 790, 880, 890 Series**
Revision Number:	**A2**
Issue Date:	**January 2007**

ZARLINK SEMICONDUCTOR

FEATURING Legerity® VOICE SOLUTIONS

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1–7
1.1 Purpose 1–7
1.2 Overview 1–7
1.2.1 API-II and Application View 1–7
1.2.2 Linux Overview 1–8
1.2.3 API-II and Linux Kernel Overview 1–8
1.2.4 Memory Management 1–8
1.2.5 Use of Profiles 1–10

CHAPTER 2 LINUX DEVICE DRIVERS 2–1
2.1 Major and Minor Numbers 2–1
2.1.1 Creating a File Descriptor 2–1
2.1.2 Using File Descriptors in the Application 2–1
2.1.3 File Descriptors as Line Contexts, and Accessing the Device 2–2
2.2 ioctl() 2–2
2.2.1 User Space API-II Functions 2–3
2.3 Init, Open, Close, Exit 2–3

CHAPTER 3 REFERENCES 3–1

CHAPTER

1 INTRODUCTION

1.1 PURPOSE

This guide is intended to assist anyone using the API-II in a Linux Kernel. It accompanies the API-II library files for the VE790 and VE880 devices (Conventional SLACs).

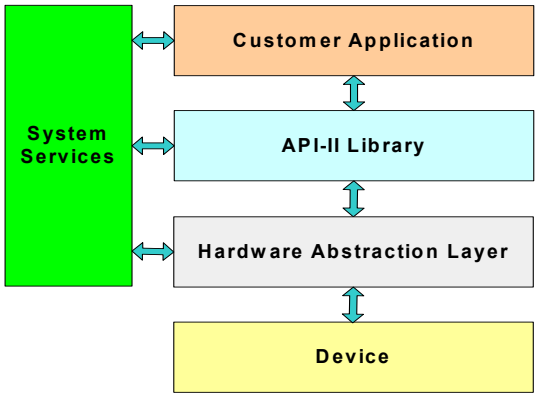
1.2 OVERVIEW

The API-II library is designed to provide a non-OS specific software implementation of the API-II Specification for use in any software architecture. Porting of the API-II code to a non-OS or flat OS (where applications share the same memory as the OS) are somewhat straightforward and not primarily discussed in this document. Porting the API-II to Linux, where restrictions exist on applications, creates some problems that can be resolved in many ways. This document will discuss the tradeoffs and propose a solution for solving those problems.

1.2.1 API-II and Application View

The API-II library is a set of 'C' files that together provide a portable abstraction of Zarlink Semiconductor devices from the application. It relies on a user implemented Hardware Abstraction Layer (HAL), System Services, and upper level application code to create and manage memory used by the API-II. Figure 1–1 is provide to clarify.

Figure 1–1 API-II and Application View



This organization means that the API-II Library can be compiled and tested once, then linked with the remaining components that can be changed to support different hardware.

The recommendations of this document will aim at maintaining the key ideas of this architecture. Specifically:

- API-II Library (driver) is modular in that it contains no memory and is not affected by changes to the hardware, application, or system services.
- HAL/System Services are specific to the OS and Hardware and not affected by changes to the API-II library or application.
- The Customer Application is not affected by changes to the API-II Library or HAL, and affected by changes to System Services only if the application is using OS specific functionality.

1.2.2 Linux Overview

Linux is an Operating System (OS) that supports multi-threading (more than one application running at a time) and divides Applications from the Kernel. Specifically, an application cannot in general access memory used by the Kernel directly (although there are some mechanisms to allow it, which is discussed in Section 1.2.4). The advantage to this architecture is that a corrupt application cannot halt the entire system. The disadvantage is there is overhead associated with function calls and memory access between the application space and Kernel space.

1.2.3 API-II and Linux Kernel Overview

The recommendation of this document is that the API-II should be put in the Linux Kernel and not part of the application space. The following discussion explains.

The primary function (for Conventional SLAC devices) in the API-II is VpApiTick(). It manages all timing related operations from Caller ID generation, Cadencing (Tone and Ringing), to Dial Pulse detection. VpApiTick() must be run at a minimum 8.33mS to support Caller ID, and it is recommended to run at 5mS to support Dial Pulse detection. The problem this requirement creates for user mode Linux applications is that the standard Linux Kernel provides a 10mS resolution for applications. This can be changed at compile time but the cost of reducing the Kernel tick rate is there will be less available processor bandwidth for all applications. While that might be acceptable for a voice demo environment (running primarily the API-II), it is not generally acceptable or recommended for efficient production platforms.

Another problem running the core code of the API-II in user space is then the hardware access is from user space to kernel. During a given "tick" for a single device, there may be ten or more calls into the device. Such a situation is highly inefficient and could affect timing requirements and performance of running applications.

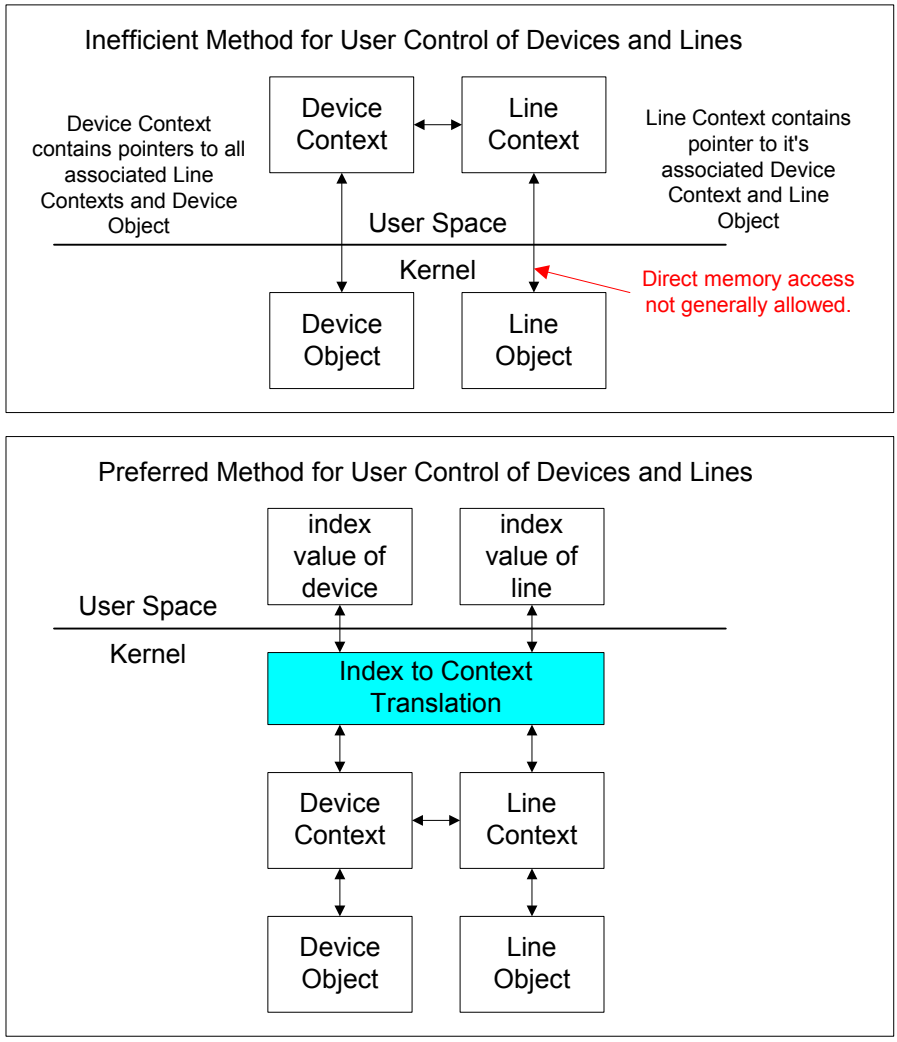
1.2.4 Memory Management

The issue of Memory Management in the Linux Kernel is too large of a topic to cover in this document. The interested reader should refer to "Linux Device Drivers", 2nd Edition, June 2001. This section will focus only on those components relevent to the API-II.

The API-II maintains portability by containing no memory that is not provided by the application or system. The problem this creates is that the Kernel (API-II code) cannot normally access user space memory directly. In fact, Linux provides a way for the Kernel to access user memory directly using the function get_user_pages(), but it is not recommended for character drivers, is complex to use, and does not always provide the efficiency one would expect. Given these comments and complexities, and the fact that most user defined date for the API-II is in the order of ten’s of bytes, it is recommended that the API-II Kernel driver be written such that the necessary data is copied to/from user space when needed. The specific functions recommended are copy_to_user() and copy_from_user() due to their simplicity, and ability to efficiently copy all data that would reasonably be used by the user application/API-II. Other operations are possible but will not be discussed in this document.

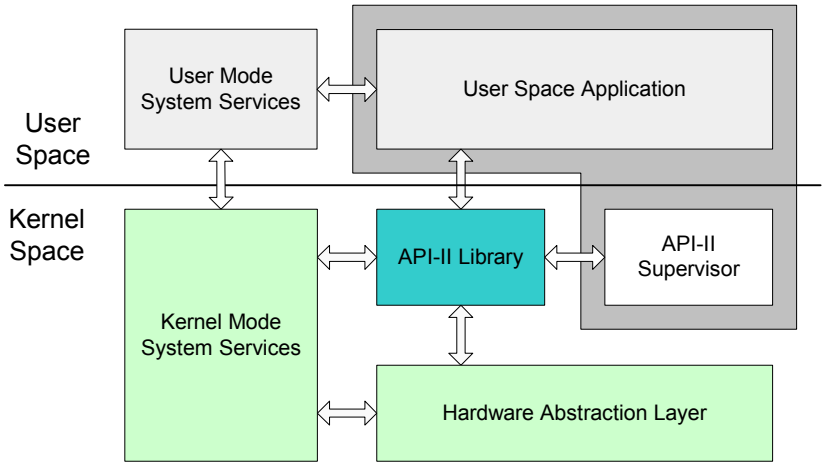
The last issue that needs to be discussed regarding memory management is that the basic concept in the API-II is that of device and line contexts and objects. These are instantiated by the application and (almost) every API-II function accepts either a device or a line context. But as stated above, the user space cannot contain a context that points to memory in the kernel directly. The recommendation is to use a feature of the Linux Kernel (minor number, discussed in Section 2.1) and provide an "index to context" mapping. This issue is shown in Figure 1–2.

Figure 1–2 Device and Line Context Pointers to Object Data



One of the goals of this recommendation is to maintain the overall organization and modularity of the API-II, System Services/HAL, and Application. As discussed above however, it is not reasonable to stricly maintain the organization shown in Figure 1–1 because of how the memory is organized. But the overall concept and modularity can be maintained by moving only the memory management aspect of the user application into the Kernel. Figure 1–3 is provided to explain.

Figure 1–3 API-II, System Services/HAL, API-II Supervisor, and User Application



The primary difference between Figure 1–1 and Figure 1–3 is the functional block "API-II Supervisor". The purpose of the API-II Supervisor is to:

- Allocate all Memory required by the API-II, and
- Provide a minor number to either device or line context mapping.

The required API-II functions that will be called in API-II Supervisor are: VpMakeDeviceObject(), and VpMakeLineObject() for each device and line in the system. This is no different than any other application requirement, but it's recommended to implement a functional block in the kernel to meet these requirements to reduce memory management complexity and overhead access between user space and kernel.

1.2.5 Use of Profiles

Profiles create memory management issues above that imposed by API-II Device and Line context and objects (which is easily handled alone by the API-II Supervisor discussed in Section 1.2.4). First, profiles are defined by the application so the API-II Supervisor "should not" predetermine the profiles as it can for the device and line contexts and objects. Device and Line contexts and objects are data that is controlled by the API-II, profiles are data that is controlled by the application. It's desireable to maintain that flexibility in the user application.

The second issue is that some profiles have an unknown length (whereas device and line contexts and object sizes are fixed) and therefore causes memory management issues (i.e., sizeof() is not directly appropriate). Every profile contains a length byte (location 3rd with index 0), so that value can be used to determine the memory requirements.

The last problem with Profiles is that some functions accept directly a Profile pointer and bypass the Profile Table.

CHAPTER

2 LINUX DEVICE DRIVERS

To discuss the solutions recommended in this document, a basic understanding of Linux Device Drivers is necessary. The intent of this chapter is to provide an overview of Linux Device Drivers as it affects use of the API-II only. A full discussion on Linux Device Drivers can be found in "Linux Device Driver", 2nd Edition.

2.1 MAJOR AND MINOR NUMBERS

All device drivers in the Linux Kernel are referred to by a major number and are passed a minor number. The Kernel takes no action on a minor number so conceptually, the driver is the entire API-II library and the minor number is an indicator that may be used by the driver (e.g., a line context pointer).

A driver may choose to ignore the minor number entirely, but this is not recomended for two reasons:

- More than one driver in the system can have the same major number, and
- It is an efficient use to indicate a line number. This will be discussed imediately.

Almost all API-II functions take a pointer to a VpLineCtxType to control a specific line, and some take both a VpDeviceCtxType and a VpLineCtxType. Most cases it is the line context that is important and only at initialization is passing a device context type of any significant value. Therefore, it is recommended that the minor number in general refer to the line (line context) and under special circumstances would a minor number refer to the device. It will help clarify to discuss the specifics with reference to the API-II functions affected.

2.1.1 Creating a File Descriptor

Linux works almost entirely on the concept of file descriptors, even when accessing real hardware. The application access the hardware through a special file in the Linux "/dev" directory that is created by the system. Each file descriptor can be of a certatin type; character, block, network, etc. The API-II driver should be declared as a character type of device because much of the data passed between the API-II and user application is "small" (typically in the range of 1 to 10 bytes). Each entry in the "/dev" directory must be given the type, major number, minor number, and a name of the entry that can be used by the application. An example of such an entry is as follows:

```
mknod voice0 -c 100 0
```

This creates "/dev/voice0", which if is listed using "ls -al" shows the following:

```
crw-r--r--  ....
```

The first "c" indicates it is a character driver and can now be accessed by an application.

2.1.2 Using File Descriptors in the Application

An application refers to the device driver by the directory name only twice -- when it is opened and when it is closed. The function to open a device driver is appropriately called "open" and returns a file descriptor. The file descriptor is an unsigned 32-bit value corresponding to a handle to be used for future access of this device (with the specific minor number). Continuing the example above and adding a second device file:

System Setup (script)

```
mknod voice0 -c 100 0
mknod voice1 -c 100 1
/* Contents of an application program */
void main() {
    uint32 fp0 = open("/dev/voice0", RDWR);
    uint32 fp1 = open("/dev/voice1", RDWR);
    ....
}
```

The value fp0 provides access to the device with major number 100 and will pass the minor number "0", the value fp1 provides access to the device with major number 100 and will pass the minor number "1".

2.1.3 File Descriptors as Line Contexts, and Accessing the Device

By associating a unique minor number with a file descriptor, the Linux OS has provided a convenient way to provide a "context" to the API-II. The only problem is that some API-II functions can use either a line context or device context, and the actions taken will depend on which is provided. So how then does does an application communicate either line or device level operations? It is recommended that the minor number correspond to the line context, because most operations occur at the line level. To communicate device level operations, there are at least two possible solutions:

- Create a structure in user space with the content required, indicating a line or device level operation, or
- Create a unique minor number that is interpreted by the driver to mean "device level" operation.

Solution #2 is recommended due to it's simplicity and reduces operations for filling in structure data passed to the kernel driver. An example of #2 implementation is as follows:

System Setup (script)

```
mknod voiceLine0 -c 100 0
mknod voiceLine1 -c 100 1
mknod voiceDevice0 -c 100 255
/* Contents of an application program */
void main() {
    uint32 line0 = open("/dev/voiceLine0", RDWR);
    uint32 line1 = open("/dev/voiceLine1", RDWR);
    uint32 dev0 = open("/dev/voiceDevice0", RDWR);
    ....
}
```

The specific function calls into the Kernel to make this operation work requires a discussion of the function ioctl().

2.2 IOCTL()

The user space function "ioctl()" is the interface into a kernel level driver that supports ioctl() operations. A driver does not necessarily need to support ioctl(), but it is recommended that an API-II driver in the Linux Kernel does. The ioctl() operation is simple, makes the code easy to read, and allows for an almost one-to-one relationship between the parameters passed to the user space function and the function implementation in the Kernel. So it is possible to maintain "significant" (but not all) functionality of the API-II using primarily ioctl(). The remainder of this section will focus on the user space abstraction of some of the API-II functions, the ioctl() function call recommendations, and the driver specific implementation of the ioctl() function (which we will call "voice_ioctl()).

2.2.1 User Space API-II Functions

First, not all API-II functions should be exported to user space. From discussion in Section 1.2.3, the function VpApiTick() does not belong in user space, and from discussion in Section 1.2.4, neither does VpMakeDeviceCtx(), VpMakeLineCtx(), VpMakeDeviceObject(), and VpMakeLineObject(). So it is better to characterize functions that should be exported to user space, and in staying in context of the discussion, those that should be implemented in ioctl().

The following functions should be abstracted by a wrapper for ioctl():

Initalization Functions	Control Functions	Status and Query Functions
VpInitLine()	VpSetLineState()	VpGetEvent()
VpConfigLine()	VpSetLineTone()	VpGetLineStatus()
VpCalCodec()	VpSetRelayState()	VpGetLineState()
VpInitRing()	VpSendSignal()	VpFlushEvents()
VpInitCid()	VpSendCid()	VpGetResults()
VpInitMeter()	VpStartMeter()	VpClearResults()
VpInitProfile	VpSetOption()	
	VpDeviceIoAccess()	

The driver supports ioctl() by registering a driver specific function with the Kernel for handling ioctl() operations (e.g., voice_ioctl()), which will be called when the user application calls ioctl() with a device file descriptor for the particular driver (defined by the major number as discussed in Section 2.1.

There are other functions that a driver can support and accomplish similar things that are done with ioctl(), but the recommendation for API-II is to use ioctl(). It is simple and provides an almost direct abstraction of the existing API-II definitions. The remainder of this section will focus on maintaining the user The name and prototype of the kernel driver implementation of ioctl() is different than the user equivelent. In other words, ioctl() in user space is the same function call for all drivers in the kernel (supporting ioclt()), but each driver implements a unique function called (for ex.) voice_ioctl(), driverx_ioctl(), drivery_ioictl(), and so on. The prototype is also different because one necessary parameter in the user ioctl() function call is the device file descriptor. Once the driver "voice_ioctl()" function is called, the file descriptor is not necessary (e.g., the major number is known by the fact that the driver code was called) and additional information may be required (like the process id that called the driver).

The basic idea of this recommendation is that ioctl() will pass a pointer to a structure that contains the parameters defined in the current API-II definition.

2.3 INIT, OPEN, CLOSE, EXIT

There are four functions supported by every device driver: init(), open(), close(), and exit().

CHAPTER

3 REFERENCES

1. "Linux Device Drivers", 2nd Edition, June 2001